Received 5 April 2024, accepted 13 May 2024, date of publication 16 May 2024, date of current version 23 May 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3401831

RESEARCH ARTICLE

The Impact of Profiling Versus Static Analysis in Precision Tuning

LEV DENISOV[1], GABRIELE MAGNANI[1], DANIELE CATTANEO[1], GIOVANNI AGOSTA[1], AND STEFANO CHERUBIN[2]

[1]Department of Electronics, Information and Bioengineering (DEIB), Politecnico di Milano, 20133 Milano, Italy
[2]Department of Computer Science, Norwegian University of Science and Technology, Gløshaugen, 7034 Trondheim, Norway

Corresponding author: Lev Denisov (lev.denisov@polimi.it)

This work was supported by the European Union's Horizon 2020 Research and Innovation Program under the Marie Skłodowska Curie Grant (APROPOS: Approximate Computing for Power and Energy Optimization) under Agreement 956090.

ABSTRACT Approximate computing techniques, such as *precision tuning*, are widely recognized as key enablers for the next generation of computing systems, where computation quality metrics play an important role. In precision tuning, a trade-off between the accuracy of computations and latency (and/or energy) is established, but identifying the opportunities for applying this approximate computing technique is often challenging. In this article, we compare two different approaches — *worst-case static* annotation and *profile-guided* annotation — and their implications when used in a precision tuning framework. To ensure a fair comparison, we implement the profile-guided approach in an existing tool, TAFFO, and experimentally compare it to the original static approach used by the tool. We validate our considerations using the well-known PolyBench/C benchmark suite, and two real-world application case studies. Our findings demonstrate that the profile-guided approach, fed with reasonable profiling data, in addition to needing less expertise to employ, delivers comparable speedup and better accuracy than the static approach.

INDEX TERMS Approximate computing, compiler, embedded systems, performance evaluation, precision tuning.

I. INTRODUCTION

Approximate computing is an emerging solution to address the increasing energy and time requirements of compute-intensive applications, particularly in embedded systems, but also in HPC scenarios [1], [2]. Several techniques have been developed to trade off the computation accuracy for energy and latency reductions [3], both at the hardware and software levels. Among them, *precision tuning* is notable for its wide applicability [4]. In a nutshell, the technique consists of reducing the precision of the individual operations on real numbers in a given kernel, for instance, from double-precision floating-point to single precision. As an alternative, precision tuning may change the representation format of real numbers, such as transforming floating-point values into fixed-point ones. While it can be done manually, the effort required grows quickly with the complexity of the program being tuned, giving rise to automated precision tuning tools.

Precision tuning tools employ several different strategies ranging from brute-force trial-and-error to sophisticated code analyses to find the optimal precision allocation. Brute-force approaches struggle with scalability as they have worst-case combinatorial complexity with the number of optimized variables. More interesting are the approaches following the code analysis path. Two general strategies dominate this area: static analysis and profiling. On one hand, static analysis typically requires some initial annotation from an application-domain expert, and it searches for the optimal precision allocation without running the program, employing dataflow analysis and symbolic execution. Profiling, on the other hand, only requires some representative datasets, from which it derives the precision allocation by instrumenting and running the program being tuned with the goal of observing the values assumed by each variable throughout the execution.

The associate editor coordinating the review of this manuscript and approving it for publication was Lorenzo Ciani.

There is an argument that the static analysis approach is superior to profiling because it can assert desirable properties such as error boundaries or the absence of overflow. However, in complex real-world software, static analysis often finds solutions that are not practically useful because, for example, too wide worst-case value ranges do not satisfy guarantees in lower precision configurations. Some common obstacles leading to that problem are loops and conditional statements, transcendental functions, or operations that may significantly expand the result range, such as multiplication or division. Because of this problem, the expert involvement needed to annotate the program to be tuned with static approaches tends to be much more significant than just indicating the input ranges. As such, it is common practice to override the results of the static analysis in some areas of the program to arrive at a useful result, to an extent that can undermine the presumed benefit of statically-asserted guarantees. Finally, the assumed expertise in precision tuning may just be absent as reasoning about the effect of lowering precision of complex mathematical calculations is often beyond the typical skill set of an average programmer. If tasked with the problem of annotating a program for precision tuning, such a programmer may resort to some inefficient combination of trial-and-error and manual profiling approaches, potentially erasing some benefits of static analysis.

The analysis approach based on profiling presents its own sets of advantages and challenges. It does not require any expert knowledge from the programmer and is more automated, which makes it easier to use. However, the representativeness of the dataset used for profiling determines the robustness of the found precision allocation. Even if multiple representative datasets are available, profiling may still not cover all the possible corner cases in a complex code base. The found solution can formally guarantee little more than that the precision-tuned program will work with the inputs observed in the profiling stage. At the same time, the profiling approach is unlikely to arrive at the impractically over-generalized solutions that static approaches are prone to. This means that the approximations found with profiling may offer more attractive trade-offs, albeit with fewer guarantees. There is also an argument that profiling may be slower than static analysis for larger programs and datasets. While it may be the case, expert annotation requires human time and attention, which is usually a scarcer resource than pure computation time that can be reduced by using a more powerful computer. In any case, faced with an unfamiliar program to tune, an average programmer is more likely to succeed with a tool based on profiling rather than static analysis, as representative datasets are usually more readily available than expertise in precision analysis.

Based on the discussion above, it comes to concrete experiments to evaluate which approach is likely to be more applicable for precision tuning of real-world applications. Of course, these experiments need to limit the effect to one dimension, in this case, the code analysis approach, keeping other things equal. There is no shortage of precision tuning tools implementing either of the approaches, but, to our knowledge, none of them consistently implements both. For that reason, we needed to choose a tool that already implements one of the approaches, and that is modular enough to allow for the incremental support of the other. Our choice falls on the state-of-the-art tool called TAFFO [5]. It is implemented as a set of compiler transformations for the industry-standard LLVM compiler framework, and it operates on its intermediate representation (LLVM IR), ensuring modularity, possibility of extension, and a degree of language independence. It also comes with the full set of PolyBench/C benchmarks [6] that are annotated by domain experts. These facts give us a fair common ground for the comparison.

In this article, we provide in-depth analysis and comparison between static and profile-based approaches to precision tuning on a well-known benchmark suite. We observe and report the relative error of the computation as the quality metric, and time-to-solution speedup as the main performance indicator. For the less tangible property of usability, we propose the *annotation complexity* metric and examine the effect of the approaches on program build time. Additionally, We contribute an implementation of the profile-guided approach within a state-of-the-art framework previously based only on static analyses and compare it to an independent state-of-the-art hybrid static precision-tuning tool. Finally, we present as complex case studies the process of precision tuning applied to two real-world applications, Field-Oriented Control (FOC) for electric motors and Image scaling with bicubic interpolation.

We conclude that while both approaches result in essentially the same speedup, profile-based analysis may result in smaller error in some cases, given that the profiling data is within the order of magnitude of the production data. Because of the complexity of the required expert intervention in the static approach, we also conclude that the profile-based approach is more likely to be adopted in an industrial scenario due to its lower annotation complexity. However, the static approach shows more robust performance in certain cases, leading to the conjecture that the best way forward is the integration of the two approaches, such as using profiling as a support tool for the static expert annotation.

The rest of this article is organized as follows. In Section II, we discuss related work and the existing precision tuning tools. Section III discusses the theoretical differences between the profile-guided and static approaches. We discuss the concrete implementation of both approaches in Section IV. In Section V, we set up the experimental framework and discuss the experimental results. The case studies are discussed in Section VI. Section VII concludes the article.

II. RELATED WORKS

Precision tuning enables the trade-off between data representation and computation accuracy, with a wide range of other quality metrics — such as data size, time-to-solution, energy-to-solution, and hardware complexity — that could be accounted for in such trade-off. A recent effort to compare a diverse set of products has been presented in HPC MixPBench [7], whose focus is on source-level rewriting precision tuning targeting only IEEE-754 *binary64* and *binary32* data representations [8]. A less detailed yet more comprehensive overview of such approaches already exists in the literature [4]. Most notable ones among the proposed solutions are *FloatSmith* [9], TAFFO [5], and the *Transprecision* approach [10]. Each of these efforts approaches the problem of precision tuning starting from different assumptions about the application to be tuned. *FloatSmith* adopts a trial-and-error methodology to test the impact of a data type alteration on the source code, coupled with static analysis approaches — such as algorithmic differentiation — to reduce the search space. As such, it evaluates solution candidates on a representative test set. TAFFO relies on annotations from the user to provide domain knowledge about the deployment and runtime conditions of the application and then performs a static data-flow analysis to derive a conservative worst-case scenario. Such a worst-case scenario may be orders of magnitude more conservative than the actual values being computed at runtime. Recent works presented on *Transprecision* [10], [11], [12] assume the compiler to be able to adjust the machine precision level. This toolchain-oriented tuning is often driven by the programmer via a set of compiler intrinsics instructions to be manually included in the machine code.

The main issue with static analysis approaches is their pessimistic nature. Small numeric errors often compensate each other when chained together in a sequence of approximated arithmetic operations, keeping the distribution of the overall absolute error distance in the shape of a Gaussian bell whose central value is orders of magnitude far from the worst case [13]. It follows that static analyses may produce results that are too conservative to be considered helpful in some application domains.

On the contrary, the approaches such as trial and error are limited to operating at a coarse granularity by the fact that finer granularity leads to the explosive growth of the application configurations to evaluate (up to $2^n - 1$ combinations in the worst case, given n variables [9]). This leads to the choice between over- or under-allocation of precision and limiting the precision tuning opportunities.

Profiling of floating-point values in a program has been previously proposed in the literature in the form of platform-specific binary instrumentation [14], and of data-type emulation [15]. In this work, we present profiling of individual floating-point values for precision tuning purposes performed via compiler plugins in a modern toolchain and compare it to the static approach implemented in the same tool.

Comparison of the profiled-guided and static approaches is complicated by the fact that most of the static precision-tuning tools discussed in the literature present a very limited set of scenarios on which they can operate, preventing their application to realistic programs. Rosa [16] and the related project Daisy [17] optimize programs written in a Scala domain-specific language and only optimize programs without loops and conditional statements. The work presented in Angerd et al. [18] focuses on precision tuning for the GPU applications that struggle with register pressure bottlenecks. Although the principles presented in the paper are still actual and relevant, this work has been made obsolete by modern-day GPU architectures with native register support for shorter data types. Salsa [19] and Herbie [20] make their goal the improvement of precision without regard for the performance. PrecTuner [21] is a hybrid precision tuning tool that uses static analysis for precision optimization and profiling for building the performance model. Its scope is, however, limited to the transformation from double to float, making it narrowly applicable.

III. THEORETICAL BACKGROUND

Code analyses in precision tuning provide insights into the distribution of values that are being processed at any time within the program. It is possible to use Distribution Arithmetic [22] to analyze the evolution of such distributions over time. However, due to memory and time constraints, such distribution of values is often approximated with an interval $[min; max]$ representing the minimum and maximum value seen within the distribution. Static code analyses are often based on interval arithmetic [23], which provides a framework to conservatively propagate this piece of information through intermediate values. This is often the most computationally intensive analysis in the precision tuning process, and it is fundamental to guarantee the absence of overflow and underflow errors in the computation. Similar, yet simpler, analyses may enforce the absence of cancellation errors. In this work, we focus on the more intensive value range discovery analysis.

One limitation of the static approach to this analysis lies in the conservative propagation of the distribution. Approximating a distribution with its interval does not imply — and rarely corresponds to — having a uniform distribution of values within that interval. As such, static analyses falsely include in the derived internals some values that the computation never reaches. The reasons behind the non-uniformity of the distribution are multiple. The input values can themselves be non-uniform, or it can be the program computation itself that gathers or scatters the data points within the distribution via nonlinear functions.

In case of complex distributions influenced by multiple factors, worst-case analyses in other application domains — such as worst-case execution time studies — recommend a more coarse-grained approach that limits the information granularity to its observable components [24]. Indeed, probabilistic approaches base their decision on the observable events after a sufficiently large number of executions.

Profiling a program can be viewed as a way to sample the probability distributions of its variables and output. While

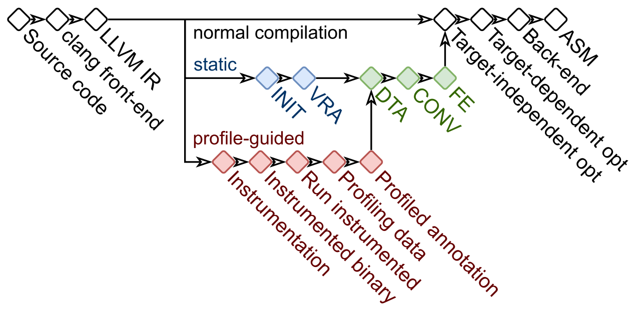


FIGURE 1. Compilation flow with TAFFO in profile-guided mode.

⟨c_ann⟩	→	`__attribute((annotate("` ⟨top⟩ `")))`
⟨top⟩	→	⟨target⟩ ⟨back⟩ ⟨datat⟩
⟨target⟩	→	`target('` ⟨id⟩ `')` | ε
⟨back⟩	→	`backtracking` | ε
⟨datat⟩	→	⟨scalar⟩ | ⟨struct⟩
⟨struct⟩	→	`struct(` ⟨structl⟩ `)`
⟨structl⟩	→	⟨structv⟩ `,` ⟨structl⟩ | ⟨structv⟩
⟨structv⟩	→	⟨datav⟩ | `void`
⟨scalar⟩	→	`scalar(` ⟨datav⟩ `)`
⟨datav⟩	→	⟨range⟩ ⟨error⟩ ⟨final⟩
⟨range⟩	→	`range(` ⟨num⟩ `,` ⟨num⟩ `)` | ε
⟨error⟩	→	`error(` ⟨num⟩ `)` | ε
⟨final⟩	→	`final` | ε

FIGURE 2. BNF grammar of user annotations according to the TAFFO precision tuning tool.

LISTING 1. Original LLVM IR code.

```
...
%var1 = fmul float %9, %12
%var2 = fadd float %10, %14
...
```

LISTING 2. Instrumented LLVM IR code.

```
 @var1_name = constant "uniq1"
 @var2_name = constant "uniq2"
 define void @log_value(%name, %value) {...}
...
 %uniq1  = fmul float %9, %12
 call @log_value(@var1_name, %uniq1)
 %uniq2  = fadd float %10, %14
 call @log_value(@var2_name, %uniq2)
...
```

sampling does not guarantee that all the possible values will be observed, by increasing the number of samples, it is possible to increase the likelihood of the observed probability distributions approximating well enough the actual distribution of the program. This approach can give tighter intervals, and it works for any program, making it a good alternative to the static methods.

IV. STATIC AND PROFILE-GUIDED ANALYSES IMPLEMENTATION

To compare the static and the profile-guided approaches to precision tuning we need to have a tool that is capable of doing both and working with realistically complex programs. Unfortunately, the tools discussed in the literature do not suit this requirement. Because of that, we chose a state-of-the-art static precision tuning framework TAFFO and implemented within it the profile-guided approach. This way, we were able to isolate the effect of the dynamic range analysis models while keeping the rest of the pipeline the same.

A. PREEXISTING STATIC ANALYSIS

As shown in Fig. 1, static TAFFO is composed of five LLVM analyses and transformations, namely: *Initializer* (INIT), *Value Range Analysis* (VRA), *Data Type Allocation* (DTA), *Conversion* (CONV), and *Feedback Estimator* (FE). The INIT pass is responsible for reading the annotations and determining the amount of code affected by the tuning. The VRA pass computes numerical intervals for all annotated variables and any other variables that depend on them. The DTA pass decides the new data type for each intermediate value and variable employing one of two different algorithms: either a greedy algorithm that always selects the fixed point type with the highest valid fractional point position or an algorithm based on a mathematical model that emits mixed precision results. CONV modifies the LLVM IR accordingly to the data type picked by DTA. Finally, FE statically estimates the error.

As initial input to its static analyses, TAFFO requires the programmer to insert annotations into the code, which appear as Clang*annotate* attributes. Annotations provide initial value range information for variables, and as a side-effect, they restrict the scope of the precision tuning optimization. TAFFO annotations follow the formal grammar shown in Fig. 2. On scalar variables or arrays that need to be converted, the annotation contains a **scalar** declaration. Within this declaration, additional optional attributes provide the initial range of the annotated variable (the **range** attribute), and if that range shall be assumed immutable (the **final** keyword). In addition, the **target** declaration may used to create a new VRA and FE analysis entry point.

B. A NEW PROFILE-GUIDED ANALYSIS

Profile-guided TAFFO is an extension for TAFFO that expands its capabilities to precision-tuning based on profiling of the program.

Fig. 1 illustrates the compilation flow for TAFFO in profile-guided mode. The flow is described in the following steps:

1) TAFFO instruments the LLVM IR code to output the values of all floating-point variables (or virtual registers) and compiles a binary.
2) The binary is run by the user, providing the input data if necessary. During its execution, which otherwise proceeds normally, the binary outputs profiling data to a separate file. The binary can be executed an indefinite number of times with different inputs.
3) TAFFO uses the profiling data collected from all previous executions of the binary to determine the value ranges of the aforementioned variables.

4) TAFFO uses the ranges to statically allocate data types in the program.
5) The program goes through optimization and backend compilation; the resulting assembly code is in mixed-precision.

Profile-guided TAFFO is based on the instrumentation of the LLVM IR code. The code changes done by the instrumentation are shown on listing 1 (original) and listing 2 (instrumented), the changes are highlighted in red. The LLVM IR file is first prepared by assigning unique names to all floating-point LLVM IR registers and these names are saved into global constants. The logging function is also inserted into the LLVM IR file. This function accepts the name of the register and its value. It performs either logging directly into a file or updates an in-memory structure that contains minimum and maximum values for the register name, and this structure is flushed into the log file later. After that the calls to the logging function are inserted for every floating-point register, passing the global constant with the register's name and the value of the register. The instrumented code is then compiled into a binary without optimization and fused operations to preserve as much of the same structure as the LLVM IR code (clang flags *-O0 -fno-slp-vectorize -fno-vectorize -ffp-contract = off*). The user can run the binary with one or more datasets, collecting the logged values in trace files, which are then used in the precision tuning process.

The goal of this step is the collection of information about the value ranges. As such, the lack of compiler optimization does not matter as it does not affect the correctness of the computation. The fact that the virtual LLVM IR registers are lowered into the physical registers during the compilation process also does not affect the validity of the instrumentation as what is logged is the result of the corresponding LLVM IR operation regardless of how it is physically computed. Indeed, since both profiling and the precision-tuning steps are done at the LLVM IR level, this approach allows to precisely match logged values to the corresponding instruction at the precision-tuning step later. In case the program relies heavily on external libraries for floating-point computations and the source code of these libraries is not available during the compilation, the instrumentation process will not be able to track the value ranges through the library code. This, however, is not a concern as the goal of collecting the intervals is the precision-tuning step that does require type changes during the compilation and cannot function in the absence of the source code. A possible alternative approach is binary instrumentation and precision tuning, which is outside of the scope of this study as it is challenging to implement in a generalizable and reliable way.

The performance overhead of the instrumentation was not evaluated at this stage and is left for future work. In the case of bigger programs, there may arise the need for optimization of the logging procedure such as limiting its application to the smaller parts of the code, skipping logging iterations, and inlining the logging operations. While the performance of this step is of interest, it is not the main concern as it is performed only once per program and it can be executed on any sufficiently powerful machine.

LISTING 3. A cluster of memory operations with the same range.

```
%DY = alloca float
%"a::var123" = load float, float* %DY
store float %"a::var109", float* %DY
%DY2 = bitcast float* %DY to i8*
%"a::var109" = fdiv float 1.000000e+00, %"a::var108"
```

The process of assigning TAFFO metadata ranges from the trace file consists of a few steps. First, a dependency graph [25] representing memory operations is built by analyzing the LLVM IR of the program. Initially, the graph is seeded with one node for each instruction that allocates a buffer in memory (*alloca*, *malloc*, etc.). Then, all the uses of each buffer that do not change its value (*load*, *store*, usage as an argument to a function call, pointer operations, etc.) are added to the graph as new nodes, connected with an arc to the node that allocated the buffer. At the end of this process, each connected component (or *node cluster*) in the graph determines a set of virtual registers that shall share the same range. An example of such a cluster can be found in Listing 3. After building the node clusters, the trace files are parsed and the minimum and maximum values witnessed during execution are extracted for every register in the trace file. If a register belongs to a node cluster, these values are shared with all other registers in the cluster. At this point, the pass writes the ranges just computed into the LLVM IR. If the user passes more than one trace file, the ranges are computed over all files combined. After all the registers have been associated with a range, the rest of the TAFFO pipeline proceeds through the DTA, CONV, and FE passes, followed by the generation of machine code and linking to produce a working executable. The implementation of these stages is left unchanged from the original TAFFO code [5].

V. EXPERIMENTAL COMPARISON

In this section, we compare the profile-guided and the static approaches on the well-known PolyBench/C benchmark suite, version 4.2.1. We evaluate both approaches in terms of annotation complexity, robustness to input data variation, generated code, and the speedup. In addition to the static precision tuning implemented in TAFFO, we provide the comparison in terms of accuracy and speedup to the state-of-the-art tool PrecTuner [21].

PolyBench [6] features computational kernels from disparate domains such as physics simulation, linear algebra, image processing, statistics, and dynamic programming. The particular choice of the algorithms in the benchmark suite is not very important to our study as we are mostly interested in the various computationally intensive kernels present in those algorithms, such as matrix multiplication, that are general enough to be found in almost any compute-intensive application ranging from image processing to machine learning, AI, etc.

For running the benchmarks, we use an embedded platform, specifically an STM32L010 microcontroller with Arm

Cortex-M0+ CPU (32 MHz), 128 Kbytes of Flash memory and 20 Kbytes of SRAM. Since this device lacks hardware support for floating-point, we use software implementation of floating-point operations with the *-msoft-float* compilation flag. The choice of an embedded platform for the evaluation is explained by the fact that these are the most likely candidates to benefit from conversion from floating-point to fixed-point due to the slower FPU implementation or lacking hardware support for floating-point altogether. Additionally, these platforms are widespread and of rising importance, due to cost, energy efficiency, and size reasons. The same approach can be applied on desktop machines or HPC nodes, however, the benefits on such architectures are often constrained by the specific CPU architecture of the machine [5].

Due to the low amount of RAM available on the microcontroller, PolyBench was configured with the dataset size *MINI_DATASET*. For this same reason, an additional adjustment was required for the *deriche* benchmark, where the size arguments were reduced from $W = 64, H = 64$ to $W = 32, H = 32$. Benchmarks *floyd-warshall* and *nussinov* were not evaluated as they implement integer algorithms.

The time measurements were averaged over 10 executions of the benchmark. Since the benchmarks were run on a microcontroller without an operating system, the execution time had essentially no variation between each execution.

For error measurements, we used Mean Relative Error, which we calculated by the following formula (1):

$$\mathrm{MRE} = \frac{1}{n}\sum_{t=1}^{n}\frac{p_t - a_t}{p_t} \tag{1}$$

where p_t is the floating-point result and a_t is the approximate result, and n is the output size. For the increased resolution, we use double precision as a reference for MRE. Speedup is computed by the formula (2)

$$S = \frac{T_{float}}{T_{approx}} \tag{2}$$

where T_{approx} is the execution time of the approximated program and T_{float} is the execution time of the floating-point program (*float* was used as the reference for speedup).

In the speedup charts, we mark the 1 value (no speedup) as a vertical red line. Similarly, we mark with a vertical red line the value of 1% (10^{-2}) for the relative error charts as a conservative estimate of the acceptable relative error threshold. In comparison, in the previous works, the acceptable error of approximated applications is set as high as 10% for some applications [26].

The average value on charts is computed as geometric mean for the error charts and build overhead and as arithmetic mean for the rest.

A. ANNOTATION COMPLEXITY

To measure the developer effort to add the required annotations to a program, we need to introduce a notion of *annotation complexity*. To this end, we leverage concepts from the well-known cognitive complexity [27], adapting them to the declarative nature of TAFFO annotations. Much like in cognitive complexity, we measure effort in unitary increments, which are provided by each choice or piece of information that the developer needs to collect to build the annotation. As such, each annotation provides one increment, since the developer needs to identify the need to add it. The **range** attribute is the key source of effort, as it requires the identification of the input range, thus leading to a +2 increment. The addition of the **final** keyword is another conscious choice, with a unitary increment. Finally, the **target** annotation identifies an instance of the *VRA* process, which can be compared to a function or method in the context of cognitive complexity, thus leading to a further unitary increment of annotation complexity. Therefore, we define the annotation complexity as (3):

$$\#\,annotations + 2 \times \#\,range + \#\,final + \#\,target \tag{3}$$

TAFFO provides the expert annotations for the PolyBench/C suite, so we calculated the annotation complexity of these benchmarks. Table 1 reports the static annotation complexity of the PolyBench/C suite (excluding integer benchmarks where the annotations are not needed). It can be seen that for each benchmark, 3 to 14 variables need to be annotated, with an annotation complexity ranging from 7 to 31. Even relatively simple benchmarks require a significant developer effort to annotate, with some bench-

TABLE 1. Annotation complexity for PolyBench/C.

Benchmark name	Float variables	Annotations	Annotation Complexity
2mm	25	9	20
3mm	23	9	23
adi	18	10	27
atax	14	7	20
bicg	18	7	19
cholesky	8	6	14
correlation	16	8	23
covariance	13	6	22
deriche	22	14	31
doitgen	11	6	19
durbin	11	7	20
fdtd-2d	19	6	15
gemm	19	7	10
gemver	36	14	14
gesummv	23	9	30
gramschmidt	14	6	20
heat-3d	8	5	22
jacobi-1d	9	3	15
jacobi-2d	9	4	7
lu	8	6	11
ludcmp	17	9	20
mvt	22	7	14
seidel-2d	9	7	19
symm	20	7	9
syr2k	19	7	14
syrk	15	6	12
trisolv	11	5	15
trmm	12	5	14
Average	16.1	7.2	17.66

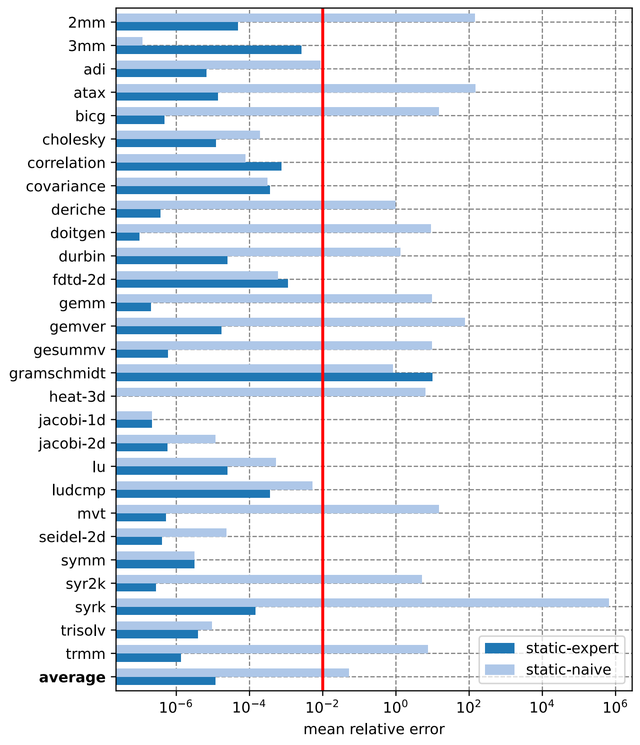


FIGURE 3. Mean relative error of expert and naive static annotations.

marks requiring dozens of different items. The simplest benchmark that can be precision-tuned with one annotation on one input variable would score 3 points in the annotation complexity. Thus, the benchmarks in the PolyBench/C suite require 2× to 10× the number of annotations the simplest benchmark needs, indicating the necessity of significant expert involvement in the annotation process.

The profile-guided approach does not need any annotations on the source code, so its annotation complexity is 0 for all the benchmarks.

B. NAIVE STATIC ANNOTATION

To further evaluate how important expert knowledge is in the static annotation process, we annotated the PolyBench benchmarks with a naive approach and compared the resulting accuracy of the precision-tuned programs. The naive approach consists of only annotating the input variables with the ranges obtained from the input dataset by finding the minimum and maximum values. This is a reasonable attempt to annotate the program from someone who is not an expert in precision tuning.

Fig. 3 demonstrates the error comparison between the expert annotation and the naive approach to annotation. As is evident from the chart, the naive approach is not viable for the majority of the programs. In 15 out of 28 benchmarks the naive approach resulted in an error significantly bigger than the 10^{-2} threshold, while it was the case for only 1 benchmark with expert annotations. Only in 6 benchmarks the naive annotation resulted in an error lower or equal to expert annotation. Lower error in these cases is likely because the expert annotation was done with some safety margin to allow for the input range variation.

C. ROBUSTNESS TO INPUT DATA VARIATION

Instead of expert knowledge, the profile-guided precision tuning needs input data to be trained on. To evaluate the effect the variation of the input data has on the tuning quality, we profile benchmarks on the data with random noise added to the input. The amplitude of the noise would define the range in which the input data of the benchmark can vary from the one used in the evaluation stage. Since benchmark *trisolv* involves division by the input values, we could not reliably randomize the data as sometimes it resulted in division by a value close to 0 which leads to unrealistically large results. Therefore we exclude the benchmark from this evaluation and consider it as having a worse error for the purpose of comparison with the static approach. This is mostly an artifact of how we randomize the data, and it should not affect applications with realistic datasets.

In Fig. 4 we compare the relative error for the static annotation by an expert and two modes of profile-guided annotation: without the noise in the input (the best possible case), and with the random noise $\pm 100\%$ of the original values. We have chosen random noise amplitude $\pm 100\%$ as an example of a situation when the profiling data is considerably different from the production data, but is comparable to it. Note that we explore the higher levels of noise later in this section in Fig. 6. As an additional baseline for the static analysis, we provide the comparison with the PrecTuner within the same chart.

Overall, all three TAFFO-based approaches have a mean relative error below 1% (10^{-2}) in all benchmarks apart from *gramschmidt*, as it implements a numerically unstable algorithm that is not a good target for precision tuning optimization. PrecTuner in most cases has the lowest error, which is expected since it only converts programs down to float precision, while the other approaches operate with fixed-point mixed precision. In case of *bicg* and *symm* PrecTuner has a significantly higher error than the alternatives, indicating the possible instability of the approach. The profile-guided approach even with the noise in the training data has better or equal mean relative error than the static annotation in all but 3 benchmarks: *deriche*, *durbin*, and *ludcmp*. *Deriche* implements an image-processing algorithm, and its input is a grayscale image with all pixel values in the range [0, 1]. The added random noise made some values go outside this range, leading to a suboptimal type allocation. In the case of both *durbin* and *ludcmp* the randomization process increased the dynamic range of the variables leading to fewer bits being allocated to the fractional part, losing accuracy in the result. In all these benchmarks, the difference between the static annotation and the profile-guided without the noise was already very small, which indicates that there was not much room for optimization in these algorithms for the profile-guided approach to begin with. This could

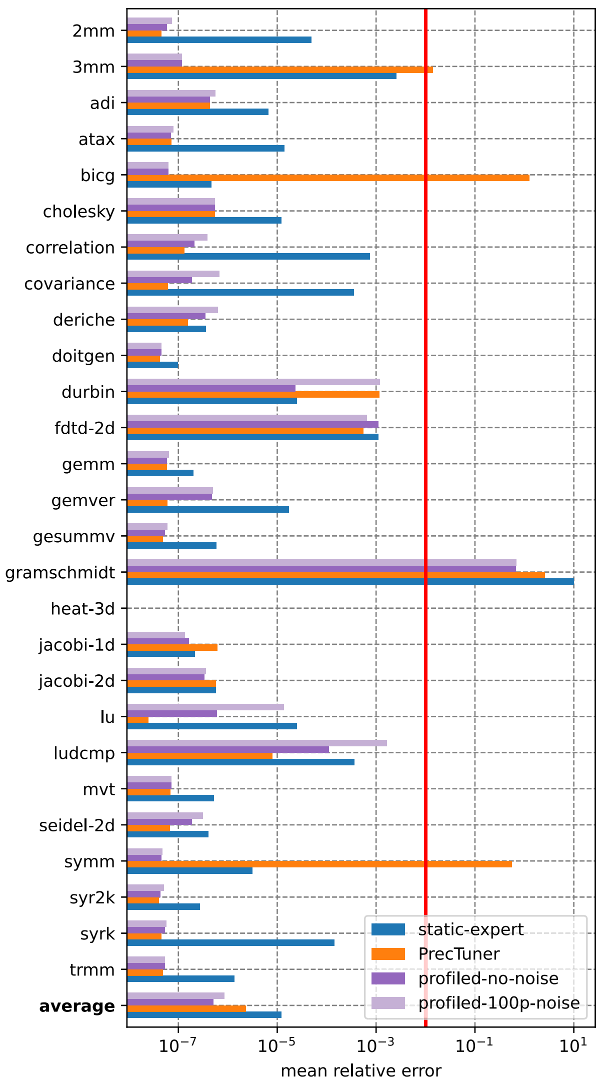


FIGURE 4. Mean relative error of expert annotation vs profile-guided annotation trained on input with and without noise.

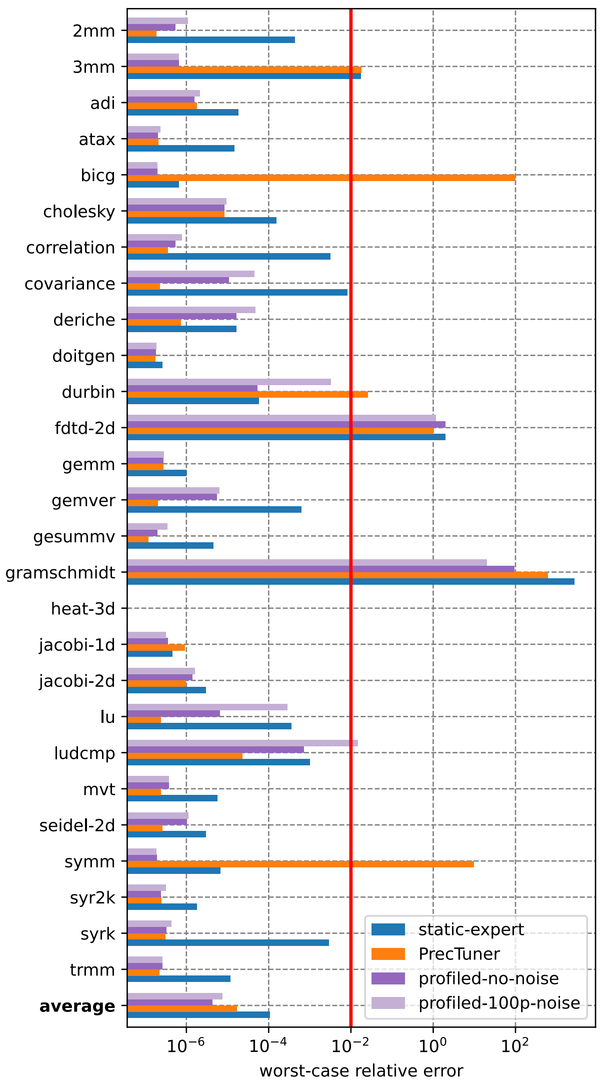


FIGURE 5. Worst-case relative error of expert annotation vs profile-guided annotation trained on input with and without noise.

mean that these algorithms were not good targets for the profile-guided approach as most of the knowledge about the dynamic ranges throughout the program can be captured with only a few static annotations. In most cases, adding the noise resulted in less than 1 order of magnitude increase in the mean relative error in comparison to the profiling without the noise, except for *durbin*, *lu*, and *ludcmp*. In these benchmarks, the noise introduced enough difference in the dynamic ranges to significantly affect the type allocation. In 15 out of 28 benchmarks, the difference in error between the static and profile-guided annotation is an order of magnitude or higher. This can be explained by the profile-guided approach achieving tighter ranges, leading to more bits being assigned to the fractional part, making it more accurate. In *heat-3d* the error is 0 for all approaches because this benchmark is very amenable to conversion into fixed-point arithmetic. Overall, the profile-guided approach in ideal circumstances achieves much higher accuracy than the static approach: in more than 80% of benchmarks the error for profile-guided tuning is less than 10^{-6}, while static tuning is below this threshold in only 40% of benchmarks.

In addition, we include the chart showing the worst-case relative error in Fig. 5. The worst-case relative error follows the same general pattern as the mean relative error but is 1–2 orders of magnitude higher. In case of *fdtd-2d*, *gramschmidt*, and *ludcmp* for profile-guided approach and *3mm* for static approach, the worst-case relative error exceeds the 1% threshold. In general, however, the worst-case relative error stays within reasonable margins and is similar between the static and profile-guided approaches, leading to the conclusion that it depends more on the program being precision-tuned rather than the precision-tuning approach.

To further explore the effect of the input data variation on the quality of profile-guided tuning, we tested the benchmarks with the random noise added to the input during the profiling, with an amplitude varying from 0% to $\pm 1000\%$. The result can be seen in Fig. 6: most of the benchmarks are resilient to a noise amplitude as high as $\pm 10\% - 200\%$. The

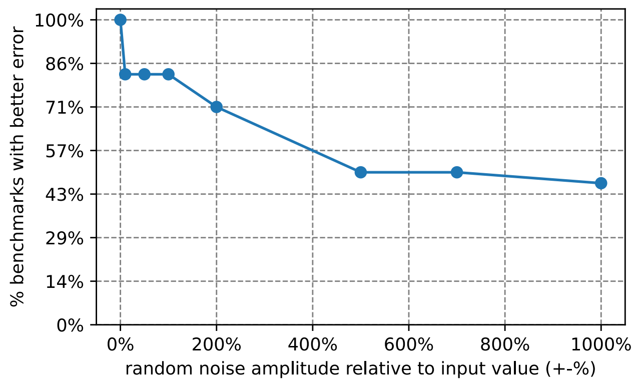


FIGURE 6. **Percentage of benchmarks where the profile-guided approach has better error as a function of noise in the data used for tuning.**

point at which half of the benchmarks have a worse error with profile-guided tuning than with static tuning is about ±1000% noise amplitude, which shows that getting profiling data within an order of magnitude of the real distribution can be good enough for many applications. This result shows that profile-guided tuning is applicable even in the case when the exact distribution of the input data might be hard to determine precisely at the tuning stage.

D. SPEEDUP

To see the effect the two approaches have on execution time, we ran the benchmarks on an embedded platform and compared the speedups each approach has relative to the unmodified float version. The speedup did not significantly change with the noise level of the training data, so we compare directly the expert-annotated versus profile-guided without noise. As with the error discussed in section V-C, we additionally compare the speedup to PrecTuner.

Fig. 7 summarizes the findings. For all benchmarks, except *heat-3d*, TAFFO-based approaches show speedup higher than 1, while PrecTuner never exceeds 1. This is explained by the fact that PrecTuner only converts programs from double to float, and we used float as the base type (an option in PolyBench) for calculating the speedup. This limitation prevents PrecTuner from addressing problems that benefit from data types other than the standard IEEE-754 ones, which are increasingly common in modern scenarios. It can be seen that in 26 out of 28 benchmarks, the profile-guided approach achieves the same speedup as the static approach. In most cases, the speedup is in the range of 2-3×, with the occasional 5-10× speedup. There are 2 benchmarks where speedup obtained with the profile-guided approach is significantly better than with the static approach: *deriche* and *heat-3d*. In the case of *heat-3d* profile-guided approach achieves a 10× speedup, while the static approach slows down the program. The slowdown can be explained by the static approach producing overly pessimistic ranges for the intermediate results, leading to significant parts of the program using conversions between the fixed-point and floating-point formats, leading to the slowdown. In *deriche* static approach gives an insignificant speedup, while

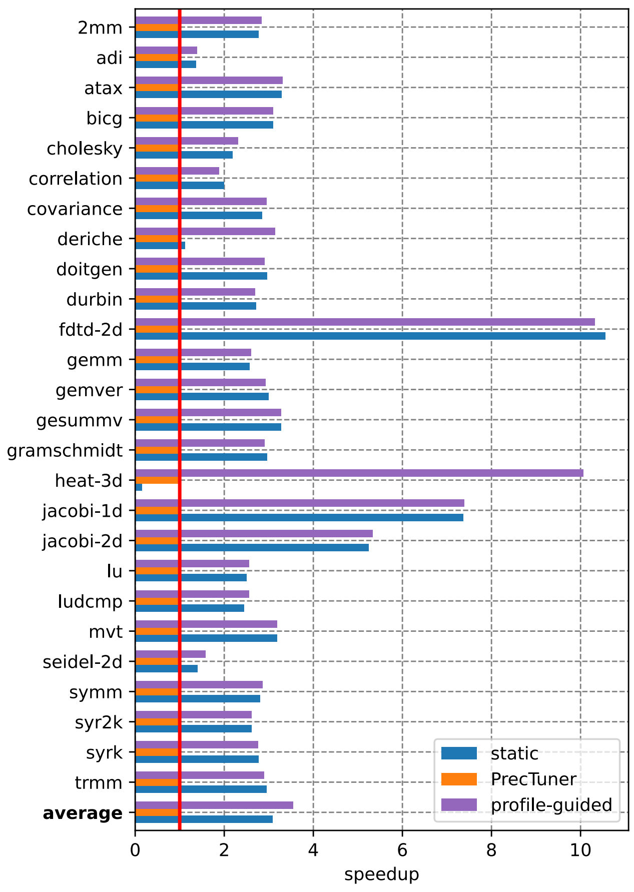


FIGURE 7. **Speedup of static and profile-guided approaches.**

the profile-guided approach results in a speedup of more than 3×. The reason for this is that the static approach was not able to derive tight enough ranges amenable for converting the storage arrays to fixed-point, leading to a cast to floating-point being inserted every time the values were read or stored. The profile-guided approach did not have this problem and converted the data arrays to fixed-point throughout the whole benchmark, reducing the total number of floating-point operations.

E. GENERATED CODE

To better understand the difference between static and profile-guided annotation we analysed the code generated by each approach. We provide a per-benchmark breakdown of the number of floating-point operations (Fig. 8) and shift operations (Fig. 9) in the benchmark kernels. We use the number of floating-point operations and shifts as useful proxies for reasoning about the observed performance. In particular, the number of floating-point operations is a proxy for the execution time of the floating-point executable, as is often the case on embedded platforms without hardware support for floating-point. Instead, the shift operations are introduced as a result of the precision-tuning as a part of the conversion between different fixed-point formats, and are a proxy for the execution time for the reduced-precision executable. Indeed, a higher number of

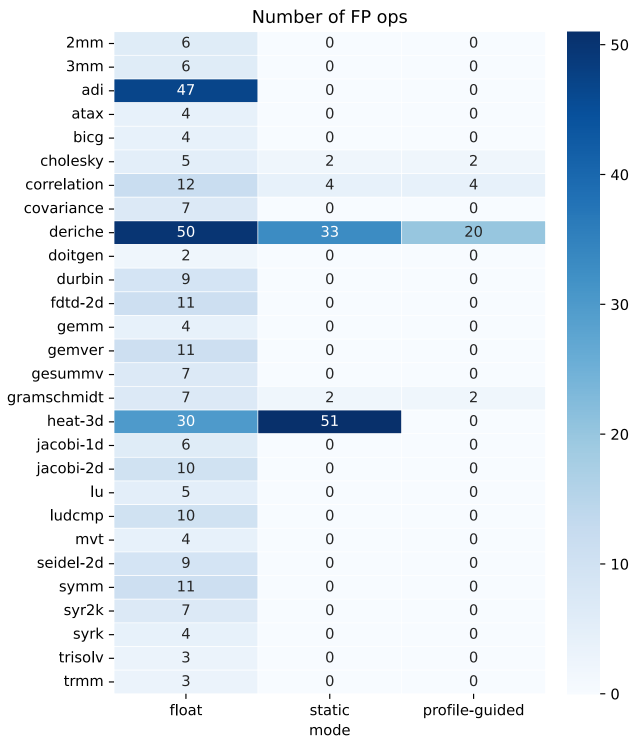


FIGURE 8. **Number of floating-point operations in the benchmark kernels in the original (float) code, and the code optimized with static and profile-guided approaches.**

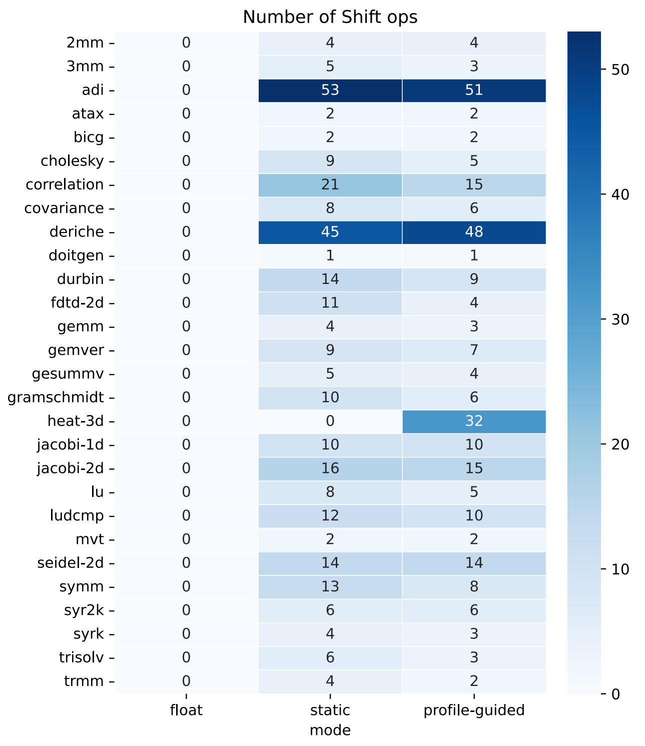


FIGURE 9. **Number of shift operations in the benchmark kernels in the original (float) code, and the code optimized with static and profile-guided approaches.**

shift operations means that the code is running on a more diverse mix of fixed-point types, and this might also cause a slowdown. Conversion between floating-point and fixed-point formats itself introduces a floating-point operation, either a multiplication or division by a constant, which explains why in some cases the transformed code contains more floating-point operations than the original. It needs to be noted that the measures in figures 8 and 9 are computed by inspecting the code, and they do not directly reflect the number of operations being executed at runtime, which in some cases can be significantly different due to loops or conditional statements.

The high number of floating-point operations in *heat-3d* in Fig. 8 confirms the hypothesis from Section V-D that the slowdown in the static approach was caused by the conversions between the fixed-point and floating-point formats. The profile-guided approach also uses a diverse mix of data types, but it manages to keep more of the code in the fixed-point format, as can be seen from the high number of shift operations. The same goes for *deriche*: the statically tuned version has more floating-point operations than the profile-guided tuned one, keeping the speedup much lower.

In general, as it can be seen from Fig. 8 and Fig. 9, both profile-guided and static approaches generate a similar number of floating-point and shift operations. This results in an almost identical speedup between the two approaches, with a small variation that can be attributed to the fact that the profile-guided approach has a more uniform allocation of fixed-point types, resulting in fewer conversions. Therefore, the significant improvements in the error obtained by the profile-guided approach are entirely due to the better positioning of the decimal point in the fixed-point variables. By being able to derive tighter ranges at every step of computation, more fractional bits are preserved, resulting in higher accuracy computations.

F. BUILD TIME OVERHEAD

Adding precision tuning as a build step increases the compilation time no matter the approach. Profile-guided tuning introduces the building of an instrumented binary and running it on a test dataset. Because of that, the profile-guided tuning is likely to have a higher overhead than the static approach for all but the simplest cases. PrecTuner which uses a hybrid approach that involves both profiling and static analysis is expected to add the most overhead. Indeed, Fig. 10 shows that for the tested benchmarks, the overhead of the static approach is on average $4\times$, for the profile-guided approach, it is $8\times$, and for PrecTuner it is more than $32\times$. It should be noted that these numbers can only be used for a qualitative comparison as the tool implementations were not optimized for the build time.

The overhead of the static approach is a function of the code complexity of the program being tuned. For the profile-guided approach in addition to the program complexity, the overhead not only depends on the code complexity but also on the amount of the profiling data used. In general, for every floating-point operation in the run time, profiling introduces two additional operations that keep track

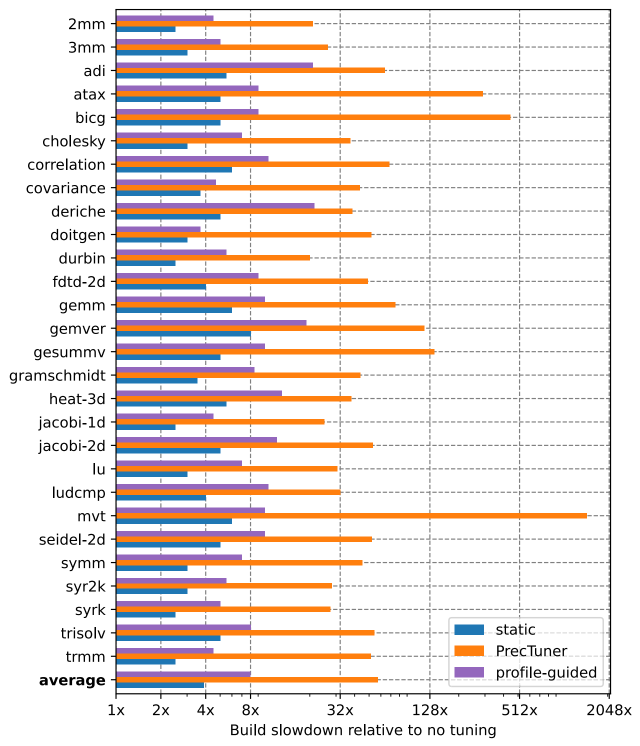


FIGURE 10. Build slowdown of static and profile-guided tuning relative to no tuning build.

of the minimum and the maximum values observed at the code location.

For complex programs operating on large datasets, profile-guided or hybrid tuning may lead to a significant increase in build time, potentially making the static tuning a more viable solution. However, build time is often less of a concern in optimization programs for production. Since the profile-guided tuning only uses instrumentation during the build process and the final binary does not contain any of the instrumentation code, it does not affect the run time of the program in production. In addition, saving the development time spent on the manual annotation of the program might be a more significant factor in increasing development productivity. Given these considerations, the difference in build time is not likely to influence the choice between the static and the profile-guided tuning.

VI. CASE STUDIES

To see the effect of static and profile-guided precision tuning on real applications, we apply them to the programs implementing Field-Oriented Control and Bicubic image scaling.

A. FIELD-ORIENTED CONTROL

Field-Oriented Control (FOC) [28] is a widely used advanced control method for AC induction motors. FOC proves highly advantageous in scenarios requiring optimal motor performance to attain a precise torque or rotation velocity. These motors have drive coils fixed on the stator while the rotor rotates freely around them. The FOC control equations output the voltage that needs to be applied to the coils at a specific time to obtain the desired rotation rate. The FOC algorithm is highly compute-intensive, as it relies on the use of trigonometric functions that are not normally available as elementary operations in microcontrollers.

TABLE 2. Experimental results on the FOC case study.

Solution	Execution time [ms]	Speedup (%)	Mean Rel. Error	Annot. Complexity
Baseline	839.8	100	0	0
Static	285.9	293	5.9×10^{-8}	14
Profile-guided	141.1	594	7.9×10^{-8}	0

The inputs of the algorithm are signals from sensors that measure the rotor flux angle. While these are expressed as floating-point numbers in reference implementations of FOC, the algorithm is proven to be amenable to precision tuning, reducing the computation to fixed point arithmetic [29], with the reported speedup of up to 278%.

To compare static precision tuning with the profile-guided approach, we optimized the same FOC algorithm reported in [29] using the profile-guided approach, and tested it on the hardware described in Section V. We used different datasets for profiling and testing. Table 2 compares the unmodified floating-point code and the statically and profile-guided tuned versions. Due to the use of different hardware than what was reported in the original study, we observed a slightly higher speedup for the static precision tuning (293% vs 278% in the original). This, however, does not affect the validity of our study since we use the same hardware to test the profile-guided tuning. The profile-guided approach achieves almost a 6× speedup while achieving a similar mean relative error to the static approach ($< 10^{-7}$). The improved speedup in comparison to the static annotation is explained by the profile-guided approach finding more floating-point variables to convert that were missed in the process of expert annotation. The annotation complexity for the static approach is 14, while the profiling approach does not require any annotations. This demonstrates that in real-world scenarios, the profile-guided precision-tuning approach may be more useful since it requires less expertise from the programmer and in some cases provides better speedup and error.

B. BICUBIC IMAGE SCALING

Bicubic interpolation [30] in image processing is an image scaling algorithm that determines the pixel value from the weighted average of the 16 closest pixels. It employs cubic polynomials to increase the quality of the scaled image, but the algorithm is characterized by a considerable amount of computations.

We optimized this algorithm with both static and profile-guided approaches and the results of the comparison are shown in Table 3. This case is difficult for the static method because of the long chains of computations,

TABLE 3. Experimental results on the Bicubic image scaling case study.

Solution	Execution time [ms]	Speedup (%)	Mean Rel. Error	Annot. Complexity
Baseline	70532.7	100	0	0
Static	83057.8	85	2.3×10^{-5}	31
Profile-guided	8866.4	795	0	0

making the static analysis conservatively expand the potential dynamic intervals of the intermediary results. Past the point where they can be contained in the fixed-point format, the variables are switched to the floating-point format for some parts of the computation. This is inefficient, and it leads to the slow-down relative to the reference implementation in floating-point, with the tuned program operating at the 85% speed. The overestimation of the dynamic intervals also leads to the under-allocation of the fractional part in the fixed-point variables, leading to some accuracy loss. Even though the static tuning of this program was not a success, the annotation still required significant effort, with the annotation complexity of this code amounting to 31. On the other hand, the profile-guided approach shows a significant speedup of 795% with no accuracy loss. This case study demonstrates that there are cases where the profile-guided approach is clearly preferable over the static analysis approach.

VII. CONCLUSION AND PERSPECTIVES

In this work, we explored the choice between static and profile-guided analysis in the construction of automated precision tuning tools.

We discussed the advantages and disadvantages of both approaches, arguing that while the static approach theoretically provides more guarantees, the profile-based approach may be more applicable in the industry due to the complexity of the static annotation process and its tendency to favor overly conservative solutions that do not provide enough room for the approximation.

Starting from a static-analysis-based state-of-the-art precision tuning tool leveraging the LLVM compiler framework, TAFFO, we extended it to implement the profile-guided analysis. The resulting tool allows us to directly compare static vs profile-guided analysis techniques, with all other aspects being managed with the same effectiveness. The experimental analysis was performed on the PolyBench/C benchmark suite on an ARM-based embedded platform.

We started with an evaluation of the effort an expert would need to make to annotate a program for the static analysis. To that end, we introduced the *annotation complexity* metric, and using it showed that the annotation complexity of PolyBench/C benchmarks is 2× to 10× higher than the simplest program would require.

We compared the resulting accuracy of programs annotated by an expert against those annotated with a naive approach. The result showed that the programs annotated with the naive approach exceeded the 1% relative error threshold in more than 50% of benchmarks, and had higher relative error than the programs annotated by an expert in general.

We evaluated the effect of input data variation on the robustness of the profile-based approach by adding random noise to the profiling inputs of the programs. We showed that with low levels of noise, the profile-based approach has lower relative error than the static approach. The point at which 50% of the benchmarks perform better with the static approach than the profile-based one is at the maximum amplitude of random noise at about 10× the original values.

We showed that the speedup of the static and the profile-based approaches is approximately the same, with some variance that is explained by the tightness of the variable ranges found by each approach. We confirm our observations about speedup by looking at the instruction allocation of the tuned programs, in particular the number of floating-point operations and the number of shift operations.

We also compared our implementation to another state-of-the-art implementation of a hybrid static approach, PrecTuner, which showed comparable error but no speedup because of the limitation of only using floating-point types. We explored the effect the three tuning approaches have on the build time of the program and concluded that although the overhead might be significant, it is not likely to affect the adoption of these approaches.

The case study done with the real application Field-Oriented Control (FOC) has shown that the profile-based approach can achieve 2× higher speedup than the static approach at approximately the same error cost. It is explained by the profile-based approach finding optimization opportunities missed by the expert in the annotation process. Another real case study, Bicubic image scaling, proved to be challenging for the static approach, showing slowdown relative to the baseline, but the profile-based approach still achieved almost 8× speedup with 0 error.

Future research directions point towards the integration of the two approaches, as specific benchmarks proved more amenable to one or the other approach. As one of the avenues of the integration of the two approaches, we envision using profiling as the support in the expert annotation.

REFERENCES

[1] W. Liu, F. Lombardi, and M. Shulte, "A retrospective and prospective view of approximate computing," *Proc. IEEE*, vol. 108, no. 3, pp. 394–399, Mar. 2020.

[2] H. J. Damsgaard, A. Ometov, and J. Nurmi, "Approximation opportunities in edge computing hardware: A systematic literature review," *ACM Comput. Surveys*, vol. 55, no. 12, pp. 1–49, Mar. 2023.

[3] P. Stanley-Marbell, A. Alaghi, M. Carbin, E. Darulova, L. Dolecek, A. Gerstlauer, G. Gillani, D. Jevdjic, T. Moreau, M. Cacciotti, A. Daglis, N. E. Jerger, B. Falsafi, S. Misailovic, A. Sampson, and D. Zufferey, "Exploiting errors for efficiency," *ACM Comput. Surveys*, vol. 53, pp. 1–39, Jul. 2020.

[4] S. Cherubin and G. Agosta, "Tools for reduced precision computation: A survey," *ACM Comput. Surveys*, vol. 53, no. 2, pp. 1–35, Apr. 2020.

[5] D. Cattaneo, M. Chiari, N. Fossati, S. Cherubin, and G. Agosta, "Architecture-aware precision tuning with multiple number representation systems," in *Proc. 58th ACM/IEEE Design Autom. Conf. (DAC)*, Dec. 2021, pp. 673–678.

[6] L.-N. Pouchet. (2012). *Polybench: The Polyhedral Benchmark Suite*. [Online]. Available: http://www.cs.ucla.edu/pouchet/software/polybench

[7] K. Parasyris, I. Laguna, H. Menon, M. Schordan, D. Osei-Kuffuor, G. Georgakoudis, M. O. Lam, and T. Vanderbruggen, "HPC-MixPBench: An HPC benchmark suite for mixed-precision analysis," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, Oct. 2020, pp. 25–36.

[8] *IEEE Standard for Floating-point Arithmetic*, IEEE Standard 754-2019, 2019, pp. 1–84.

[9] M. O. Lam, T. Vanderbruggen, H. Menon, and M. Schordan, "Tool integration for source-level mixed precision," in *Proc. IEEE/ACM 3rd Int. Workshop Softw. Correctness HPC Appl.*, Nov. 2019, pp. 27–35.

[10] G. Tagliavini, S. Mach, D. Rossi, A. Marongiu, and L. Benini, "A transprecision floating-point platform for ultra-low power computing," in *Proc. Design, Autom. Test Eur. Conf. Exhibition*, Mar. 2018, pp. 1051–1056.

[11] F. Montagna, S. Mach, S. Benatti, A. Garofalo, G. Ottavi, L. Benini, D. Rossi, and G. Tagliavini, "A low-power transprecision floating-point cluster for efficient near-sensor data analytics," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 5, pp. 1038–1053, May 2022.

[12] G. Ottavi, A. Garofalo, G. Tagliavini, F. Conti, A. D. Mauro, L. Benini, and D. Rossi, "Dustin: A 16-cores parallel Ultra-Low-Power cluster with 2b-to-32b fully flexible bit-precision and vector lockstep execution mode," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 70, no. 6, pp. 1–14, Jul. 2023.

[13] P. Hennig, M. A. Osborne, and M. Girolami, "Probabilistic numerics and uncertainty in computations," *Proc. Roy. Soc. A, Math., Phys. Eng. Sci.*, vol. 471, no. 2179, Jul. 2015, Art. no. 20150142.

[14] M. O. Lam. (2016). *Shadow Value Analysis Library*. [Online]. Available: https://www.osti.gov/biblio/1351612

[15] G. Flegar, F. Scheidegger, V. Novaković, G. Mariani, A. E. Tomás, A. C. I. Malossi, and E. S. Quintana-Ortí, "FloatX: A C++ library for customized floating-point arithmetic," *ACM Trans. Math. Softw.*, vol. 45, no. 4, pp. 1–23, Dec. 2019.

[16] E. Darulova and V. Kuncak, "Towards a compiler for reals," *ACM Trans. Program. Lang. Syst.*, vol. 39, no. 2, pp. 1–28, Mar. 2017.

[17] E. Darulova, E. Horn, and S. Sharma, "Sound mixed-precision optimization with rewriting," in *Proc. ACM/IEEE 9th Int. Conf. Cyber-Phys. Syst. (ICCPS)*, Apr. 2018, pp. 208–219.

[18] A. Angerd, E. Sintorn, and P. Stenström, "A framework for automated and controlled floating-point accuracy reduction in graphics applications on GPUs," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 4, pp. 1–25, Dec. 2017.

[19] N. Damouche and M. Martel, "Mixed precision tuning with salsa," in *Proc. 8th Int. Joint Conf. Pervasive Embedded Comput. Commun. Syst.*, 2018, pp. 47–56.

[20] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock, "Automatically improving accuracy for floating point expressions," in *Proc. 36th ACM SIGPLAN Conf. Program. Lang. Design Implement.*. New York, NY, USA: Association for Computing Machinery, Jun. 2015, pp. 1–11.

[21] J. Xu, G. Song, B. Zhou, F. Li, J. Hao, and J. Zhao, "A holistic approach to automatic mixed-precision code generation and tuning for affine programs," in *Proc. 29th ACM SIGPLAN Annu. Symp. Princ. Pract. Parallel Program.*. New York, NY, USA: Association for Computing Machinery, Mar. 2024, pp. 55–67.

[22] W. Li and J. Mac Hyman, "Computer arithmetic for probability distribution variables," *Rel. Eng. Syst. Saf.*, vol. 85, nos. 1–3, pp. 191–209, Jul. 2004.

[23] T. Hickey, Q. Ju, and M. H. Van Emden, "Interval arithmetic: From principles to implementation," *J. ACM*, vol. 48, no. 5, pp. 1038–1068, Sep. 2001.

[24] F. Reghenzani, L. Santinelli, and W. Fornaciari, "Dealing with uncertainty in pWCET estimations," *ACM Trans. Embedded Comput. Syst.*, vol. 19, no. 5, pp. 1–23, Sep. 2020.

[25] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe, "Dependence graphs and compiler optimizations," in *Proc. 8th ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang. - POPL*. New York, NY, USA: Association for Computing Machinery, 1981, pp. 207–218.

[26] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke, "Paraprox: Pattern-based approximation for data parallel applications," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 35–50, Feb. 2014.

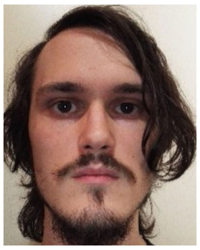
[27] G. A. Campbell, "Cognitive complexity—An overview and evaluation," in *Proc. IEEE/ACM Int. Conf. Tech. Debt*. New York, NY, USA: Association for Computing Machinery, May 2018, pp. 57–58.

[28] F. Blaschke, "Das verfahren der feldorientierung zur regelung der drehfeldmaschine," Ph.D. thesis, Res. NOT TU/e, Graduation NOT TU/e, Technische Universität Braunschweig, Dec. 1973.

[29] G. Magnani, D. Cattaneo, M. Chiari, and G. Agosta, "The impact of precision tuning on embedded systems performance: A case study on field-oriented control," *PARMA-DITAM*, vol. 88, pp. 31–313, Jun. 2021.

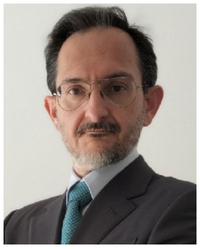
[30] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*. Upper Saddle River, NJ, USA: Prentice-Hall, 2008.

LEV DENISOV received the M.Sc. degree in big data management and analytics from TU Berlin, Germany, in 2019. He is currently pursuing the Ph.D. degree with Politecnico di Milano, Italy. His research interests include approximate computing, big data, and compilers.



GABRIELE MAGNANI received the M.Sc. degree in computer science and engineering from Politecnico di Milano, Italy, in 2021, where he is currently pursuing the Ph.D. degree. His M.Sc. thesis was on precision tuning of mathematically intensive programs. His research interests include the application of precision tuning in a concrete applicative context, computer security, and real-time systems.

DANIELE CATTANEO received the M.Sc. degree (cum laude) in computer science and engineering, in December 2018. He is currently pursuing the Ph.D. degree with Politecnico di Milano. His M.Sc. thesis work based on the architecture and implementation of a mixed precision compiler, which exploits the fixed point numerical representation. His research interests include embedded systems, compiler technology, and software–hardware co-design.



GIOVANNI AGOSTA received the M.S. (Laurea) and Ph.D. degrees from Politecnico di Milano, in 2000 and 2004, respectively. He is an Associate Professor with Politecnico di Milano. He has published over 100 papers in international journals and conferences. His research interest includes compiler technologies for the enforcement of extra functional properties. He is a Senior Member of the ACM and a member of HiPEAC NoE. He is an Associate Editor of *SoftwareX*.

STEFANO CHERUBIN received the Ph.D. degree in information engineering from Politecnico di Milano, Italy, in 2019, with a thesis on compiler-assisted dynamic precision tuning. He is an Associate Professor with NTNU, Norway. His research interests include compilers and approximate computing.

Open Access funding provided by 'Politecnico di Milano' within the CRUI CARE Agreement