Dynamic Precision Autotuning with TAFFO

STEFANO CHERUBIN, Politecnico di Milano, Italy
DANIELE CATTANEO, Politecnico di Milano, Italy
MICHELE CHIARI, Politecnico di Milano, Italy
GIOVANNI AGOSTA, Politecnico di Milano, Italy

Many classes of applications, both in the embedded and high performance domains, can trade off the accuracy of the computed results for computation performance. One way to achieve such a trade-off is precision tuning, that is to modify the data types used for the computation – by reducing the bit width, or by changing the representation from floating point to fixed point. We present a methodology for high-accuracy dynamic precision tuning based on the identification of input classes, i.e. classes of input data sets that benefit from similar optimizations. When a new input region is detected, the application kernels are re-compiled on the fly with the appropriate selection of parameters. In this way, we obtain a continuous optimization approach that enables the exploitation of the reduced precision computation, while progressively exploring the solution space, thus reducing the time required by compilation overheads. We provide tools to support the automation of the run-time part of the solution, leaving to the user only the task of identifying the input classes. Our approach provides a significant performance boost (up to 320%) on the typical approximate computing benchmarks, without meaningfully affecting the accuracy of the result, since the error remains always below 3%.

CCS Concepts: • **Software and its engineering** → **Software design tradeoffs**; **Compilers**; *Software performance.*

Additional Key Words and Phrases: Precision Tuning, Approximate Computing, Compiler; Fixed Point

ACM Reference Format:
Stefano Cherubin, Daniele Cattaneo, Michele Chiari, and Giovanni Agosta. 2019. Dynamic Precision Autotuning with TAFFO. *ACM Trans. Arch. Code Optim.* 1, 1, Article 1 (June 2019), 25 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

Approximate computing, a class of techniques that trade off computation quality with the effort expended [22] has recently gained significant traction as a way to achieve better performance (and reduced energy to solution) at the expense of a reduction in computation accuracy. Within approximate computing, one specific technique, *precision tuning*, deals with leveraging the trade off between the error introduced in the computation and the performance of the application by changing numeric representations.

Large classes of applications – such as media streaming and processing applications – can trade off unnecessary precision to improve performance metrics. The precision tuning approach uncovers optimization opportunities by exploiting lower precision data types with respect to the original

Authors' addresses: Stefano Cherubin, Politecnico di Milano, via Ponzio 34/5, Milano, MI, 20133, Italy, stefano.cherubin@polimi.it; Daniele Cattaneo, Politecnico di Milano, via Ponzio 34/5, Milano, MI, 20133, Italy, daniele.cattaneo@polimi.it; Michele Chiari, Politecnico di Milano, via Golgi 42, Milano, MI, 20133, Italy, michele.chiari@polimi.it; Giovanni Agosta, Politecnico di Milano, via Ponzio 34/5, Milano, MI, 20133, Italy, agosta@acm.org.

XXXX-XXXX/2019/6-ART1 $15.00
https://doi.org/0000001.0000001

implementation in different portions of the application. The user specifies the accuracy requirement, and precision tuning tools provide a precision mix that satisfies the requirement while minimizing a given cost function, which typically depends on the execution time [9, 11, 21, 26, 27].

Precision tuning is a complex process which requires several code analysis and transformation tools. It can be performed either statically or dynamically. Static precision tuning produces a single mixed precision version that runs for every possible input data. Dynamic precision tuning adapts the mixed precision version to varying runtime conditions, such as resource availability and input data. Despite many similarities between those two processes, they present different challenges and, thus, they require a slightly different toolchain structure.

Static mixed precision tuning is performed once for every application and it is considered part of the system design. In fact, precision tuning tools can be part of wider hardware/software co-design approaches. The base component of every mixed precision tool is the set of data type representations the tool can use – typically, the floating point representations defined in the IEEE-754 standard. Some tools also support fixed point representations, which are usually implemented either as signed or unsigned integer representations. Different tools use different approaches to choosing the representation. The most common approach usually involves a profiling phase of the application, to empirically measure the range of possible values that each variable can assume at runtime. Another approach involves static analysis of the application code. The first approach requires running the application, the second does not. However, the first approach cannot be considered a dynamic approach as the profiling happens before the application is deployed. Once a representation has been consolidated, it is necessary to appropriately change the data type used in the application. This code conversion can be performed on the source code of the application, on the binary machine code, or at an intermediate level within the compiler. Consequently, several variables in the code may require conversion to a different data type.

Dynamic mixed precision tuning is a recurring task, which is invoked multiple times while the application is running. In this case, the precision tuning is considered as a possible code transformation to perform *Continuous Program Optimization* [3, 19], which suggests to dynamically adapt the application to the system runtime conditions and to the state of the execution. Thus, the re-configuration overhead must be minimized. In the dynamic case, the code conversion tool should either support the dynamic generation of a mixed precision version, or rely on ahead-of-time compilation techniques. Thus, we introduce a *Dynamic Integration* component. Another difference with respect to static precision tuning lies in the *Tuning Policy*: this software component decides which mixed precision version should be used at runtime, and when to dynamically generate a new mixed precision version. In our experimental campaign, we generate new mixed precision versions 2 to 7 times in the program lifetime. The profiling phase of the analysis of the application is performed with a reduced input test set, or it is replaced by heuristics or static analysis to further reduce the overhead. Similarly to the aforementioned range analysis, the verification phase is also considered part of the overhead of generating a new mixed precision version. Therefore, faster verification techniques are recommended with respect to the static precision tuning.

In this paper we propose a partially automated approach to perform dynamic precision tuning. Our approach is based on state-of-the-art frameworks and on application-dependent software components. We describe such frameworks and we explain how we improved them to leverage our approach in Section 2. We also provide guidelines to effectively design and implement application-dependent software components. In Section 3 we detail the application use cases, our proposed implementation for the application-dependent components, and the evaluation metrics we use. In Section 4 we discuss the experimental campaign we conducted to assess our solution. We compare our work with similar tools and approaches in the state of the art in Section 5. Finally, Section 6 concludes the paper.

2 PROPOSED APPROACH

Software is usually written with the specific purpose to be general enough so it can properly process diverse input data sets. We assume to work with programs that process input data sets whose data layout is known a-priori. This is the case of most application domains. However, the values such data sets contain may vary significantly. Algorithmic decisions – such as which data type is the most appropriate for each variable – must be safe enough to properly deal with all the possible values in the input data. We propose to improve the performance of the software system by specializing the application kernel for the average case(s). Software specialization can be done a priori whenever the relevant average cases are known in advance. However, in a real world scenario, the characteristics of the information source – and of the input data it generates – are often unknown. In these cases, the characterization of the input and the requirements, to properly process it, have to be computed during the runtime of the application. As the number of possible code versions resulting from characterizations is unbounded, dynamic code specialization is required. This approach is a particular case of *Continuous Optimization* [19], which is a well-known practice that suggests to dynamically adapt the application to the system runtime conditions and to the state of the execution.

The runtime reconfiguration it performs modifies the precision of the data representations exploited by the application. This transformation aims at optimizing a given performance metric, which is usually the execution time. Such reconfiguration happens every time the runtime conditions vary significantly. As the runtime conditions are defined in terms of the input data, their specific definition is application-dependent. Additionally, the reconfiguration process also requires a certain amount of time to complete, which must be taken into account. Tuning the threshold on the runtime conditions that triggers the reconfiguration is a task that depends both on the reconfiguration time, and on the benefits that the mixed precision version brings. Consequently, we prefer to exploit a suboptimal mixed precision configuration whenever its setup is significantly faster with respect to a more optimal configuration.

The reconfiguration process consists in the dynamic re-compilation of the application kernel. In particular, our approach is based on five software components. The **code analysis** provides the accuracy requirement for each variable and intermediate value in the program. The **code manipulation** elaborates the application code and produces a mixed precision version of the application kernel. The **code validation** verifies that the mixed precision version actually satisfies the accuracy requirements, and that it represents an improvement with respect to the original version according to a given performance metric. The **tuning policy** decides which atomic input unit to be processed – input batch – should be executed using which code version, and it eventually triggers the generation of a new code version. Finally, the **dynamic code integration** component allows us to dynamically generate a new code version, and to dynamically execute it.

The need for a quick reconfiguration time changes the structure of the precision tuning toolchain from the one traditionally employed for static precision tuning. Examples of such static approaches are *Precimonious* [26] and *HiFPTuner* [9]. In particular, we suggest to limit the code analysis and code transformation time, in order to achieve a quicker time-to-solution. For this reason, their profiling phase should be replaced by a static analysis, whose complexity depends on the code size and not on the test input data. Additionally, the source-to-source compilation stage should be replaced by a compiler-level transformation, which allows us to limit the code transformation and compilation time. Finally, the validation can be performed by a static estimation of the error bounds instead of by a reference-based error verification.

The implementation we propose is based on the dynamic compilation library libVersioningCompiler [5] and on the compiler-level Tuning Assistant for Floating point to Fixed point Optimization

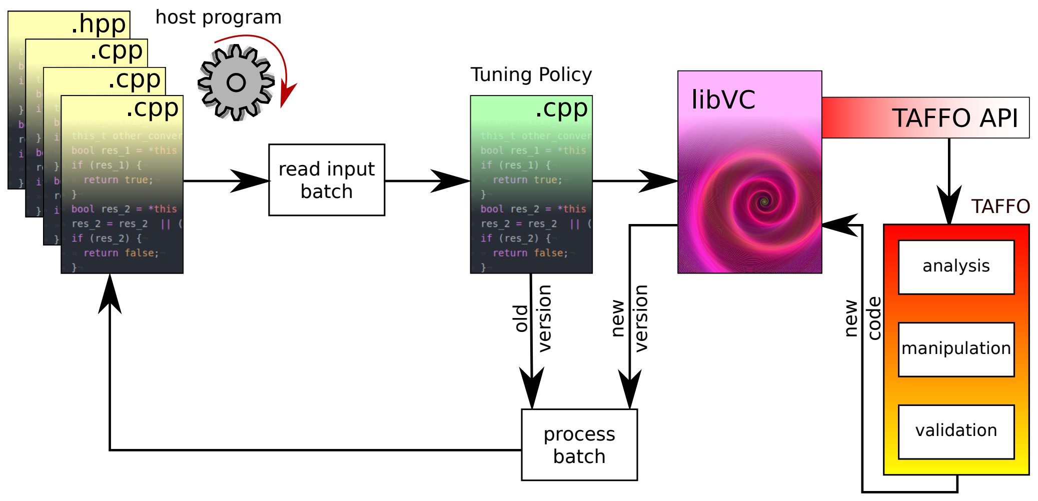

Fig. 1. Dynamic compilation toolchain overview.

TAFFO [8]. LIBVERSIONINGCOMPILER– abbreviated LIBVC– allows us to dynamically invoke compilation and compiler optimization stages on generic `C/C++` code. In our workflow, it represents the dynamic code integration component. LIBVC APIs enable also the dynamic invocation of precision tuning software. However, LIBVC itself is unable to perform any precision tuning task. Hence, we coupled LIBVC with a dedicated component.

TAFFO is a precision tuning assistant which has already been employed in the embedded systems [4] and in the Approximate Computing domain [8] to convert floating point code into fixed point. More specifically, the TAFFO toolchain supplies the code analysis, the code manipulation, and the code validation software components. We choose TAFFO as it meets the requirements of being based on static analyses, which are required for minimizing reconfiguration time.

Figure 1 summarizes our approach. It also highlights which component of our toolchain covers which role.

We adapt LIBVC and TAFFO to meet the needs of dynamic precision tuning. Also, we define which features of the application are desired to make dynamic precision tuning effective, and which guidelines to follow when formulating the tuning policy. We continue our discussion by describing in detail all of these aspects. In particular, Section 2.1 analyses the dynamic code compilation and integration task performed through LIBVC. Section 2.2 provides a deeper discussion on how TAFFO performs the code analysis, code manipulation, and code verification. Finally, the tuning policy component is characterized in Section 2.3.

2.1 LIBVERSIONINGCOMPILER

LIBVC is a software tool designed to apply continuous optimization. It uses dynamic compilation as a means to adapt the kernel code to the input batch. The typical usage of LIBVC follows a well-structured flow: declaration of available compilation tools, setup of the configuration for the code version, dynamic compilation, and on-demand code invocation through a function pointer, as shown in Figure 2. LIBVC exposes APIs to individually control each of those stages, for each individual kernel. We demand the reader to our previous work for in-depth details [5].

In the libVC model, a `Compiler` is defined as an instance of a given compiler interface that manages the process of compiling the compute kernel. libVC itself supports different `Compiler` implementations. They are either based on the *compiler-as-a-library* paradigm, or they rely on other software, which is usually already available on the host system – such as gcc or clang.

The fundamental units on which libVC is able to perform continuous optimization are syntactically-valid C or C++ procedures or functions. In the context of a typical application of libVC, such procedures or functions are implementations of a computational kernel. A `Version` associates the computational kernels – defined as a list of source files – with a given `Compiler` and the options to be given to said `Compiler`. `Version` are instances whose configuration is considered immutable throughout the object lifetime.

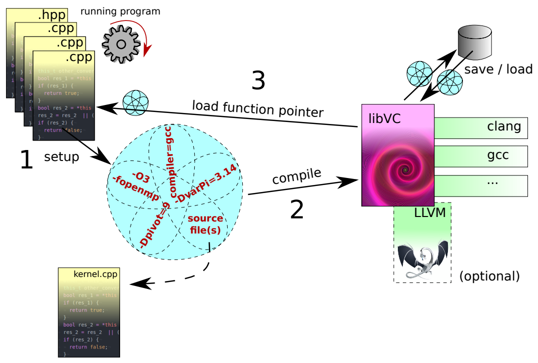

Fig. 2. libVC dynamic compilation flow.

Multiple `Version`s can be created from the same kernel(s) using, for instance, different compiler options. However, libVC itself does not provide any automatic selection of which `Version` is the most suitable to be executed under any circumstance. This decision is left to other software components such as application autotuners [16, 17], or to user-defined policies. In our approach, we design policies to select the most suitable `Version` with a minimal overhead.

At the end of the compilation stage, libVC generates a binary shared object, which is used to dynamically load the specialized code into the running application. libVC allows code versioning and reuse. Thus, `Version` instances can be re-used multiple times during the lifetime of the program. The compilation process may trigger more than one sub-task. In this case, we apply split-compilation techniques [10] to separately invoke each stage of the TAFFO precision tuning toolchain.

The target kernel(s) may include other files or refer to external symbols. libVC will act just as a compiler invocation and will try to resolve external symbols according to the given compiler and linker options. libVC defers the resolution of the compilation parameters to run-time. The only piece of information that is needed at design-time is the signature of each kernel, which has to be used for a proper function pointer cast.

The source code of libVC is available under the LGPLv3 license. Please refer to the official code repository (https://github.com/skeru/libVersioningCompiler) for a detailed and exhaustive documentation.

2.2 TAFFO

TAFFO is an open source (https://github.com/HEAPLab/TAFFO) compiler-based precision tuner. It is packaged as a set of plugins for the LLVM compiler toolchain. Although TAFFO was originally intended for static precision tuning, we extended and improved it to leverage the continuous optimization use case.

In the static precision tuning use-case, in order to use TAFFO, the programmer first adds *annotations* to the source code. The annotations appear in the source code as custom C/C++ attributes applied to variable declarations. An example of how annotations appear in the source code is shown in Listing 1.

Annotations identify which variables in the program must be taken into account for optimization, in order to allow the programmer to restrict the scope of the optimization. For example, to perform

```c
float a __attribute((annotate("scalar(range(-1,1))")));
float b __attribute((annotate("scalar()")));
```

Listing 1. Example of annotated C code where the programmer is asking to transform the variables a and b to a fixed point. Furthermore, the programmer is providing the value range only for the a variable. The value range for the b variable will be automatically inferred.

precision tuning of the entire kernel of an application, the programmer annotates all the variables used in the kernel. Optionally, the annotations may also specify the range of the values which each variable will store at runtime. This feature is intended for variables whose value ranges cannot be inferred statically. One example is a variable which is initialized by reading a number from a file.

From this annotated source code, TAFFO produces a single reduced precision version of the same code, employing fixed point representations.

The structure of TAFFO can be summarized in Figure 3. It is composed by five blocks: INIT, VRA, DTA, CONV, and FE.

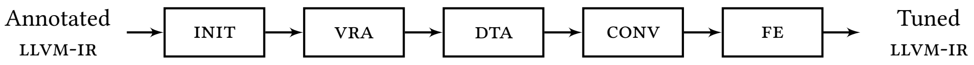

Fig. 3. TAFFO toolchain component structure

INIT is the first pass of the TAFFO toolchain. It performs various preprocessing steps on the software module to be optimized. In particular, it reads the variable annotations, it determines which portions of code will be affected by the code transformation steps, and it represents all such information as LLVM Metadata[1]. All subsequent passes of TAFFO use the metadata originally produced by INIT in order to share information that cannot be inferred by the LLVM-IR code itself.

VRA is the analysis component that statically performs the *Value Range Analysis*. It takes as input the ranges of possible input values provided by the user (as stored into metadata by INIT), and it provides a safe estimation of the range of intermediate and output values thorough the given application. This estimation is computed according to the rules of *interval arithmetic* [24].

DTA (*Data Type Allocation*) chooses an appropriate data type for each intermediate value of the computation, based on the output of VRA. This component mitigates the overhead caused by type casts via heuristics that minimize the number of such operations in the data flow.

CONV is the *Code Conversion* stage, in which the reduced precision code is actually generated by rewriting the original LLVM-IR code of the application. This component preserves the semantics of the original code as much as possible. In fact, whenever the code conversion using the previously decided data type is not possible due to unknown pieces of code (e.g. external function calls), it preserves the original data type for that portion of code.

FE is the *Feedback Estimator* component. It provides an estimation of the rounding error on the output of the converted code. It also exploits statistics derived from the analysis of the program to provide an estimate of the execution time improvements achieved. Both of these estimates are produced statically, without running the application.

As discussed, the *Feedback Estimator* is made of two sub-modules which perform the *Error Propagation* and the *Performance Estimation*, respectively.

[1]https://llvm.org/docs/LangRef.html#metadata

The *Error Propagation* component statically computes an approximation of the upper bound for the absolute numerical error, by propagating the initial errors on input variables and the rounding errors introduced by the fixed-point data types throughout the computation. Absolute errors are represented by means of affine forms [14], which allow the estimator to take into account the contribution of each error source separately. Subsequently, they are combined with the variable ranges computed by the VRA pass, in order to estimate the final upper bound on the error. Our approach is compliant with other static analysis methodologies in the literature [12, 13].

The *Performance Estimator* employs a supervised learning approach to predicting whether a specific application kernel gains a speedup on a given hardware architecture. It extracts pre-defined metrics from the LLVM-IR code of the application, both before and after the code rewriting process performed by the CONV pass. Afterwards, such metrics are used as the input of a statistical model which has previously been trained by the user. This statistical model predicts whether a speedup is achieved or not.

2.2.1 Dynamic compilation system integration. TAFFO is a framework originally intended for static precision tuning at compiler level. The extensible approach to dynamic compilation provided by LIBVC allows us to use any kind of compiler or compiler optimization within a dynamic setting. However, the dynamic precision tuning use case requires to enable parametric annotations on the source code that TAFFO requires as input. In fact, the dynamic ranges of values change depending on the workload, and these changes must be notified to TAFFO without an explicit human intervention that provides a different source file featuring the updated annotations. To this end, we implement software support for generating TAFFO annotations via the LIBVC framework. The values in such annotations are then inserted into the source code by means of C preprocessor definitions – `define` –, which we specify at (dynamic) compilation time.

2.3 Tuning policy

Within a continuous optimization environment, the software system iteratively processes several batches of input data. For each of such batches, it can decide whether to keep running the same kernel version or to change it. For the sake of simplicity, we describe our approach in the case of a sequential application. This approach, however, can also be ported to the parallel case by implementing the decision process on each processing element. In such scenario, the tuning policy is the system component that controls the trade off between code reuse and exploration of new versions.

Our precision tuning policies can decide whether to generate a new kernel version on-the-fly, or to reuse an already available version. In the former case, the precision tuning policy also defines which is the most suitable set of parameters required to generate that version. This property enforces the tuning policy to collect the precision requirements of the kernel before triggering the precision tuner. Indeed, there should be a function that computes the set of tuning options – such as the range of input variables required by TAFFO– for each input batch.

Additionally, we recall that TAFFO also provides an estimation of the quality of the reduced precision version. Since this estimation is performed via a static analysis during the code compilation stages, the precision tuning policy can double-check whether the freshly generated version meets the requirements.

In our proposed solution, we define a policy which provides an input partitioning based on a *workload equivalence function* for each target platform. The workload equivalence function is the component that properly partitions the input space into classes which can be considered equivalent from the continuous optimization point of view. Therefore, the same code version should equally apply to any input batch of the same class. More specifically, we designed a set of requirements to guide the design of the workload equivalence function. All the batches that belong to the same input class must:

(1) share the same hardware resource **requirements**;
(2) share the same input and output **format**;
(3) share the same **verification** metrics and methods;
(4) have a defined ratio between their problem **sizes**.

The first two constraints guarantee that two batches from the same input class can be processed using the same platform-optimized kernel version. The latter ones enforce that the execution statistics for those input batches can be fairly compared in terms of output quality and performance per input size. For example, input batches whose processing involves different hardware accelerators should be classified in separate partitions. Furthermore, applications that can perform multiple tasks on different batches should consider the task to be performed in the batch partitioning. Additional desirable similarities between elements of the same partition sets given by the equivalence function are:

(5) kernel **code coverage**;
(6) kernel **memory footprint**;
(7) **accuracy** requirements.

These properties minimize the distance between elements of the same partition measured in terms of quality of the output and of performance metrics. Indeed, a good workload equivalence function classifies in the same class input batches that are indistinguishable from the application profile point of view. For example, an equivalence function based on the magnitude of the input values is likely to satisfy such requirements. On the contrary, another function that partitions input batches according to whether an input values is odd or even, hardly meets such properties.

The workload equivalence function is required to partition the input into classes large enough to allow the optimization time to be paid back by the performance improvements.

3 APPLICATION-SPECIFIC POLICIES

To demonstrate how the user can define the tuning policies, we select as benchmarks six applications from the AxBench suite [30], which is intended for use in approximate computing research. AxBench consists of numerical programs that are representative of several application domains, as they contain substantial numerical kernels based on floating point computations. In this section we do not focus on the purely mechanical process of using the APIs of libVC to program the tuning policies. Such aspects are discussed in detail in [5] and [8].

We apply the guidelines described in Section 2.3 to characterize the behavior of the kernels of each application with respect to a variation of the input data. Thus, we partition their dataset into batches with common accuracy requirements and optimization opportunities. Such partitions are characterized by means of workload equivalence functions, which classify each possible input of a benchmark by assigning it to a batch. For each benchmark B with input i_B in the set I_B, workload equivalence functions are of the form $f_B^{we} : I_B \longrightarrow \mathbb{N}$. The output value $n = f_B^{we}(i_B)$ represents the equivalence class the input i_B is assigned to. Workload equivalence functions allow our software stack to dynamically optimize the floating point kernel by converting it to a mix of fixed point formats tailored to those input values. We thus pursue both accuracy and performance improvements over the floating point versions.

Our approach can optimize any performance metric to guide the workload equivalence function. In the rest of the paper, we consider the time-to-solution as a performance metric.

The choice of the workload equivalence function can be provided by the user, if they have a sufficient expertise on the numerical aspects of the application. In the general case, we provide a computation-agnostic way of defining them. Since the choice of the fixed point format for each instruction depends on the order of magnitude of its range, we define such functions based on the order of magnitude of the input variables. Let B be a benchmark with m input variables, i.e. $I_B = V_1 \times V_2 \times \cdots \times V_m$, and let $\rho_m : \mathbb{N}^m \longrightarrow \mathbb{N}$ be a bijection. Then we define the workload equivalence function of B as

$$f_B^{we}(v_1, v_2, \ldots, v_m) = \rho_m(\lfloor \log_{10}(v_1) \rfloor, \lfloor \log_{10}(v_2) \rfloor, \ldots, \lfloor \log_{10}(v_m) \rfloor),$$

with $v_p \in V_p$ for $1 \le p \le m$. The granularity of batches can be controlled by changing the base of the logarithms. Base 2 allows us to define batches whose size of the integer parts differ by roughly one bit.

This definition, however, may induce an excessive number of input batches, especially for benchmarks with a high number of input variables. This can be avoided by selecting a subset of input variables to be used to detect the input class. A possible criterion to make this choice is to only select those variables that are involved in –possibly non-linear– operations that cause a significant variation in the value ranges. This can be easily done by running a breadth-first search on the data-flow graph of the program, after running the VRA pass of TAFFO. The search looks for operations whose results have a range at least an order of magnitude (i.e. ten times) larger or smaller than those of their operands. Whenever such an operation is found, all input variables that appear earlier in the data-flow graph are included in the workload equivalence function.

We rely on the error metrics defined by the AxBench benchmark suite. Note that the input dataset provided by the benchmark suite may be under-representative of the possible input variability. Thus, it entails a limited number of equivalence classes. We slightly modify some of the input datasets defined in the AxBench suite to enable a meaningful input partitioning. In general, the number of possible input partitions can be either unlimited, or too high to permit an ahead-of-time optimization approach. The efficiency of our approach does not depend on the number of classes identified by the workload equivalence function. Indeed, the number of classes concurrently allowed to be processed is limited only by the number of `Version` objects that can concurrently fit in memory. The spatial complexity of the `Version` objects is linear with respect to the size of the code produced by the dynamic optimization process. Although LIBVC allows the extension of this limit by offloading some unused `Version` objects to mass memory, during the experimental campaign discussed in this paper we assume this limit is never reached. We also assume that disk memory page swapping does not occur. No assumption is required on the smoothness – nor on any other property of the distribution – of the input classes in the application dataset.

The applications we discuss in this paper are *Black-Scholes*, *FFT*, *Inversek2j*, *Jmeint*, *K-means*, and *Sobel*. We do not consider *JPEG encoder*, since most of its code already uses integer data types. We briefly describe such benchmarks, and we detail their workload equivalence functions below.

3.1 Black-Scholes

This benchmark computes the equation for the value of call options according to the Black-Scholes model of a financial market. We rely on the un-modified input dataset, which consists of 48, 000 options. Each option is characterized by the input parameters of the Black-Scholes equations:

- the *spot price* S of the underlying asset,
- the *strike price* K,
- the *risk-free rate* r,
- the *volatility* σ of the option,
- and its *time to maturity* t.

The benchmark outputs the estimated value of the option, for each input tuple. The accuracy of the computation for the fixed-point versions is evaluated by computing the average relative error with respect to the floating point version.

Arithmetic and Architectural Characterization. The implementation of this benchmark processes each option singularly, and each computation takes an approximately constant number of operations. In particular, the most frequent operations performed in the kernel are multiplications, which amount to 50 per iteration. Additionally, these multiplications appear in a long data dependency chain consisting of at least 20 operations. The longest dependency chain consisting of multiplications alone has a depth of 5 operations. Thus, we conclude that this benchmark primarily stresses the Floating Point Multiplication unit.

This benchmark also uses the `exp`, `sqrt` and `log` mathematical functions. However they only appear once each – with the exception of `exp` which appears 3 times. Consequently, they are not important in the architectural characterization of the benchmark.

Regarding the arithmetic characterization, we observe that each option is represented by 6 floating point numbers and one integer. This data amounts to 28 bytes, when using the single-precision floating point representation. The instruction mix – measured from the LLVM-IR– evidences that each iteration of the kernel performs a total of 90 floating point operations. Thus, the kernel processing rate is approximately $3.2 \frac{\text{FLOPS}}{\text{byte}}$.

Tuning Policy. Since this benchmark is essentially made of straight-line code with no variations on the input/output format, any workload equivalence function based on the range of the input values satisfies requirements (1)–(3) and (5) of Section 2.3. The application processes the input one option per iteration, and every option has a fixed memory footprint which does not depend on the value of the option itself. Therefore, the number of options processed define the problem size. This characteristic allow us to meet requirements (4) and (6) for every workload equivalence function. As for point (7), i.e. accuracy requirements, we notice that they are highly influenced by the order of magnitude of some of the input parameters. In order to reduce the number of input classes, we limit the number of parameters considered by the workload equivalence function as described in the previous paragraph, which yields the following function:

$$f^{we}_{\text{Black-Scholes}}(S, K, r, \sigma, t) = \rho_2(\lfloor \log_{10}(\sigma) \rfloor, \lfloor \log_{10}(t) \rfloor)$$

Since σ and t respectively range in $[0.05, 0.65]$ and $[0.05, 1.00]$ in the input dataset provided by AxBench, the dataset is divided into six batches, two of which are empty (those value combinations never occur in the dataset). However, in real-world applications σ and t could range on a wider domain. Indeed, the orders of magnitude of σ and t highly influence the ranges of local variables of intermediate computations, and restraining them allows TAFFO to emit a more specialized version of the benchmark, yielding more accurate results.

3.2 FFT

The FFT benchmark computes the in-place Radix-2 Cooley-Tukey Fast Fourier Transform algorithm, which is widely used in signal processing to convert a signal from the time domain to the frequency domain. The input is an integer value K, which determines the size of the FFT. As the input signal of the FFT proper, the benchmark internally generates a discrete rectangular wave of period K and duty cycle 1%. This signal is fully real, and has no imaginary part. The accuracy of the output is evaluated by computing the average relative error with respect to the floating point version.

Arithmetic and Architectural Characterization. The inner loop of the kernel consists of the computation of a cosine, the computation of a sine, 8 independent multiplications, and 4 pairs of additions.

Each pair of additions depends on two multiplications. Each multiplication depends on either the computation of the cosine, or the sine (but not both). Thus, the longest dependency chain found in the kernel consists of 4 operations.

Since all the other operations in the kernel depend on the computation of trigonometric functions, we conclude that they are the main bottleneck that contributes to the execution time of each iteration. Additionally, this benchmark heavily exploits the multiplication and addition units, and is particularly suited for exercising the superscalar features of the architecture, because of the limited length of the longest data dependency chain.

From the same observations we can deduce the arithmetic intensity of the kernel. In first instance, we notice that the inner loop of the kernel is executed $2K - 1$ times. Thus, the total amount of floating point operations performed for processing a signal of K samples is $18\,(2K - 1)$. The total size of the data being processed is $8K$ bytes, as each sample has both a real part and an imaginary part, represented as single-precision floating point numbers. We can conclude that this benchmark has an arithmetic intensity of $\frac{18(2K-1)}{8K} \approx 4.5\,\frac{\text{FLOPS}}{\text{byte}}$.

Tuning Policy. The computation is mostly straight-line, so the code coverage, the input/output format and, thus, the verification metrics do not vary depending on the input, satisfying requirements (1)–(3) and (5). However, K influences both the absolute value and the amount of different frequency intervals, thus influencing the problem size, the memory footprint and the accuracy requirements. The latter are bound to the value of K: a high value of K yields wider dynamic ranges for intermediate and output values. Therefore, a fixed point optimization that avoids overflow errors for every possible value of K should use representations with too low of a number of fractional bits. This constraint prevents us from achieving a reasonable relative error when they are executed with lower values of the input K. Whenever the value K raises above a certain threshold, almost all bits of the fixed point variables are allocated to the integer part. As a result, catastrophic bit cancellation is bound to happen during the processing. In this case, our tuning policy decides to run the original floating point kernel, which is best suited for highly dynamic variable ranges.

Since the radix of this FFT implementation is 2, we use a base-2 logarithm for the input batches.

$$f^{we}_{\text{FFT}}(K) = \rho_1(\lfloor \log_2(K) \rfloor)$$

The estimated and actual errors suggest that the cancellation issues arise with $f^{we}_{\text{FFT}}(K) > 20$, so for input values above this threshold we fall back on original floating-point implementation. In Section 4.3 we evaluate our solution by executing a dynamically optimized version of this application on three of the classes below 20.

3.3 Inversek2j

Inversek2j implements the kinematics of a 2-joint robotic arm. It reads the angular coordinates of the two arms, in radians, converts them to Cartesian and then back to angular, using the length of the two arms as a parameter. The average absolute and relative errors with respect to the original benchmark are used to estimate accuracy.

Arithmetic and Architectural Characterization. This benchmark is composed by two distinct kernels, with distinct arithmetic features. One performs the conversion from polar to Cartesian coordinates (named `forwardk2j`), the other performs the inverse conversion (`inversek2j`).

`forwardk2j` is the simplest of the two. It features two independent expressions, one of which contains two `sin` operations, the other two `cos` operations. The longest data dependency chain is only four operations deep, and there are two such dependency chains that can be executed in parallel. They all involve two additions, one multiplication, and one trigonometric function which depends on one of the two additions. Regarding the arithmetic intensity, this kernel executes 12 floating point operations for each coordinate. The coordinates are represented in 8 bytes when exploiting the floating point mixed-precision representation, thus the arithmetic intensity is $1.5\frac{\text{FLOPS}}{\text{byte}}$

In contrast to `forwardk2j`, the instruction mix of `inversek2j` features a wider variety of operations: 10 multiplications, 4 calls to trigonometric functions, 3 subtractions, 2 additions and 2 divisions. The longest data dependency chain consists of 3 subtractions, 3 calls to trigonometric functions, 2 multiplications and 2 additions, which amounts to more than 50 % of the operations performed by the kernel. The arithmetic intensity of this kernel is approximately $2.6\frac{\text{FLOPS}}{\text{byte}}$

It is clear that `inversek2j` is computationally more complex than `forwardk2j`, and both of the kernels provide a good mix of operations which exercise several functional units. However, the calling overhead intrinsic to the trigonometric operations is very important. Thus, this benchmark stresses the performance of trigonometric operations the most, while also exercising – albeit at a lesser degree – the multiplication and addition units of the FPU.

Tuning Policy. This application is made of mostly straight-line code, so all requirements of the workload evaluation function are automatically satisfied, except accuracy. By employing the variable selection heuristic described earlier, we determined that the computation is actually more sensitive to arm length than to angular coordinates, the latter being constrained between 0 and $\frac{\pi}{2}$. Therefore, the workload equivalence function for this benchmark is

$$f^{we}_{\text{Inversek2j}}(l_1, l_2) = \rho_2(\lfloor \log_{10}(l_1) \rfloor, \lfloor \log_{10}(l_2) \rfloor),$$

where l_1 and l_2 are the two arm lengths. We evaluate the optimized versions of Inversek2j on different combinations of the arm length values, ranging from 0.5 to 50.

3.4 Jmeint

The Jmeint benchmark computes whether a pair of triangles intersects or not, using an algorithm originally intended for use in computer game engines [23]. The dataset for this benchmark consists of a list of pairs of triangles to check for intersection. Each pair is evaluated one at time, and a triangle from one pair is never compared with a triangle from another pair. Thus, the problem size is in direct proportion to the number of triangle pairs contained in the dataset. The output of the benchmark is a binary value for each triangle pair. If the output equals *true*, then the two triangles of the pair intersect, otherwise they do not. The quality of the approximation is measured by the number of incorrect classifications of the triangle pairs.

Arithmetic and Architectural Characterization. The characterization of this benchmark is complicated by the fact that the algorithm contains several code paths, that are only taken if the triangles being tested satisfy specific conditions. In particular, the conditions are the following:

(1) All points of triangle 1 are on the same side (trivial rejection).
(2) All points of triangle 2 are on the same side (trivial rejection).
(3) The triangles were found to be coplanar while computing the intervals for triangle 1.
(4) The triangles were found to be coplanar while computing the intervals for triangle 2.
(5) Ordinary non-intersecting triangles.
(6) Ordinary intersecting triangles.

It is noteworthy that exit conditions 3 and 4 involve a code path that is not executed when the other exit conditions are taken, named `coplanar_tri_tri`.

To evaluate the arithmetic intensity of this benchmark, we compute the amount of floating point operations performed by the kernel in each one of conditions 1-6. First, we modify the source code of the benchmark as to remove the statements that are not executed in each case. Then, the modified source code is compiled into LLVM-IR files. The number of floating point operations performed per iteration is calculated by counting the number of floating point instructions in the LLVM-IR. Conversely, the data processed by each kernel iteration amounts to 72 bytes, which is the number of bytes required to represent 18 single-precision floating point numbers. From these considerations we compute the arithmetic intensity for each case, which is shown in Table 1.

Table 1. Instruction mix and arithmetic intensity of each possible execution path in the Jmeint benchmark.

	Case				
	1	2	3	4	5, 6
# `fabs`	6	24	54	54	51
# `fadd`	11	66	330	330	248
# `fsub`	10	60	1563	1563	201
# `fmul`	26	132	1461	1472	252
# `fdiv`	0	0	0	0	0
FLOPS	53	282	3408	3419	702
FLOPS/byte	0.7	3.9	47.3	47.5	9.8

We notice how cases 5 and 6 have the same arithmetic intensity. In fact they execute the same code, except for a single branch, which does not execute any additional floating point operation. The disparity between cases 3 and 4, and the other cases, is due to the additional operations performed by the `coplanar_tri_tri` code path. If we consider all cases to be equally probable, the average arithmetic intensity is approximately $21.8\,\frac{\text{FLOPS}}{\text{byte}}$.

From the architectural point of view, from the instruction mix we observe a fairly equal split between the addition and multiplication workloads. While this benchmark does indeed exploit a math library function (in particular, `fabs`), we observe that LLVM implements such function through an intrinsic which does not translate into a function call in machine code. Thus, there are no additional overheads due to function calls. Additionally, we observe that the code makes heavy use of vector operations implemented through arrays. This gives the compiler several opportunities to vectorize the code. In summary, the performance of this benchmark mainly depends on the vector capabilities of the architecture, and on the latency and throughput of the multiplication unit.

Tuning Policy. As the output of the algorithm is a binary value for each triangle pair, the absolute error for a single triangle pair is a binary number as well: 1 in case of miscalculation, and 0 otherwise. The error metric produced by AxBench is the sample mean of the error across all the triangle pairs in the dataset, which approximates the probability of incorrect classification. However, the feedback estimator component is not able to perform probabilistic reasoning on binary values. Hence, we assess its performance by using the absolute error of the last intermediate value computed by the benchmarked algorithm before it establishes a classification for the triangle pair.

Due to the structure of the input dataset, it is not possible to meaningfully differentiate the input classes through requirements (1) through (4). More in detail, requirements (1) and (4) are always satisfied, as every single triangle intersection test is independent and of a predetermined constant size. Since only a single input format and a single output format are supported, requirement (2) is always trivially true as well. Requirement (3) is always true because the problem is homogeneous across all possible datasets. However, we observe that different triangle pairs will generate a different code coverage, which allows us to use equivalence condition (5) as a discriminating factor. Since the algorithm operates *in-place*, and none of the possible code paths in the algorithm perform memory allocations, condition (6) is always satisfied.

Finally, for what concerns condition (7), the target application of this benchmark – computer games – is notoriously tolerant of imprecise computations when they enable a performance tradeoff. Thus, it is hard to define a baseline of acceptable classification error rate. Additionally, the only parameter that may influence the precision of the results is the range of the coordinates of the vertices of the triangles.

We conclude that the workload equivalence function must represent both the effective code coverage determined by the inputs, and their range. However, we cannot reliably determine which exit condition will cause the termination of the algorithm unless the algorithm itself is executed. Thus we represent the code coverage by choosing a function dependent on the distance between the centroids of the triangles. Such a function cannot be formulated in an effective way through generic workload equivalence function we have previously introduced, because in the dataset provided by AxBench the coordinates of the vertices are uniform in order of magnitude. However, even though the expression of the function is not in the generic form, it can be derived without any in-depth knowledge of the algorithm, as it relies only on basic facts of euclidean geometry. The vertices of the triangles are represented by the variables A, B, and C for the first triangle, and D, E and F for the second.

$$\vec{c}(P,Q,R) = \frac{P+Q+R}{3}$$

$$d(P,Q) = \sqrt{(P_x - Q_x)^2 + (P_y - Q_y)^2 + (P_z - Q_z)^2}$$

$$r(P,Q,R) = \max_{S \in \{P,Q,R\}} \left(d\left(\vec{c}(P,Q,R), S\right)\right)$$

$$f^{we}_{\text{Jmeint}}(A,B,C,D,E,F) = \left\lceil 2 \max_{P_i \in \{A,B,C,D,E,F\}_{x,y,z}} (\log_{100}|P_i|) \right\rceil + \left\lfloor \max\left(\frac{2 \times d(\vec{c}(A,B,C), \vec{c}(D,E,F))}{r(A,B,C) + r(D,E,F)}, 1 \right) \right\rfloor$$

As all the datasets included in AxBench use a world coordinate system with coordinates restricted to values between 0 and 1, to represent a larger variety of use cases, we added data which uses coordinates between -100 and 100.

3.5 K-means

This benchmark uses a popular machine learning algorithm to classify pixels from an image into a user-specified number of clusters n. We rely on the input dataset provided by AxBench, consisting of a set of RGB pictures. Independently from the data, we parameterize the number of clusters n, by replacing the default constant value – 6 – by multiple alternatives, namely 6, 50, and 100.

Arithmetic and Architectural Characterization. We observe that the workload represented by this benchmark varies depending on the number of clusters. In fact, the task of classifying each pixel is performed by a linear iteration through a list. This operation involves 8 floating point operations, and the longest data dependency chain is 4 operations long. Among such operations, one is a call to the square root procedure. Thus, the throughput of the operations in the classification task proper is limited, as only few operations can take advantage of pipelining.

Additionally to the classification step, the benchmark also computes independent 3 floating point additions. In conclusion, as a single pixel is represented by 3 single-precision floating point numbers – 12 bytes –, the arithmetic intensity amounts to $\frac{8n+3}{12} \approx (0.67n + 0.25)\ \frac{\text{FLOPS}}{\text{byte}}$.

Tuning Policy. For the workload equivalence function, we only consider n, since taking into account pixel color values would cause an excessive overhead of the input class selection phase, especially with large pictures. The resulting input class subdivision is still significant. Indeed, the input is rather homogeneous, because all pictures in the dataset are equally sized, and the values of each of the three color components range from 0 to 255. This characteristic ensures sufficient similarity for all the requirements for the equivalence function, except for the last one. The accuracy requirements, in fact, depend on the number of clusters n, which influences the ranges of variables used to compute the cluster centroid positions. We thus obtain the following function:

$$f^{we}_{\text{K-means}}(n) = \rho_1(\lfloor \log_{10}(n) \rfloor)$$

The AxBench error evaluation metric is the image difference computed as RMSE with respect to a reference image. The final error mostly depends on misclassification of multiple pixels due to the numerical error on the computation of their distance from the cluster centroids. Since the feedback estimation pass only computes absolute errors, it cannot estimate the probability of cluster misclassification. Therefore, we assess the FE component by using an additional error metric, which can be computed by TAFFO. More specifically, we estimate and measure the average relative error on the computed Euclidean distances of each pixel from the centroids.

For each image of the input dataset, we evaluate our solution on this application by supplying it with the three above values of the number of clusters n.

3.6 Sobel

Sobel is an implementation of the well-known Sobel filter, used in computer vision applications, often for edge detection. This benchmark takes a grayscale image as its input – or an RGB image which is converted to grayscale – and outputs it after applying the filter.

Arithmetic and Architectural Characterization. For each pixel, the Sobel algorithm performs two convolution operations with different 2D kernels. Then, the new value of the pixel is computed as the euclidean distance of the results of the two convolutions, clamped to 255. The convolution operation consists of 18 floating point operations – 9 multiplications, and 9 additions. The additions all depend on exactly one multiplication, so the chance for pipeline utilization is slim. In total, the whole kernel totals 41 floating point operations. As each pixel is represented by one single-precision floating point number, the arithmetic intensity is approximately $10 \frac{\text{FLOPS}}{\text{byte}}$.

Tuning Policy. As in K-means, using individual pixel color values is unfeasible because of the computational overhead, which could be comparable to the execution of the benchmark itself. In this case, we chose to discriminate on the bit depth of the input pictures. Instead of using the dataset supplied with AxBench, which contains 8 bit pictures only, we use the pictures from [25], which are available in both 8 bit and 16 bit color depth. Thus, denoting the color depth as d, we use the following workload equivalence function:

$$f^{we}_{\text{K-means}}(d) = \rho_1(\lfloor \log_{10}(d) \rfloor).$$

The default AxBench error metric is the image difference computed as RMSE. Since the FE component of TAFFO cannot estimate this metric directly, we also evaluate the raw value returned by the kernel function, i.e. the normalized color value of each pixel.

4 EXPERIMENTAL EVALUATION

We evaluate our solution on two different machines, whose architecture is described below.

AMD a server NUMA node featuring four Six-Core AMD Opteron 8435 CPUs (@2.6 GHz, AMD K10 microarchitecture), with 128 GB of DDR2 memory (@800 MHz);

Intel a server NUMA node featuring two Intel Xeon E5-2630 V3 CPUs (@2.4 GHz) with 128 GB of DDR4 memory (@1866 MHz) on a dual channel memory configuration.

The AMD machine is running the Ubuntu 16.04 operating system. The Intel machine is running the Ubuntu 18.04 operating system. Both of the machines are kept unloaded for the duration of the experiments. The implementation of our solution is based on version 8.0.1 of the LLVM compiler infrastructure and its frontend CLANG.

Our experimental campaign replicates all the experiments on both machines.

In particular, we prepare the following versions for each benchmark application:

v_s the floating point application, as originally implemented in AxBench;

v_s^T modified application created by adding the annotations required by TAFFO, thus enabling static precision tuning on the original application;

v_d^T dynamically autotuned application, which has been created by applying dynamic compilation and tuning policies over the static precision tuned version.

For each such version, we follow the same compilation procedure. First, we insert annotation attributes in the source code of the application, as required by TAFFO. The content of such annotations changes depending on the version. For version v_s and v_s^T, the annotations specify value ranges appropriate for all input batches. Instead, for version v_d^T, the annotations are replaced with symbols that will be defined at a later stage. From the modified source code files, we then generate the corresponding LLVM-IR files. For versions v_s^T and v_d^T, we optimize the LLVM-IR using TAFFO. Version v_s^T is optimized statically, by running TAFFO once. Specifically in the case of version v_d^T, we execute TAFFO once for each input batch that we use, using annotations specialized for the input class of the batch. Lastly, we compile the optimized LLVM-IR files using the default set of optimizations of CLANG by using the -O3 command line option.

Our experimental campaign is designed to highlight three aspects: the effectiveness of the integration of LIBVC and TAFFO within the application as a whole, the individual performance of the *error propagation* component of TAFFO, and the contribution of the dynamic compilation component in improving the existing optimizations. The main performance components we consider are the execution time, and the error. Considering the application as a whole, these components are properties specific to the time of its execution, which may include compilation activities. Instead, for the error propagation component of TAFFO, we consider the error estimates it provides. For what concerns time values – both for the execution time and for the compilation time – they are always reported as a median over 100 samples.

4.1 Effectiveness Assessment

We start by evaluating the effectiveness of our solution as a whole. For each application kernel, and for each input class – according to the classification described in Section 3 –, we collect the following measurements:

T_{gen} time required to dynamically re-compile the application kernel from the v_d^T variant using the appropriate TAFFO annotations, and to validate it, i.e. running the two components of the TAFFO FE step;

T_d^T execution time of v_d^T, measured on a single input batch;

T_s execution time of v_s, measured on a single input batch;

From these measurements we compute the *speedup* of the dynamically compiled kernel with respect to the statically compiled kernel as:

$$\text{Speedup} = \frac{T_s}{T_d^T}$$

By examining Figure 4, we can observe that our approach – v_d^T – is able to improve the performance of the original version v_s with speedups up to roughly 320%. We observe slowdowns up to approximately 20% derived by the use of the dynamically tuned versions generated by TAFFO. In particular, the speedup we achieve is higher for the *K-means* and *Sobel* applications, whilst the *FFT*,

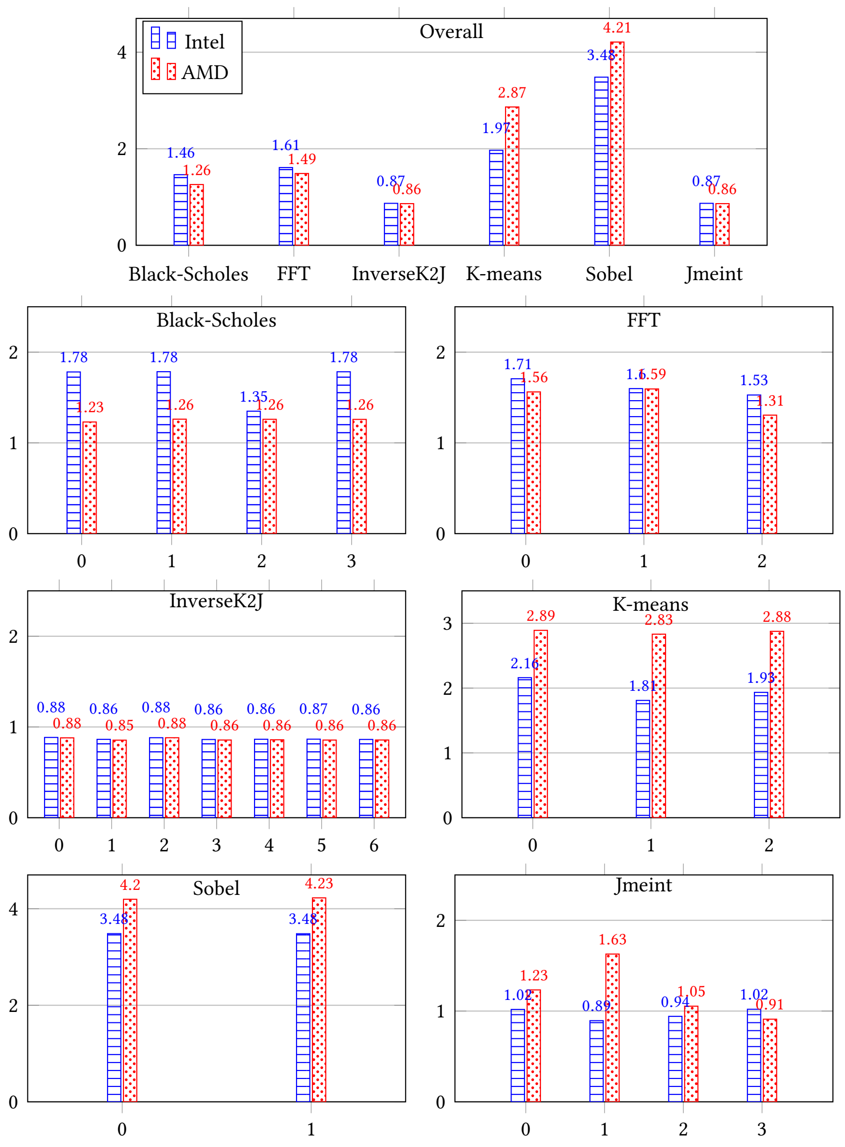

Fig. 4. Speedup of the v_d^T version over the baseline v_s version, overall and for each input batch.

Jmeint and *Black-Scholes* applications have more limited benefits. *InverseK2J* is the only application that does not benefit in any way from the transformation performed by TAFFO.

We believe that the large difference between *Sobel* and the other applications can be ascribed to the fact that *Sobel* is less reliant on multiplication, division, and transcendental functions. In fact, the fixed point implementations of these operations require additional type casts (bit shift operations) with respect to other operators. Additionally, we conducted an analysis of the assembly code generated by the compiler, in order to verify if different optimizations are applied to the fixed point version with respect to the floating point version. In the case of *K-means*, *Sobel*, *FFT*, *Black-Scholes* and *InverseK2J*, the fixed point versions do not appear to be vectorized differently than the floating point versions. More precisely, no vectorization has been observed, either in the floating point versions nor in the fixed point versions. Instead, for the *Jmeint* benchmark, we observe that the floating point version presents vectorization in specific code paths. Specifically, vectorization is observed in pieces of code corresponding to the macros `SUB`, `CROSS` and `DOT`. The *Jmeint* does not present significant speedups, which is a behavior consistent with the fact that the fixed point versions do not take advantage of vectorization. Thus, we conclude that the the speedup we obtain is caused by improved exploitation of pipelining in the CPU for the TAFFO-generated versions. This hypothesis has been partially verified by using the analysis tool `llvm-mca`[2]. The `llvm-mca` tool exploits the same performance models used in the back-end provided by the LLVM compiler framework, in order to simulate the pipelining behavior of a given piece of machine code. This analysis has been restricted to the *Intel* machine, as `llvm-mca` does not support the microarchitecture employed by the *AMD* machine. Our analysis of the computational kernel of the benchmarks performed by `llvm-mca` highlights that the benchmarks where we observe a speedup are not heavily pipelined, which allows the fixed point version to exploit the lower latency of integer operations as opposed to floating point operations.

Additionally, we compute the *break-even point* between our solution and the unmodified static solution. We define the *break-even point* as the ratio between the overhead required to generate a dynamically optimized version and the performance gain derived from such version.

$$\text{Break-Even} = \left\lceil \frac{T_{gen}}{T_s - T_d^T} \right\rceil$$

The value of the *break-even point* represents the size of the smallest input class that benefits from our approach.

For each machine, we report in Table 2 the analysis of the break-even point. There are applications where the gain in execution time is of a small magnitude, such as *Black-Scholes*. In these cases, the break-even point is very high. In particular, the class 1 of *Black-Scholes* requires a support of at least $2,605$ batches to start gaining advantage from our proposed solution. However, it is worth noticing that this value represents only less than 6% of the AxBENCH dataset. In a real world scenario, this value of break-even point can be considered acceptable. This is a consequence of the typical workload scale of such applications, which is typically larger by several orders of magnitude with respect to the dataset provided by AxBENCH.

We do not show the break-even point if the application does not present a gain in execution time. In case the computational kernel of the application runs only few times, the dynamic compilation model could be usefully replaced with more lightweight approaches, such as [15].

The accuracy of numerical computations is another crucial parameter that must be taken into account. We employ the error metrics defined by the AxBENCH benchmark: Average Relative Error (ARE) for *Black-Scholes* and for *FFT*, Root-Mean-Square Error (RMSE) between RGB pictures for *K-means* and *Sobel*, and Triangle Pair Classification Miss Rate for *Jmeint*. Additionally, we evaluate additional error metrics for specific benchmarks. For *K-means*, we evaluate the error on the computation of Euclidean distances between centroids – which can also be quantified with the ARE. For *Jmeint* we evaluate the error on several intermediate values computed by the benchmark, quantified as Absolute Errors. In Table 3 ("dynamic" columns), we show for each batch the numerical errors computed on the v_d^T version of the benchmarks with respect to the original version v_s, for all benchmarks except Jmeint. We show the same numerical errors for Jmeint in Table 4, in the *Error (dynamic)* column.

[2]https://llvm.org/docs/CommandGuide/llvm-mca.html

Table 2. Performance break-even analysis.

Benchmark		Batch Class	$T_{gen}[s]$	$T^{T}_{d}[ms]$	$T_{s}[ms]$	Break-Even [#batches]
Intel	Black-Scholes	0	0.89	0.838	1.493	1353
		1	0.89	0.835	1.490	1353
		2	0.89	17.100	23.074	149
		3	0.89	4.312	7.691	263
	FFT	0	1.49	1.319	2.251	1599
		1	1.48	6.248	9.985	396
		2	1.49	39.261	59.980	72
	InverseK2J	0	0.86	207.192	183.307	—
		1	0.88	213.518	184.407	—
		2	0.88	208.930	184.201	—
		3	0.88	213.423	183.951	—
		4	0.87	212.608	183.821	—
		5	0.87	212.437	183.905	—
		6	0.87	213.320	184.004	—
	K-Means	0	1.74	20.089	43.401	75
		1	1.73	126.138	228.282	17
		2	1.74	229.483	443.804	9
	Sobel	0	2.29	220.891	769.563	5
		1	2.29	222.110	773.396	5
	Jmeint	0	3.03	415.638	423.119	405
		1	3.03	510.399	456.315	—
		2	3.07	403.776	380.226	—
		3	3.07	337.569	344.294	457
AMD	Black-Scholes	0	1.80	3.099	3.813	2521
		1	1.80	2.643	3.334	2605
		2	1.81	53.729	67.697	130
		3	1.80	13.728	17.281	507
	FFT	0	2.09	4.413	6.892	846
		1	2.10	10.017	15.971	353
		2	2.11	72.940	95.252	95
	InverseK2J	0	1.77	499.433	439.518	—
		1	1.79	513.516	438.875	—
		2	1.80	498.506	439.394	—
		3	1.78	512.881	439.084	—
		4	1.77	511.111	438.962	—
		5	1.77	513.078	439.052	—
		6	1.77	512.825	439.138	—
	K-Means	0	2.52	35.121	101.526	38
		1	2.52	222.941	631.335	7
		2	2.52	424.765	1221.485	4
	Sobel	0	4.25	489.677	2054.688	3
		1	4.25	483.870	2044.418	3
	Jmeint	0	4.89	245.044	243.199	—
		1	4.89	333.827	220.580	—
		2	4.96	229.983	216.697	—
		3	4.96	134.484	146.766	404

All the resulting relative errors are below 3 %. In the case of *Jmeint*, we register miss rates below 0.05 %. In particular, TAFFO is able to exploit the peculiarities of each input batch to provide a more accurate optimized kernel, by feeding the VRA with tighter initial ranges.

For example, notice how the lower value of the input parameter in *FFT* allows the optimized kernel to output a negligible absolute error. The same happens — to a lower extent — for *InverseK2J*. As for *K-means*, the RMSE is mostly correlated to the probability of cluster misclassification by the v_d^T kernel with respect to the v_s one. Its values — being lower than 0.5% — suggest that the misclassification rate is very low for all pictures in the input dataset, even when a large number of clusters is used. Also in *Black-Scholes*, *Sobel* and *Jmeint* the absolute errors of the dynamic versions are lower than the static ones.

Henceforth, the performance benefits introduced by TAFFO do not come at the cost of a significant loss in accuracy.

4.2 Validation of the Error Propagator

As we discussed in Section 2, one of components of the TAFFO *Feedback Estimator* is the *Error Propagator*. This component estimates the quality of the output of the fixed point versions v_s^T and v_d^T, with respect to the floating point version v_s. Our goal is to assess whether such an analysis is appropriate for making decisions on the version to use at runtime.

In addition to the error metrics defined by AxBench– which have been discussed in Section 4.1 –, we also collect the estimation of the Absolute Error metric provided by TAFFO. We show these metrics in Table 3, both for the statically tuned version v_s^T of all benchmarks except *Jmeint*, and for the dynamically tuned version v_d^T thereof. For the *Jmeint* benchmark, they are shown in Table 4.

The worst-case error estimates computed by the analysis are conservative with respect to the measured absolute error. Thus, we have experimentally verified that the approximation provided through range arithmetic is sound in our use cases. In general, the estimated worst-case absolute error on the outputs is overestimated by 2 to 5 orders of magnitudes. This is in line with the results obtained by this kind of error estimators [12]. Additionally, we observe that the computed error properly responds to the modifications performed by the dynamic compilation process. In fact, the estimated error increases or decreases according to the changes in data types performed by TAFFO across the application.

4.3 Evaluation of Continuous Program Optimization

The last evaluation we perform is aimed at verifying the impact of the dynamic compilation approach enabled by LIBVC on the speedup and the error of the AxBench applications. For roughly half of the benchmarks, the speedup, shown in Figure 4, is not greatly affected by the usage of different dynamically compiled versions. In particular, the difference between the highest and the lowest speedup is lower than 0.05 for *Sobel* and *InverseK2J* on the Intel architecture, and for the same benchmarks and *Blackscholes* on the AMD architecture. All other benchmarks achieve a maximum improvement on the speedup higher than 0.05. The highest difference on the Intel platform is reached by the *Blackscholes* benchmark, which obtains an improvement of 0.35 on the speedup from input classes 3 to the others. On the AMD platform, the highest improvement amounts to 0.71 for the *Jmeint* benchmark, between the versions for input classes 1 and 3. In particular, for *Black-Scholes* and *K-means* the maximum difference between the speedup of each version is lower than 0.1%, while *FFT* scores the highest difference of 0.4% on the Intel platform between the versions for input class 0 and input class 2.

The response of the speedup metric to the input partitioning is correlated to whether the policies affect the code coverage of the kernel. In fact, the absolutely highest difference between the speedups is obtained by *Jmeint*, which code coverage is greatly influenced by the input batch. For other

Table 3. Error statistics of the dynamic mixed-precision versions according to AxBench metrics. ARE = Average Relative Error, RMSE = Root-Mean-Square Error.

Bench.	Batch Class	Error (dynamic)			Error (static)			Metric
		absolute		relative	absolute		relative	
		feedback est.	*measured*	*measured*	*feedback est.*	*measured*	*measured*	
Black-Scholes	0	1.45×10^{-5}	1.9×10^{-7}	0.0067 %	3.74×10^{-5}	1.70×10^{-7}	0.0062 %	ARE
	1	3.58×10^{-5}	1.8×10^{-7}	2.6 %	3.74×10^{-5}	2.90×10^{-7}	2.6 %	ARE
	2	5.63×10^{-5}	2.2×10^{-7}	0.30 %	3.74×10^{-5}	4.10×10^{-7}	0.30 %	ARE
	3	1.10×10^{-4}	2.6×10^{-7}	0.0017 %	3.74×10^{-5}	4.60×10^{-7}	0.0023 %	ARE
FFT	0	3.05×10^{-3}	9.08×10^{-5}	0.0017 %	$6.30 \times 10^{+2}$	1.48×10^{-1}	2.8 %	ARE
	1	6.10×10^{-2}	9.26×10^{-4}	0.0088 %	$6.30 \times 10^{+2}$	3.25×10^{-1}	3.0 %	ARE
	2	$4.88 \times 10^{+0}$	2.44×10^{-2}	0.16 %	$6.30 \times 10^{+2}$	9.65×10^{-1}	5.5 %	ARE
Inversek2j	0	2.11×10^{-4}	6.50×10^{-7}	0.0046 %	3.19×10^{-3}	2.15×10^{-3}	0.077 %	ARE
	1	2.30×10^{-5}	2.38×10^{-6}	0.014 %	3.19×10^{-3}	7.97×10^{-5}	0.030 %	ARE
	2	6.44×10^{-5}	1.78×10^{-6}	0.013 %	3.19×10^{-3}	6.07×10^{-5}	0.025 %	ARE
	3	2.00×10^{-5}	1.25×10^{-6}	0.012 %	3.19×10^{-3}	6.83×10^{-5}	0.025 %	ARE
	4	1.41×10^{-3}	2.49×10^{-5}	0.015 %	3.19×10^{-3}	7.25×10^{-5}	0.026 %	ARE
	5	9.57×10^{-5}	1.07×10^{-5}	0.011 %	3.19×10^{-3}	5.58×10^{-5}	0.021 %	ARE
	6	5.63×10^{-3}	6.80×10^{-5}	0.021 %	3.19×10^{-3}	6.80×10^{-5}	0.021 %	ARE
K-means	0	–	–	0.48 %	–	–	0.48 %	RMSE
		1.13×10^{-1}	4.31×10^{-4}	0.10 %	1.13×10^{-1}	4.31×10^{-4}	0.10 %	ARE
	1	–	–	0.25 %	–	–	0.50 %	RMSE
		2.83×10^{-2}	9.40×10^{-5}	0.022 %	1.13×10^{-1}	3.68×10^{-4}	0.086 %	ARE
	2	–	–	0.26 %	–	–	0.52 %	RMSE
		2.83×10^{-2}	9.52×10^{-5}	0.024 %	1.13×10^{-1}	3.83×10^{-4}	0.095 %	ARE
Sobel	0	–	–	0.014 %	–	–	0.018 %	RMSE
		1.51×10^{-5}	4.41×10^{-7}	1.7 %	3.86×10^{-3}	1.11×10^{-4}	49 %	ARE
	1	–	–	0.26 %	–	–	0.26 %	RMSE
		3.86×10^{-3}	1.12×10^{-4}	2.2 %	3.86×10^{-3}	1.12×10^{-4}	2.2 %	ARE

benchmarks, the differences in the speedup across input batches is attributable to the fact that TAFFO generates different code for each version, in terms of shifting operations and data sizes.

We notice even more important differences in terms of quality of the result. The dynamic version (cf. Table 3 and Figure 5) presents lower errors, with striking differences of up to 5 orders of magnitude compared to the static version. This result is a natural consequence of the fact that partitioning the input into batches with similar accuracy requirements results in the generation of tighter ranges in the VRA pass of TAFFO.

Static errors are the same for all input batches, and the comparison with the errors of the v_d^T version shows the strength of the latter approach from this point of view. In fact, the errors measured on the static versions are all lower than those measured on their dynamic counterparts.[3]

In particular, *FFT* shows the greatest improvement, since the errors of the static version are between 2 and 6%, while those of the dynamic version are still below 1%. This is due to the variability between several orders of magnitude of the ranges of values involved in the computation. Such a variability is naturally handled by floating-point formats by explicitly storing the exponent, but is often problematic with fixed-point types. Indeed, the static version of FFT employs fixed-point data

[3]The errors of the v_s^T version of K-means and those of batch 0 of the v_d^T version thereof are the same, because the two versions happen to coincide. This coincidence does not occur with batches 1 and 2. The same happens for batch 1 of Sobel.

Table 4. Error statistics of the dynamic and static mixed-precision versions of the *Jmeint* benchmark. All errors are absolute, except the *miss rate*.

Batch Class	**Parameter**	**Error (dynamic)**		**Error (static)**	
		feedback est.	*measured*	*feedback est.*	*measured*
0	miss rate (%)		0.021		44
	d	8.66×10^{-8}	2.83×10^{-9}	7.95×10^{-4}	1.44×10^{-4}
	d0*d1	1.12×10^{-6}	8.11×10^{-9}	$5.77 \times 10^{+2}$	8.46×10^{-8}
	d0*d2	1.12×10^{-6}	8.16×10^{-9}	$5.77 \times 10^{+2}$	8.69×10^{-8}
	delta_isect_1	$2.99 \times 10^{+1}$	1.66×10^{-4}	$7.51 \times 10^{+7}$	$1.56 \times 10^{+3}$
	delta_isect_2	$2.99 \times 10^{+1}$	1.94×10^{-4}	$7.51 \times 10^{+7}$	$1.56 \times 10^{+3}$
	du0du1	1.54×10^{-5}	5.88×10^{-8}	$9.39 \times 10^{+8}$	7.85×10^{-3}
	du0du2	1.54×10^{-5}	5.89×10^{-8}	$9.39 \times 10^{+8}$	7.88×10^{-3}
	dv0dv1	1.54×10^{-5}	5.89×10^{-8}	$9.39 \times 10^{+8}$	8.36×10^{-3}
	dv0dv2	1.54×10^{-5}	5.88×10^{-8}	$9.39 \times 10^{+8}$	8.37×10^{-3}
	e	8.66×10^{-8}	6.18×10^{-9}	7.95×10^{-4}	1.57×10^{-4}
	f	8.66×10^{-8}	4.80×10^{-9}	7.95×10^{-4}	2.28×10^{-4}
1	miss rate (%)		0.00095		18
	d	8.66×10^{-8}	5.93×10^{-8}	7.95×10^{-4}	2.42×10^{-4}
	d0*d1	1.12×10^{-6}	7.70×10^{-9}	$5.77 \times 10^{+2}$	4.71×10^{-7}
	d0*d2	1.12×10^{-6}	7.72×10^{-9}	$5.77 \times 10^{+2}$	4.80×10^{-7}
	delta_isect_1	$2.99 \times 10^{+1}$	3.51×10^{-3}	$7.51 \times 10^{+7}$	$2.07 \times 10^{+3}$
	delta_isect_2	$2.99 \times 10^{+1}$	3.50×10^{-3}	$7.51 \times 10^{+7}$	$2.06 \times 10^{+3}$
	du0du1	1.54×10^{-5}	1.29×10^{-8}	$9.39 \times 10^{+8}$	1.62×10^{-3}
	du0du2	1.54×10^{-5}	1.29×10^{-8}	$9.39 \times 10^{+8}$	1.61×10^{-3}
	dv0dv1	1.54×10^{-5}	1.28×10^{-8}	$9.39 \times 10^{+8}$	1.80×10^{-3}
	dv0dv2	1.54×10^{-5}	1.28×10^{-8}	$9.39 \times 10^{+8}$	1.81×10^{-3}
	e	8.66×10^{-8}	3.19×10^{-9}	7.95×10^{-4}	2.36×10^{-4}
	f	8.66×10^{-8}	1.81×10^{-9}	7.95×10^{-4}	2.01×10^{-4}
2	miss rate (%)		0.00012		0.00012
	delta_isect_1	$7.51 \times 10^{+7}$	2.27×10^{-4}	$7.51 \times 10^{+7}$	2.27×10^{-4}
	delta_isect_2	$7.51 \times 10^{+7}$	2.19×10^{-4}	$7.51 \times 10^{+7}$	2.19×10^{-4}
	du0du1	$9.39 \times 10^{+8}$	$1.02 \times 10^{+5}$	$9.39 \times 10^{+8}$	$1.02 \times 10^{+5}$
	du0du2	$9.39 \times 10^{+8}$	$1.02 \times 10^{+5}$	$9.39 \times 10^{+8}$	$1.02 \times 10^{+5}$
	dv0dv1	$9.39 \times 10^{+8}$	$1.08 \times 10^{+5}$	$9.39 \times 10^{+8}$	$1.08 \times 10^{+5}$
	dv0dv2	$9.39 \times 10^{+8}$	$1.07 \times 10^{+5}$	$9.39 \times 10^{+8}$	$1.07 \times 10^{+5}$
3	miss rate (%)		0.00009		0.00009
	delta_isect_1	$7.51 \times 10^{+7}$	1.23×10^{-3}	$7.51 \times 10^{+7}$	1.23×10^{-3}
	delta_isect_2	$7.51 \times 10^{+7}$	1.27×10^{-3}	$7.51 \times 10^{+7}$	1.27×10^{-3}
	du0du1	$9.39 \times 10^{+8}$	$3.04 \times 10^{+4}$	$9.39 \times 10^{+8}$	$3.04 \times 10^{+4}$
	du0du2	$9.39 \times 10^{+8}$	$3.05 \times 10^{+4}$	$9.39 \times 10^{+8}$	$3.05 \times 10^{+4}$
	dv0dv1	$9.39 \times 10^{+8}$	$3.37 \times 10^{+4}$	$9.39 \times 10^{+8}$	$3.37 \times 10^{+4}$
	dv0dv2	$9.39 \times 10^{+8}$	$3.35 \times 10^{+4}$	$9.39 \times 10^{+8}$	$3.35 \times 10^{+4}$

types with an integer part large enough not to overflow even with large input values, sacrificing the number of fractional bits and, thus, accuracy. While this loss in precision may be negligible with large inputs, it produces a significant relative error with smaller ones. The dynamic version overcomes this obstacle by generating versions of the kernel tailored to each input-value class, optimizing the trade-off between wideness of the representable value-range, and accuracy. When the input values produce intermediate values in the computation that are too large to be represented

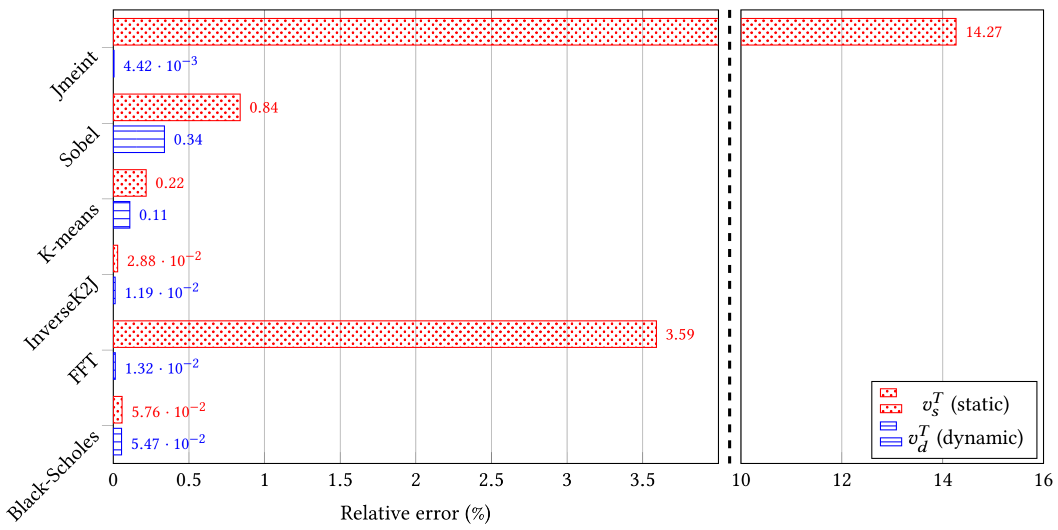

Fig. 5. Overall measured relative errors for each benchmark.

with a fixed point format without an excessive loss of precision, the dynamically optimized kernel resorts to its floating-point version, thus preserving the program's correctness.

We conclude that the dynamic approach in our solution augments in an effective way the precision tuning process performed by TAFFO, as it allows a fixed-point approach to reach much higher levels of precision, at a reasonably low cost in terms of setup overhead.

5 COMPARISON WITH THE STATE-OF-THE-ART

Precision Tuning is a well-established practice in the state-of-the-art, whose roots date back to the definitions of data types. The interest for precision tuning is increasing in the recent years along with the increased popularity of error-tolerant applications, such as image and video processing, data mining, and artificial intelligence. For a comprehensive discussion of approaches to performance/energy optimization, including precision tuning and other approximate computing techniques, the reader is referred to [1, 6].

So far most of the works described in the literature focus on *static precision tuning*, i.e. they aim at providing a single mixed precision version that satisfies the user-defined precision requirements while they optimize a given performance metric. This is the case of precision tuning tools for hardware/software codesign [18, 20, 28]. The hardware specification generated by a typical hardware/software codesign environment requires a single configuration to physically implement. Thus, the static approach fits this use case well.

Static precision tuning has been explored also on general-purpose and high performance computing hardware [7, 11, 21, 26]. In these cases there is no strict limitation on the number of mixed precision version that can coexist at runtime, as they are software implementations. However, only few works have explored this possibility so far. Among the most relevant works there is *PetaBricks* [2]. *PetaBricks* is a programming language featuring a dedicated compiler and runtime environment. They explicitly expose the concept of accuracy levels to the end-user, which may manually provide several precision mix versions. All the user-defined versions coexist at runtime. It is up to the PetaBricks runtime environment to monitor the runtime conditions and to decide which version to execute each time. In contrast, our work focus on the problem of automatically generating appropriate mixed precision version.

Another experimental work in this direction is *DPS (Dynamic Precision Scaling)* [31]. The *DPS* concept is similar to the *PetaBricks* environment, as they have a monitoring component and an autotuning controller. They let the user define policies to control the accuracy loss in the program similarly to the benchmark characterization we suggest. *DPS* aims at exploiting reduced precision floating point hardware unit using the same software version to optimize the power consumption. Such software version has to be available ahead-of-time. On the contrary, we provide the same capabilities with the flexibility of generating the software versions at runtime.

A more recent research work attempts to bring into the field of hardware/software codesign the concept of continuous optimization with precision tuning. Indeed, it is possible to design and exploit computational units with variable precision specifications depending on the software precision requirements. This effort is known as *Transprecision computing* [29] and it is still an open research challenge. The scope of our work is currently limited to existing hardware architectures.

6 CONCLUSIONS

We have introduced a methodology for continuous optimization of programs leveraging reduced precision computation. Our methodology is based on the identification of input classes where the approximation behavior is the same, and for which the same set of data types can therefore be used. We leverage the LIBVC partial dynamic recompilation library, and the TAFFO framework for reduced precision compilation to define a dynamic compiler to support the proposed approach.

We demonstrate the effectiveness of our methodology on a set of benchmarks from the AXBENCH suite, achieving a speedup between 25% and 320% on Intel and AMD server processors, at a very limited cost in terms of accuracy – less than 3% error for each benchmark.

Future works will involve further improvements on the error bound computation to be closer to the actual error. Furthermore, the determination of the input classes could be performed through an automated profile-based approach.

ACKNOWLEDGMENTS

This work was supported in part by the European Union's Horizon 2020 programme under grant 801137 RECIPE.

REFERENCES

[1] Massimo Alioto, Vivek De, and Andrea Marongiu. 2018. Energy-Quality Scalable Integrated Circuits and Systems: Continuing Energy Scaling in the Twilight of Moore's Law. *IEEE J. Emerg. Sel. Topics Circuits Syst.* 8, 4 (Dec 2018), 653–678. https://doi.org/10.1109/JETCAS.2018.2881461

[2] Jason Ansel, Yee L. Wong, Cy Chan, Marek Olszewski, Alan Edelman, and Saman Amarasinghe. 2011. Language and compiler support for auto-tuning variable-accuracy algorithms. In *Int. Symp. on Code Generation and Optimization (CGO 2011).* 85–96. https://doi.org/10.1109/CGO.2011.5764677

[3] Pablo De Oliveira Castro, Chadi Akel, Eric Petit, Mihail Popov, and William Jalby. 2015. CERE: LLVM-Based Codelet Extractor and REplayer for Piecewise Benchmarking and Optimization. *ACM Trans. Archit. Code Optim.* 12, 1, Article 6 (April 2015), 24 pages. https://doi.org/10.1145/2724717

[4] Daniele Cattaneo, Antonio Di Bello, Stefano Cherubin, Federico Terraneo, and Giovanni Agosta. 2018. Embedded Operating System Optimization through Floating to Fixed Point Compiler Transformation. In *21st Euromicro Conference on Digital System Design (DSD).* 172–176. https://doi.org/10.1109/DSD.2018.00042

[5] Stefano Cherubin and Giovanni Agosta. 2018. libVersioningCompiler: An easy-to-use library for dynamic generation and invocation of multiple code versions. *SoftwareX* 7 (2018), 95 – 100. https://doi.org/10.1016/j.softx.2018.03.006

[6] Stefano Cherubin and Giovanni Agosta. 2020. Tools for Reduced Precision Computation: A Survey. *ACM Comput. Surv.* 53, 2, Article 33 (March 2020), 25 pages.

[7] Stefano Cherubin, Giovanni Agosta, Imane Lasri, Erven Rohou, and Olivier Sentieys. 2018. Implications of Reduced-Precision Computations in HPC: Performance, Energy and Error. In *Parallel Computing is Everywhere*, Vol. 32: Advances

in Parallel Computing. 297 – 306. https://doi.org/10.3233/978-1-61499-843-3-297 International Conference on Parallel Computing (ParCo), Sep 2017.

[8] Stefano Cherubin, Daniele Cattaneo, Michele Chiari, Antonio Di Bello, and Giovanni Agosta. 2019. TAFFO: Tuning Assistant for Floating to Fixed point Optimization. *IEEE Embedded Syst. Lett.* 12, 1 (2019), 5–8. https://doi.org/10.1109/LES.2019.2913774

[9] Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. 2017. Rigorous Floating-point Mixed-precision Tuning. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017).* 300–315. https://doi.org/10.1145/3009837.3009846

[10] A. Cohen and E. Rohou. 2010. Processor virtualization and split compilation for heterogeneous multicore embedded systems. In *Design Automation Conference.* 102–107. https://doi.org/10.1145/1837274.1837303

[11] Eva Darulova, Einar Horn, and Saksham Sharma. 2018. Sound Mixed-precision Optimization with Rewriting. In *Proc. 9th ACM/IEEE Int. Conf. on Cyber-Physical Systems (ICCPS '18).* 208–219. https://doi.org/10.1109/ICCPS.2018.00028

[12] Eva Darulova and Viktor Kuncak. 2011. Trustworthy Numerical Computation in Scala. *SIGPLAN Not.* 46, 10 (Oct. 2011), 325–344. https://doi.org/10.1145/2076021.2048094

[13] Eva Darulova and Viktor Kuncak. 2017. Towards a Compiler for Reals. *ACM Trans. Program. Lang. Syst.* 39, 2, Article 8 (March 2017), 28 pages. https://doi.org/10.1145/3014426

[14] Luiz Henrique de Figueiredo and Jorge Stolfi. 2004. Affine Arithmetic: Concepts and Applications. *Numerical Algorithms* 37, 1 (01 Dec 2004), 147–158. https://doi.org/10.1023/B:NUMA.0000049462.70970.b6

[15] Fernando Endo, Damien Couroussé, and Henri-Pierre Charles. 2016. Pushing the Limits of Online Auto-tuning: Machine Code Optimization in Short-Running Kernels. In *Proc. IEEE 10th Int. Symp. on Embedded Multicore/Many-core Systems-on-Chip (MCSoC-16).* Lyon, France.

[16] Grigori Fursin, Anton Lokhmotov, and Ed Plowman. 2016. Collective Knowledge: towards R&D sustainability. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'16).* 864–869.

[17] Davide Gadioli, Gianluca Palermo, and Cristina Silvano. 2015. Application autotuning to support runtime adaptivity in multicore architectures. In *Int. Conf. Embedded Comp. Sys.: Architectures, Modeling, and Simulation.* IEEE, 173–180.

[18] H. Keding, M. Willems, M. Coors, and H. Meyr. 1998. FRIDGE: A Fixed-point Design and Simulation Environment. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '98).* 429–435.

[19] Thomas Kistler and Michael Franz. 2003. Continuous Program Optimization: A Case Study. *ACM Trans. Program. Lang. Syst.* 25, 4 (jul 2003), 500–548. https://doi.org/10.1145/778559.778562

[20] Ki-Il Kum, Jiyang Kang, and Wonyong Sung. 2000. AUTOSCALER for C: an optimizing floating-point to integer C program converter for fixed-point digital signal processors. *IEEE Trans. Circuits Syst. II. Analog Digit. Signal Process.* 47, 9 (Sept 2000), 840–848. https://doi.org/10.1109/82.868453

[21] Michael O. Lam, Jeffrey K. Hollingsworth, Bronis R. de Supinski, and Matthew P. Legendre. 2013. Automatically Adapting Programs for Mixed-precision Floating-point Computation. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing (ICS '13).* 369–378. https://doi.org/10.1145/2464996.2465018

[22] Sparsh Mittal. 2016. A survey of techniques for approximate computing. *Comput. Surveys* 48, 4 (2016), 62.

[23] Tomas Möller. 1997. A fast triangle-triangle intersection test. *Journal of graphics tools* 2, 2 (1997), 25–30.

[24] Ramon E Moore, R Baker Kearfott, and Michael J Cloud. 2009. *Introduction to interval analysis.* Vol. 110. Siam.

[25] Rawzor. [n.d.]. Image Compression Benchmark. http://imagecompression.info/test_images/.

[26] Cindy Rubio-González et al. 2013. Precimonious: Tuning Assistant for Floating-point Precision. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13).* Article 27, 12 pages. https://doi.org/10.1145/2503210.2503296

[27] Cristina Silvano et al. 2019. The ANTAREX domain specific language for high performance computing. *Microprocessors and Microsystems* 68 (2019), 58 – 73. https://doi.org/10.1016/j.micpro.2019.05.005

[28] N. Simon, D. Menard, and O. Sentieys. 2011. ID.Fix-infrastructure for the design of fixed-point systems. In *University Booth of the Conference on Design, Automation and Test in Europe (DATE)*, Vol. 38. http://idfix.gforge.inria.fr

[29] Giuseppe Tagliavini, Stefan Mach, Davide Rossi, Andrea Marongiu, and Luca Benini. 2018. A transprecision floating-point platform for ultra-low power computing. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE).* 1051–1056. https://doi.org/10.23919/DATE.2018.8342167

[30] Amir Yazdanbakhsh, Divya Mahajan, Hadi Esmaeilzadeh, and Pejman Lotfi-Kamran. 2017. AxBench: A Multiplatform Benchmark Suite for Approximate Computing. *IEEE Design Test* 34, 2 (April 2017), 60–68. https://doi.org/10.1109/MDAT.2016.2630270

[31] Serif Yesil, Ismail Akturk, and Ulya R. Karpuzcu. 2018. Toward Dynamic Precision Scaling. *IEEE Micro* 38, 4 (Jul 2018), 30–39. https://doi.org/10.1109/MM.2018.043191123

Received June 2019; revised November 2019