TAFFO: Tuning Assistant for Floating to Fixed Point Optimization

Stefano Cherubin, Daniele Cattaneo, Michele Chiari, Antonio Di Bello, and Giovanni Agosta

***Abstract*—While many approximate computing methods are quite application-dependent, reducing the size of the data representation used in the computation has a more general applicability. We present a tuning assistant for floating to fixed point optimization (TAFFO), an LLVM-based framework designed to assist programmers in the precision tuning of software. We discuss the framework architecture and we provide guidelines to effectively tradeoff precision to improve the time-to-solution. We evaluate our framework on a well-known approximate computing benchmark suite, AxBench, achieving a speedup on 5 out of 6 benchmarks (up to 366%) with only a limited loss in precision (<3% for all benchmarks). Contrary to most related tools, TAFFO supports both C and C++ programs. It is provided as a plugin for LLVM, a design solution that improves significantly the maintainability of the tool and its ease of use.**

***Index Terms*—Approximate computing, fixed point, floating point, precision tuning.**

I. Introduction

FLOATING to fixed point conversion is a key task in the field of embedded application design. Since it is generally performed manually, it can incur in delays and potentially additional errors. No existing open source tool is mature enough for industry adoption, and most of them have little hope of achieving production status, as they are designed as primary research tools, thus employing compiler frameworks that do not benefit from industry-grade maintenance. The quest for tools that can support an increasingly large number of languages is still an open challenge. To bridge these gaps, we introduce a tuning assistant for floating to fixed point optimization (TAFFO), a toolset that automatically converts computation from floating point to fixed point, and that tunes the precision of the resulting fixed point code according to the application goals. TAFFO leverages programmer hints to understand the characteristics of the input data, and performs the conversion to the appropriate data types. The tools are robust enough to support automated conversion for complex C++ benchmarks without rewriting the computational kernels to less expressive languages, such as ANSI C. We verify the effectiveness of TAFFO on the well-known approximate computing benchmark suite AxBench [17].

Manuscript received February 20, 2019; revised April 8, 2019; accepted April 22, 2019. Date of publication April 28, 2019; date of current version February 26, 2020. This manuscript was recommended for publication by P. R. Panda. *(Corresponding author: Stefano Cherubin.)*

The authors are with Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, 20133 Milan, Italy (e-mail: stefano.cherubin@polimi.it; daniele3.cattaneo@mail.polimi.it; michele.chiari@polimi.it; antonio.dibello@mail.polimi.it; agosta@acm.org).

Digital Object Identifier 10.1109/LES.2019.2913774

Key Contributions: TAFFO provides two main improvements over the state-of-the-art. First, TAFFO allows programmers to equally apply fine-grained precision tuning to a wide range of programming languages, whereas most current competitors are limited to C. This improvement is a direct consequence of the structure of TAFFO, which is implemented as a plugin for the industry-grade LLVM compiler framework. Moreover, it is easily applicable to most embedded (see [2]) and high performance applications, and it allows easy maintenance and extensions. Second, TAFFO provides an integrated performance estimation of the mixed precision code, which is performed statically and which does not require any program simulation, thus allowing the compiler to decide whether to switch to fixed point or not.

II. State-of-the-Art

Tools whose purpose is to enable reduced precision computation are hardly the main focus of developers when it comes to writing new software and to configuring a programming toolchain. Therefore, such tools target already existing specialized toolchains. *FRIDGE* [7] is a framework that represents one of the earliest effort to provide automatic conversion of floating point code into its fixed point equivalent. Whilst *FRIDGE* provides a source-level precision mix specification, TAFFO allows a more fine-grained tuning by operating within the intermediate representation of the compiler.

Autoscaler For C [8] estimates the range of each value by instrumenting and by profiling the intermediate representation of the compiler. The unavailability of both the source code of the framework and that of the hardware platforms they rely on in their work prevents us from comparing TAFFO with Autoscaler.

Precimonious [14] is a tool derived from the LLVM [10] compiler framework. It supports only the standard C floating point data types, and it does not deal with any fixed point format.

Other works, such as *Rosa* [4], feature static analysis of the code to formally guarantee safe bounds on the introduced error. However, they introduce a contract-based specification language to allow the programmer to describe proper preconditions and precision requirements for each function. Thus, their approach is not the best for the embedded system nor for the high performance computing (HPC) domains.

CRAFT [9] is a precision tuning framework whose goal is the minimization of the instructions that exploit the floating point double precision format. It replaces those instructions with equivalents based on the single precision format. It does not consider fixed point data types. Even though dynamic analysis can reduce the search space, *CRAFT* requires multiple

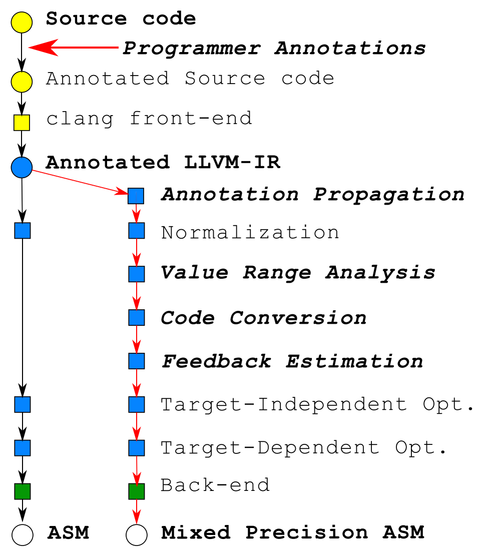

Fig. 1. Outline of the compilation pipeline using the CLANG compiler front-end with and without TAFFO. We highlight with red arrows the TAFFO pipeline stages. Yellow elements refer to source code and the compiler front-end. Blue elements refer to passes of the optimizer. Finally, the green element represents the compiler back-end.

profile runs to evaluate the precision mix candidates. Holistic approximate computing frameworks [11], [16], [19] have limited or no support for mixed precision tuning. We propose a more flexible and lightweight framework. Flexibility comes from the capability to address an arbitrary large number of source languages. Although an approach based on static analysis provides less strict bounds on the runtime values, the achieved result is proved to be safer with respect to profile-based approaches, which depend on input representativeness. Moreover, the combination of profile-based approaches with whole-program analysis proved to be extremely time-consuming [15].

III. Proposed Solution

The implementation of TAFFO does not require any modification of the standard compiler toolchain. As shown in Fig. 1, TAFFO introduces new compiler passes without modifying neither the existing compiler passes nor the compiler front-end—as most competitors do [4], [7], [8].

The complexity of finding the best precision mix for a given program grows exponentially with the number of values to be tuned and with the possible precision levels. An exhaustive exploration is feasible only with a small number of values, and with a reduced selection of precision levels. Although the space of possible solutions is wide, a large portion of it is composed of precision mix permutations of marginal code. These code versions can be safely removed from the exploration. To this end, we rely on the programmer's knowledge of the application. We ask the programmer to restrict the scope of TAFFO's analysis and transformation via annotations on the source code. The annotation-based approach is a common practice in compiler construction to forward pieces of information toward further compiler stages. We use compiler metadata to keep information about the range of possible values for each variable. TAFFO requires the user to specify the

```
float a __attribute((annotate("taffo␣20␣12")));
float b __attribute(
    (annotate("taffo␣7␣25␣signed␣0.4␣0.9␣1e-8")));
float c __attribute((annotate("range␣0.05␣1␣0")));
```

Listing 1. Example of annotated C code where the programmer is asking to transform the variables a and b to a fixed point, and is providing the value range for the c variable.

range only on the input values. Our framework makes sure these values gets propagated to all the intermediate values. We then decide the allocation of data types and—in case of fixed point—the position of the point for each value. This process takes place in the intermediate representation of the compiler. Thus, we allow each intermediate value to be represented with a potentially different data format.

After the code conversion, we evaluate the converted code to check if it actually represents an improvement with respect to the baseline. This process—called *Feedback Estimation*—entails both a functional and a performance evaluation. The error bounds are computed via a data flow analysis whilst the performance is estimated via a previously computed platform-dependent performance model.

A. Annotations

Annotations specify which portion(s) of the code should be analyzed for precision tuning. The insertion of `annotate` attributes is natively supported by the CLANG compiler. Thus, there is no need to extend the compiler front-end for any language extension. This approach holds also with other front-ends for the LLVM compiler infrastructure.

As shown in Listing 1, the programmer may specify additional parameters to TAFFO. Indeed, the annotation on line 3 adds the value range, and the value uncertainty. If the programmer defines only the range, TAFFO can derive the most appropriate fixed point format on its own. It is possible to convert the computations whose result is used by the annotated variable by using "`force_taffo`" instead of "`taffo.`"

B. Data Type Allocation and Code Conversion

After the propagation of the annotations, we run a data flow analysis based on interval arithmetic [12] to propagate the value ranges to all the intermediate values defined in the LLVM-IR. Let $r_i = <l, u>$ be the range of the variable v_i with lower bound l and upper bound u. Equation (1) relates the range of a variable with the minimum bit width required

$$n_{\text{bits}} = \begin{cases} \lceil \log_2 \max(\text{abs}(l), \text{abs}(u)) \rceil & l \geq 0 \\ \lceil \log_2 \max(\text{abs}(l), \text{abs}(u)) \rceil + 1 & l < 0. \end{cases} \quad (1)$$

Our solution transforms the LLVM-IR as if a type change was performed in the original source code. Integer and fractional parts are logically partitioned so as to prevent *a priori* any overflow problem. The transformation generates code to perform the computation with fixed point arithmetic alongside the already existing floating point code. We allocate a separate memory location for the fixed point values. The code conversion process supports the interprocedural transformation of memory operation on scalar, array, and pointers values via `load`, `store`, and `getelementptr` instructions. By their nature, the conversion of constants—both literals and in-memory constants—do not require any memory duplication.

Function calls are handled via duplication of the function in the LLVM-IR. We apply the same code conversion procedure to the cloned function as if its parameters were annotated by the programmer. When the code conversion pass meets an instruction with an unknown conversion—as in the case of calls to an external function—it restores the original data type and it leaves that instruction unchanged. This *fallback* behavior guarantees a strong preservation of the program semantic.

In the absence of fallbacks, all the uses of the floating point values should have been replaced by their fixed point equivalent. Finally, we schedule a dead code elimination (DCE) optimization pass from the LLVM compiler infrastructure, which safely removes all the floating point instructions—including the `allocas`.

C. *Feedback Estimation*

We evaluate the mixed precision LLVM-IR bitcode with two metrics. First, we run a static error propagation analysis to project the truncation error we introduced with the fixed point computation on the output. The propagation is performed by representing the absolute error associated to each LLVM-IR instruction by means of *affine forms* [5], combined with the intervals resulting from the value range analysis (see [4]). This approach allows us to keep track of each single error source, exploiting error cancellation when possible. Nonlinear operations such as mathematical functions from the C standard library are treated with linear approximations, as suggested in [5] and [3]. Whenever it is possible we exploit the LLVM facilities to unroll loops on a copy of the code which is later discarded, to analyze the error they introduce.

The second evaluation is based on a platform-dependent performance model. The goal of this step is to estimate the impact of the type cast overhead, and the performance gain due to the precision lowering. To this end, we train a performance model using code statistics before and after the code conversion. We rely on machine-learning tools from `Scikit-learn` [13] to analyze the impact of the introduction and removal of different instructions. We identify 26 classes of LLVM-IR instructions that can be used as features in statistical learning. For each class, the relevant feature is the change in instruction frequency from the floating point version of the code to its mixed precision version. As the target response, we consider the ratio $T_{\text{fix}}/T_{\text{flt}}$ between the execution times of the fixed point conversion and that of the original code. We use the results of the conversion of a set of small computational kernels as the training set for a range of ensemble classification and regression methods. We consider the most stable approach to build the performance estimation model.

After the code conversion, we can decide whether the mixed precision satisfies the user requirements on the error and whether it can provide a speedup over the baseline.

IV. Experimental Evaluation

A. *Experimental Setup and Benchmarks*

We evaluate TAFFO on two different types of hardware: an HPC-like computer architecture and an embedded systems' development board: a server NUMA node (AMD) featuring four Six-Core AMD Opteron 8435 CPUs (@2.6 GHz, AMD K10 microarchitecture), with 128 GB of DDR2 memory (@800 MHz); and an STM3220G-EVAL board (f207) featuring a 120 MHz ARM Cortex M3 microcontroller without hardware floating point support. This board has 1 MB of on-chip flash memory, and 2 MB of off-chip SRAM.

To assess the effectiveness of TAFFO we exploit the set of CPU applications from the AxBench [17] benchmark suite, which is composed of representative error-tolerant applications (a key feature, since we need to objectively assess the precision impact of our transformation). The benchmark suite provides metrics to measure the quality of the result for each application. In particular, *Blackscholes*, *FFT*, and *Inversek2j* use the average relative error (ARE); *Jmeint* uses miss rate (MR); *K-means* and *Sobel* use the root mean square error (RMSE) of the image. We do not consider the *JPEG Encoding* benchmark as it does not feature any floating-point kernel.

All the experiments on the AMD node use the largest data set available in the AxBench benchmark suite. On the *f207* node we used the largest data set from AxBench that could fit the memory. In particular, input images for *K-means* and *Sobel* have been scaled down to 256×256 resolution, *Blackscholes* runs on the 10K data set, *FFT* uses the 65 536 data set, *Inversek2j* uses the 100K data set, and *Jmeint* uses the 10K data set.

B. *Model Construction*

We selected the PolyBench/C benchmark suite [18] as training set for the performance model of the feedback estimator. For each kernel we run three different versions: *vanilla*, *optimized*, and *mixed*. The *vanilla* version exploits only floating point computations using the `binary32` data format from the IEEE-754 standard [6]. The *optimized* version is obtained by converting to fixed point computation the whole kernel via TAFFO. Finally, the *mixed* version is obtained by randomly converting a section of the kernel to exploit fixed point computation, and explicitly forcing the rest of the kernel to use floating point computation. We consider only *mixed* versions that feature at least one type cast in the kernel. For each version we consider a set of features comprising the instruction count per class of instructions, relative to the original float point version. The response metric selected for regressor estimation is the ratio $T_{\text{fix}}/T_{\text{float}}$ between the execution times of the converted and of the original code. Based on the outcome of the training and of the test, we select a Bagging ensemble [1] using k-neighbors estimators which proved more stable than the other candidates.

We validate the selected Bagging ensemble of k-neighbors estimators, trained on the PolyBench/C benchmark suite, on the AxBench benchmarks. As shown in Fig. 2, although for some benchmarks the prediction is inaccurate (*FFT*, *Inversek2j*), only in one case it leads to an incorrect classification. The regression on the execution time ratio provides a useful tool to understand the effectiveness of the prediction, and may in the future be used to better tune the conversion. We expect the predictor accuracy to improve by using real-world applications instead of small benchmarks for model training.

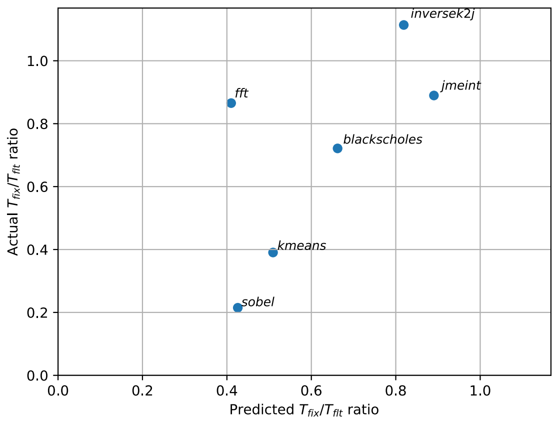

Fig. 2. Comparison between measured and estimated $T_{\mathrm{fix}}/T_{\mathrm{flt}}$ ratio using a Bagging ensemble method to boost the accuracy of a K neighbor regressor.

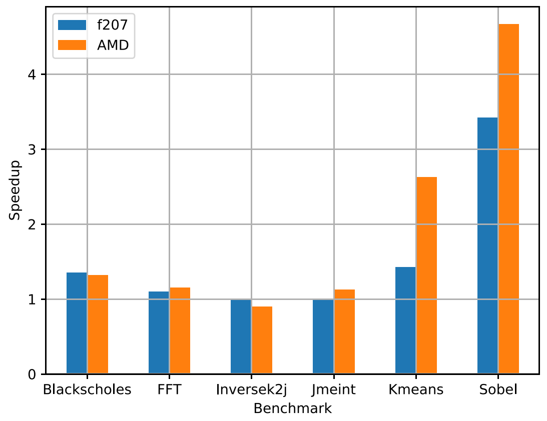

Fig. 3. Measured speedup ($T_{\mathrm{flt}}/T_{\mathrm{fix}}$) of the mixed precision versions over the reference floating point implementation.

TABLE I
QUALITY OF THE RESULT FOR THE MIXED PRECISION VERSIONS ACCORDING TO THE AXBENCH METRICS

Benchmark	$Error_{feedback}$	$Error_{abs}$	$Error_{rel}$	metric
Blackscholes	0.005579455	0.00000006	0.4502%	ARE
FFT	0.079661725	0.02281871	1.2478%	ARE
Inversek2j	0.0005	0.0000485	0.0051%	ARE
Jmeint	0.09673511	0.01654037	0.0118%	MR
K-means	-	-	2.8583%	RMSE
Sobel	-	-	0.0316%	RMSE

C. Result Discussion

Fig. 3 shows the measured speedup achieved by the mixed precision versions created by TAFFO with respect to the corresponding floating point reference implementations. The only application that does not benefit from the mixed precision approach is *Inversek2j*, whereas all the other benchmarks show speedups ranging from 12.5% to 366.8% on the HPC AMD node. Although the speedup is not as important as in the AMD platform, the trend is confirmed also on the embedded system *f207* node—with the exception of *Jmeint*, which has only 0.02% speedup. Table I shows the impact of the error introduced by the precision reduction on the output, for all applications. *K-means* and *Sobel* applications do not have a single-value output. We did not show the absolute error for those applications, as it is locally pointless, and it has to be compared on the whole output. In all the other cases, when we compare the absolute error computed on the output against the error provided by the static feedback estimation, we can observe that the prediction is always conservative.

V. CONCLUSION

We introduce TAFFO, an extension of the LLVM compiler framework to support precision tuning. TAFFO enables speedups in 5 out of the 6 benchmarks, at the cost of limited error (<3%).

Future research directions are mostly related to the improvement of feedback estimation. Tighter error bounds for loops, based on the analysis introduced in [4] could be computed, and more prediction algorithms could be explored.

REFERENCES

[1] L. Breiman, “Bagging predictors,” *Mach. Learn.*, vol. 24, no. 2, pp. 123–140, Aug. 1996.
[2] D. Cattaneo, A. D. Bello, S. Cherubin, F. Terraneo, and G. Agosta, “Embedded operating system optimization through floating to fixed point compiler transformation,” in *Proc. 21st Euromicro Conf. Digit. Syst. Design (DSD)*, 2018, pp. 172–176.
[3] E. Darulova and V. Kuncak, “Trustworthy numerical computation in scala,” *SIGPLAN Notices*, vol. 46, no. 10, pp. 325–344, Oct. 2011.
[4] E. Darulova and V. Kuncak, “Towards a compiler for reals,” *ACM Trans. Program. Lang. Syst.*, vol. 39, no. 2, p. 8, Mar. 2017.
[5] L. H. de Figueiredo and J. Stolfi, “Affine arithmetic: Concepts and applications,” *Numer. Algorithms*, vol. 37, nos. 1–4, pp. 147–158, Dec. 2004.
[6] IEEE Computer Society Standards Committee. *Floating-Point Working Group of the Microprocessor Standards Subcommittee, IEEE Standard for Floating-Point Arithmetic*, IEEE Standard 754-2008, pp. 1–70, Aug. 2008.
[7] H. Keding, M. Willems, M. Coors, and H. Meyr, “FRIDGE: A fixed-point design and simulation environment,” in *Proc. Conf. Design Autom. Test Europe*, Paris, France, 1998, pp. 429–435.
[8] K.-I. Kum, J. Kang, and W. Sung, “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.*, vol. 47, no. 9, pp. 840–848, Sep. 2000.
[9] M. O. Lam, J. K. Hollingsworth, B. R. de Supinski, and M. P. Legendre, “Automatically adapting programs for mixed-precision floating-point computation,” in *Proc. 27th Int. ACM Conf. Supercomput. (ICS)*, Eugene, OR, USA, 2013, pp. 369–378.
[10] C. Lattner and V. Adve, “LLVM: A compilation framework for lifelong program analysis & transformation,” in *Proc. Int. Symp. Code Gener. Optim.*, San Jose, CA, USA, 2004, pp. 75–86.
[11] S. Mitra, M. K. Gupta, S. Misailovic, and S. Bagchi, “Phase-aware optimization in approximate computing,” in *Proc. Int. Symp. Code Gener. Optim.*, Austin, TX, USA, 2017, pp. 185–196.
[12] R. E. Moore *et al.*, *Introduction to Interval Analysis*. Philadelphia, PA, USA: SIAM, 2009.
[13] F. Pedregosa *et al.*, “Scikit-learn: Machine learning in python,” *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, Oct. 2011.
[14] C. Rubio-González *et al.*, “Precimonious: Tuning assistant for floating-point precision,” in *Proc. Int. Conf. High Perform. Comput. Network. Storage Anal. (SC)*, Denver, CO, USA, Nov. 2013, pp. 1–12.
[15] C. Rubio-González *et al.*, “Floating-point precision tuning using blame analysis,” in *Proc. 38th Int. Conf. Softw. Eng. (ICSE)*, Austin, TX, USA, 2016, pp. 1074–1085.
[16] A. Sampson *et al.*, “ACCEPT: A programmer-guided compiler framework for practical approximate computing,” Univ. Washington, Seattle, WA, USA, Rep. UW-CSE-15-01-01, Apr. 2015.
[17] A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, and P. Lotfi-Kamran, “AxBench: A multiplatform benchmark suite for approximate computing,” *IEEE Des. Test.*, vol. 34, no. 2, pp. 60–68, Apr. 2017.
[18] T. Yuki, “Understanding PolyBench/C 3.2 kernels,” in *Proc. Int. Workshop Polyhedral Compilation Tech. (IMPACT)*, 2014, pp. 1–5.
[19] Q. Zhang, F. Yuan, R. Ye, and Q. Xu, “Approxit: An approximate computing framework for iterative methods,” in *Proc. Design Autom. Conf. (DAC)*, San Francisco, CA, USA, Jun. 2014, pp. 1–6.