Fixed Point Exploitation via Compiler Analyses and Transformations

Daniele Cattaneo[1], Antonio Di Bello[2],
Michele Chiari[3], Stefano Cherubin[4],
Giovanni Agosta[5]

DEIB, Politecnico di Milano, via Ponzio 34/5, 20133, Milano, Italy

ABSTRACT

Every data type offers different precision guarantees. However, complex data types have a higher memory and arithmetic impact with respect to simpler ones. Deciding which data type is best suited for each variable in the program is a difficult task. Traditionally developers prefer to use a conservative approach by using floating point data types. However, fixed point representations can improve code performance at a limited cost in terms of precision. We introduce *TAFFO: Tuning Assistant for Floating to Fixed point Optimization*, a compiler-level precision tuning tool. It is shipped as a set of plugin modules for the LLVM compiler infrastructure [LA04].

KEYWORDS: Fixed Point; Compiler Transformation; Precision Tuning

1 Introduction

Finite-precision is an unavoidable approximation of real numbers with the current technology. Determining which finite-precision representation to exploit for each variable in the program is a difficult task. Fixed point computation represents a key feature in the design process of embedded applications. It is also exploited as a mean to data size tuning for HPC tasks [CAL+17]. However, the common practice is to initially develop algorithms using floating point data types, and later translate them into fixed point equivalents. This manual conversion is a task that does not efficiently scale with the lines of code to be tuned.

We propose TAFFO (*Tuning Assistant for Floating to Fixed Point Optimization*) [CDBC+19], a framework that assists the end-user in the precision tuning task by automating this process. TAFFO statically analyzes the code of an application. It then applies the most appropriate precision reduction to fixed point data types.

[1]E-mail: daniele3.cattaneo@mail.polimi.it
[2]E-mail: antonio.dibello@mail.polimi.it
[3]E-mail: michele.chiari@polimi.it
[4]E-mail: stefano.cherubin@polimi.it
[5]E-mail: agosta@acm.org

2 The Architecture of TAFFO

TAFFO is structured as a set of LLVM passes. Its packaging allows it to be used as a plugin for the `opt` tool. TAFFO does not dependent on the compiler front-end, and require no modifications to LLVM itself. Other solutions found in literature – for example [K+98, K+00, D+18, DK17] – are based on specific front-ends.

In order to restrict the scope of the analysis and transformations performed by TAFFO, as a first step we require the programmer to annotate the source code to the program that they want to transform. More specifically, for each input variable in the program, the programmer will specify the range of its possible values. From this information, TAFFO will automatically determine the region of code to transform via a data flow analysis.

The source code modified as described is then fed into the front-end. Then, the intermediate representation produced is processed by the TAFFO pipeline, which consists of the *Initialization* pass, the *Value Range Analysis* pass, the *Data Type Allocation* pass, the *Conversion* pass, and the *Feedback Estimator* pass. Finally, the rest of LLVM performs other conventional optimizations, and the compiled code is produced by one of LLVM's standard back-ends.

The Initialization, Value Range Analysis and Conversion Passes The *Initialization* pass is responsible of parsing programmer-inserted annotations and converting them into LLVM metadata. Then, TAFFO runs a code analysis based on interval arithmetic [M+09] to propagate the value ranges to all the intermediate values defined in the LLVM-IR. This step is performed by the *Value Range Analysis* pass. From these ranges we derive the minimum data width required by each value via the *Data Type Allocation* pass. Integer and fractional parts are logically partitioned so as to prevent a priori any overflow problem.

After that, the *Conversion* pass transforms the LLVM-IR as if a type change was performed in the original source code. We allocate separate memory locations for the fixed point values. The conversion of constants – both literals and in-memory constants – does not require any memory duplication. TAFFO currently supports the interprocedural transformation of memory operation on scalar, array, and pointers values via `load`, `store`, and `getelementptr` instructions.

Function calls are handled via duplication of the function in the LLVM-IR. The duplicated functions are subject to conversion as well. 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* behaviour preserves the program semantics in case of uncertainty.

The Feedback Estimator Pass After the code conversion, TAFFO decides whether the mixed precision satisfies the user requirements on the error and whether it can provide a speedup over the baseline, through the *Feedback Estimator* pass. To this end, the *Error Estimation* step and *Performance Estimation* step statically evaluate the mixed precision LLVM-IR bitcode.

The *Error Estimation* step propagates the truncation error we introduced with the fixed point computation on the output by employing *affine forms* [dFS04]. An *affine form* is defined as the representation of a variable x as in Equation 1, where x_0 is the *central* value, and each

ϵ_i is a *noise symbol* of magnitude x_i.

$$x = x_0 + \sum_{i=1}^{n} x_i \epsilon_i \tag{1}$$

By combining affine forms with the intervals resulting from the value range analysis (c.f. [DK17]) it is possible to keep track of each single error source, exploiting error cancellation when possible. In particular, we associate a range r_x and an error e_x to each instruction, represented as a zero-centered affine form. Non-linear operations such as mathematical functions from the C standard library have to be approximated as suggested in [dFS04] and [DK11]. Loops and other complex control structures are evaluated by exploiting the LLVM facilities to unroll loops on a copy of the code which is later discarded.

The *Performance Estimation* is based on a platform-dependent performance model, based on machine-learning tools from `Scikit-learn` [P+11]. The goal of this step is to estimate the impact of the type cast overhead, and the performance gain due to the precision lowering. 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_{fix}/T_{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, and then we choose the most stable approach among the candidates to build the final performance estimation model.

3 Success Stories

We evaluated TAFFO on a variety of benchmark applications and use cases. In this Section we briefly summarize the use cases we analyzed so far.

Real-Time Operating System We evaluate the use of TAFFO in a performance-critical component of a real-time operating system: the scheduler. In particular, we compare the results achieved via manual conversion of the MIOSIX [LMPT13] scheduler based on control-theory against those achieved with our framework [CDBC+18]. Our experimental campaign include three development boards for embedded systems: one of them featuring hardware support for floating point computation, and two of them without such capability. Results show important speedup – up to $3.1\times$ – on the boards without hardware floating point support. The functional behaviour of the scheduler is completely preserved and no error is introduced in the computation.

Approximate Computing Benchmarks We evaluate TAFFO [CCC+19] by exploiting the set of CPU applications from the AXBENCH [Y+17] benchmark suite, which is composed of representative real-world error-tolerant applications. Additionally, AXBENCH provides metrics to measure the quality of the result for each application. Hardware-wise, we employ two different platforms: an embedded systems' development board and an HPC-like computer architecture. Most of the benchmark applications show important speedups – up to 366.8% on the HPC-like node. The error due to the floating point to fixed point conversion is kept within the 3% threshold for all the benchmark applications.

4 Conclusions

We presented TAFFO, a plugin-based extension of the LLVM compiler framework to provide to the programmers support when performing precision tuning using fixed point data types. TAFFO proved to be effective on a wide range of benchmark applications. It has been successfully applied into the embeded systems, and in the HPC domains.

In the future we aim at improving the analysis performed by TAFFO in order to provide a more accurate prediction of the dynamic range of values, and a tighter error margin.

References

[CAL+17] Stefano Cherubin, Giovanni Agosta, Imane Lasri, Erven Rohou, and Olivier Sentieys. Implications of Reduced-Precision Computations in HPC: Performance, Energy and Error. In *International Conference on Parallel Computing (ParCo)*, Sep 2017.

[CCC+19] Stefano Cherubin, Daniele Cattaneo, Michele Chiari, Antonio Di Bello, and Giovanni Agosta. TAFFO: Tuning assistant for floating to fixed point optimization. *IEEE Embedded Systems Letters*, 2019.

[CDBC+18] Daniele Cattaneo, Antonio Di Bello, Stefano Cherubin, Federico Terraneo, and Giovanni Agosta. Embedded operating system optimization through floating to fixed point compiler transformation. In *21st Euromicro Conference on Digital System Design (DSD)*, volume 00, pages 172–176, Aug 2018.

[CDBC+19] Daniele Cattaneo, Antonio Di Bello, Michele Chiari, Stefano Cherubin, and Giovanni Agosta. Fixed point exploitation via compiler analyses and transformations: Poster. In *Proceedings of the 16th ACM International Conference on Computing Frontiers*, CF '19, pages 292–294, Apr 2019.

[D+18] Eva Darulova et al. Sound mixed-precision optimization with rewriting. In *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems*, ICCPS '18, pages 208–219, 2018.

[dFS04] Luiz Henrique de Figueiredo and Jorge Stolfi. Affine arithmetic: Concepts and applications. *Numerical Algorithms*, 37(1):147–158, Dec 2004.

[DK11] Eva Darulova and Viktor Kuncak. Trustworthy numerical computation in scala. *SIGPLAN Not.*, 46(10):325–344, October 2011.

[DK17] Eva Darulova and Viktor Kuncak. Towards a compiler for reals. *ACM Trans. Program. Lang. Syst.*, 39(2):8:1–8:28, March 2017.

[K+98] H. Keding et al. FRIDGE: A fixed-point design and simulation environment. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '98, pages 429–435, 1998.

[K+00] Ki-Il Kum et al. AUTOSCALER for C: an optimizing floating-point to integer C program converter for fixed-point digital signal processors. *IEEE Trans. on Circuits and Systems II: Analog and Digital Signal Processing*, 47(9):840–848, Sept 2000.

[LA04] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. Int'l Symp. on Code Generation and Optimization*, 2004.

[LMPT13] Alberto Leva, Martina Maggio, Alessandro V. Papadopoulos, and Federico Terraneo. *Control-Based Operating System Design*. Institution of Engineering and Technology, 2013.

[M+09] Ramon E Moore et al. *Introduction to interval analysis*. Siam, 2009.

[P+11] F. Pedregosa et al. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[Y+17] Amir Yazdanbakhsh et al. AxBench: A multiplatform benchmark suite for approximate computing. *IEEE Design Test*, 34(2):60–68, April 2017.