Fixed Point Exploitation via Compiler Analyses and Transformations

D. Cattaneo, A. Di Bello, M. Chiari, S. Cherubin, G. Agosta
Dipartimento di Elettronica, Informazione e Bioingegneria – Politecnico di Milano
{daniele3.cattaneo},{antonio.dibello}@mail.polimi.it
{michele.chiari},{stefano.cherubin},{giovanni.agosta}@polimi.it

CCS CONCEPTS

• **Software and its engineering** → **Compilers**; • **Computer systems organization** → *Embedded software.*

ACM Reference Format:
D. Cattaneo, A. Di Bello, M. Chiari, S. Cherubin, G. Agosta. 2019. Fixed Point Exploitation via Compiler Analyses and Transformations. In *Proceedings of the 16th conference on Computing Frontiers (CF '19), April 30-May 2, 2019, Alghero, Italy.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3310273.3323424

1 INTRODUCTION

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 [2]. Since the conversion from floating point to fixed point is generally performed manually, it is time-consuming and error-prone. However, the full automation of such task is currently unfeasible, as existing open source tools are not mature enough for industry adoption. To bridge this gap, we introduce our Tuning Assistant for Floating point to Fixed point Optimization (TAFFO). TAFFO is a toolset of LLVM compiler plugins that automatically converts computations from floating point to fixed point. TAFFO leverages programmer hints to understand the characteristics of the input data, and then performs the code conversion using the most appropriate data types. 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. Moreover, it is easily applicable to most embedded [1] and high performance applications [10, 11], and it allows easy maintenance and extensions.

CF '19, April 30-May 2, 2019, Alghero, Italy

ACM ISBN 978-1-4503-6685-4/19/05.
https://doi.org/10.1145/3310273.3323424

2 PROPOSED SOLUTION

The structure of TAFFO introduces new compiler passes in the LLVM compilation flow without modifying neither the existing compiler passes nor the compiler front-end. This approach differentiates TAFFO from other solutions found in literature [3, 5, 7, 8]. The complexity of finding the best precision mix for a given program is exponential in the number of values to be tuned and in the possible precision levels. We ask the programmer to annotate the source code to limit the scope of TAFFO's analyses and transformations. The user to specify the range only on the input values and TAFFO propagates them to all the intermediate values. The framework operates with the intermediate representation of the compiler. TAFFO performs the allocation of data types and – in case of fixed point – the allocation of the position of the point for each value. Thus, each intermediate value may be represented with a potentially different data format. Then, TAFFO replaces the original code with equivalent code that exploits the previously defined data types.

After the code conversion, the framework analyzes the converted code to check if it actually represents an acceptable improvement with respect to the baseline, through a functional evaluation. The error bounds are computed via a data flow analysis, namely the *Error Estimation.*

2.1 Code Analysis and Conversion

The programmer-inserted annotations are parsed and converted into LLVM metadata. Then, TAFFO runs a code analysis based on interval arithmetic [9] to propagate the value ranges to all the intermediate values defined in the LLVM-IR. From these ranges we derive the minimum data width required by each value. Integer and fractional parts are logically partitioned so as to prevent a priori any overflow problem.

After that, our solution 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 – do 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.

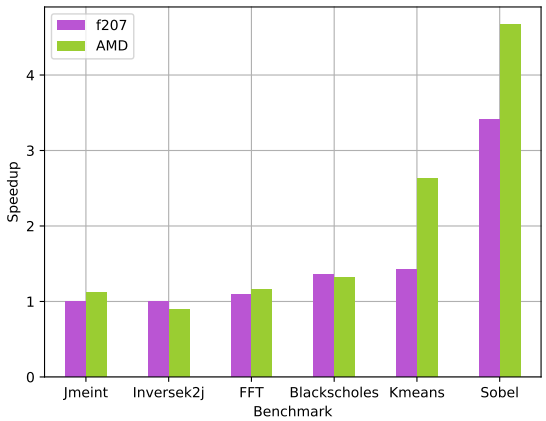

Figure 1: Speedup (T_{flt}/T_{fix}) of the mixed precision versions over the reference floating point version.

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.

2.2 Error Estimation

After the code conversion, we can decide whether the mixed precision satisfies the user requirements on the error. To this end, the *Error Estimation* step statically evaluates the mixed precision LLVM-IR bitcode. We run an 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* [6], combined with the intervals resulting from the value range analysis (c.f. [5]). This approach allows us to keep track of each single error source, exploiting error cancellation when possible.

An *affine form* [6] is the representation of a variable x as

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

where x_0 is the *central* value, and each ϵ_i is a *noise symbol* of magnitude x_i. Each noise symbol is a symbolic variable representing a single error source. Affine forms are combined with the intervals resulting from the value range analysis, as detailed in [5]. In this setting, to each instruction x we associate a range r_x and an error e_x, represented as a zero-centered affine form. Sum and subtraction are performed term-wise on the magnitudes of corresponding noise symbols. This allows us to exploit the possible cancellation of errors due to the same noise symbol. The error of a `mul` instruction with operands x and y is computed as

$$\begin{aligned} x \times y &= (r_x + e_x)(r_y + e_y) \\ &= r_x \times r_y + r_x \times e_y + r_y \times e_x + e_x \times e_y. \end{aligned}$$

Division is treated similarly, that is a multiplication by the inverse of the divisor. Non-linear operations such as mathematical functions from the C standard library are treated with linear approximations, as suggested in [6] and [4]. Whenever it is possible we exploit the LLVM facilities to unroll loops on a copy of the code which is later discarded, in order to analyze the error they introduce. The unrolling is performed on a copy of the loop, which is discarded after the analysis.

3 EXPERIMENTAL EVALUATION

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

Table 1: Quality of the result for the mixed precision versions according to the AxBench metrics

We evaluate TAFFO on two different types of hardware: an HPC-like computer architecture (*AMD* node) and an embedded systems' development board (*f207* node). We exploit the set of CPU applications from the AxBench [12] 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). This benchmark suite also provides metrics to measure the quality of the result for each application. We exclude the *jpeg* benchmark in AxBench from our assessment as it does not perform any floating point calculation. Thus, this evaluation includes 6 benchmarks. All AxBench benchmarks but *inversek2j* 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. Concerning the error estimator, we observe that its error-bounds prediction is always conservative. This guarantees its reliability in assessing whether the accuracy of the converted code is sufficient.

4 CONCLUSIONS

We presented TAFFO, a plugin-based extension of the LLVM compiler framework to provide to the programmers support when performing precision tuning with fixed point data types. TAFFO enables speedups in 5 out of the 6 AxBench benchmarks that include significant floating point computation, at the cost of limited error ($<3\%$ for all benchmarks, considering the application-specific metric provided by AxBench).

REFERENCES

[1] 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)*, Vol. 00. 172–176. https://doi.org/10.1109/DSD.2018.00042

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

[3] Eva Darulova et al. 2018. Sound Mixed-precision Optimization with Rewriting. In *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS '18)*. 208–219. https://doi.org/10.1109/ICCPS.2018.00028

[4] 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

[5] 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

[6] 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

[7] H. Keding et al. 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.

[8] Ki-Il Kum et al. 2000. 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 (Sept 2000), 840–848. https://doi.org/10.1109/82.868453

[9] Ramon E Moore et al. 2009. *Introduction to interval analysis.* Siam.

[10] Cristina Silvano, Giovanni Agosta, Andrea Bartolini, Andrea R. Beccari, Luca Benini, Loïc Besnard, João Bispo, Radim Cmar, João Cardoso, Carlo Cavazzoni, Stefano Cherubin, Davide Gadioli, Martin Golasowski, Imane Lasri, Jan Martinovič, Gianluca Palermo, Pedro Pinto, Erven Rohou, Nico Sanna, Kateřina Slaninová, and Emanuele Vitali. 2018. ANTAREX: A DSL-based Approach to Adaptively Optimizing and Enforcing Extra-Functional Properties in High Performance Computing. In *21st Euromicro Conf. on Digital System Design (DSD)*. 600–607. https://doi.org/10.1109/DSD.2018.00105

[11] Cristina Silvano, Gianluca Palermo, Giovanni Agosta, Amir H. Ashouri, Davide Gadioli, Stefano Cherubin, Emanuele Vitali, Luca Benini, Andrea Bartolini, Daniele Cesarini, Joao Cardoso, Joao Bispo, Pedro Pinto, Riccardo Nobre, Erven Rohou, Loïc Besnard, Imane Lasri, Nico Sanna, Carlo Cavazzoni, Radim Cmar, Jan Martinovič, Kateřina Slaninová, Martin Golasowski, Andrea R. Beccari, and Candida Manelfi. 2018. Autotuning and Adaptivity in Energy Efficient HPC Systems: The ANTAREX Toolbox. In *Proceedings of the 15th ACM International Conference on Computing Frontiers (CF '18)*. 270–275. https://doi.org/10.1145/3203217.3205338

[12] Amir Yazdanbakhsh et al. 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