Fixed point Precision Tuning via Compiler Analyses & Transformations

Authors: Daniele Cattaneo, Stefano Cherubin, Michele Chiari, Giovanni Agosta
Affiliation: Politecnico di Milano, Italy

Abstract

Approximate computing is an emerging approach to achieve better energy/performance points by trading off the accuracy of results. Since many embedded systems are designed around low-cost, low-energy microcontrollers where no hardware support is provided for floating point arithmetics, an interesting approximate computing approach is achieved by converting floating point computations to fixed point. This approach, known as precision tuning, is tedious and error prone to perform manually. Yet, in a typical embedded design workflow, the need to keep computation error under control leads to performing this transformation manually.

The scientific literature contains several methodologies and tools for floating to fixed point conversion. However, few of these tools are integrated into production grade compiler frameworks, and most support only a limited set of input programming languages (typically only ANSI C).

In the last few years, our team developed expertise in effective code optimization through floating point to fixed point code optimization, as well as a tool to support this process, the ***T****uning* ***A****ssistant for* ***F****loating point to* ***F****ixed point* ***O****ptimization*.
We present a summary of the main problems that this process entails, and we discuss pro and cons of the possible solutions. In particular, we discuss:

- Identification of code regions that have to be processed
- Collection of instruction-level precision requirements
- Technology used to manipulate the code
- Validation methodology
- Evaluation of the type cast overhead
- Feedback estimation to understand the performance and error impact of the transformation

We designed an open-source solution that automatize the analyses and translation processes with a reduced impact on dependencies and programming models. Within this context, the programmer exploits standard `annotate` attributes on the source code to forward the annotations to the LLVM-IR without any change to the vanilla compiler front-end (`clang`). Annotations are later parsed by a dynamically loaded compiler pass, which actually performs the analysis and code transformation. As it acts on the LLVM-IR, this approach is completely source-language agnostic. The impact on the programmer toolchain is reduced by the use of a state-of-the-art compiler framework – LLVM – which is widely used in both academia and industry.
We applied this transformation to several use cases.