Towards efficient and effective fixed point support

Authors: Stefano Cherubin, Giovanni Agosta
Affiliation: Politecnico di Milano, Italy

Abstract

Precision tuning is a classic technique in embedded systems to trade-off computation accuracy for performance, which is gaining interest also in other segments of the computing continuum, due to the energy impact of floating point computations. Fixed-point conversion is commonly employed in embedded software development, but manual conversion of floating point code into its fixed point equivalent is a tedious, time-consuming and error-prone task. Several methodologies and tools to convert floating point code into fixed point equivalents have been proposed however, such tools usually carry the burden of complex frameworks that are designed to support specific architectures and programming languages – usually, ANSI C.
Production grade software often features either manually converted floating point code to fixed point equivalent code either software emulation of the IEEE754 floating point behavior.
We discuss three alternative approaches to support real values via fixed point computation, highlighting advantages and disadvantages of each one, and providing a comparison on a set of use cases from the embedded domain, as well as from high performance computing.
As first we analyze a lightweight solution which is based on a header-only C++ template class. We use an open source reference version which provides representations for signed and unsigned fixed point data type with parametric bit width. From the programmer point of view this approach is very close the manual conversion. The only additional dependency is a compiler which is compliant with the language standard C++11 however, it requires manual analysis to define the range of all the variables, the bit width of each integer and fractional part of the fixed point data type. In addition with respect to the manual conversion, it provides abstractions over the fixed point arithmetic operations and automatic data type conversion via static cast operations.
Then, we analyze a more complex solution which is based on a Source-to-Source (S2S) compiler framework. This solution partially automatize the process by running tools that analyze the source code via static analysis and dynamic profiling. The S2S compiler can automatically perform the changes to the source code to reflect the semantic of fixed point arithmetic across the whole code base – as long as the source code is available. Capabilities of this solution are limited to the ANSI C language and a limited subset of the C++ language. This ad-hoc S2S compiler framework carries several dependencies and a different programming model into the workflow of the programmer, which may conflict with the established toolchain of the usual development process.
Finally we discuss a self-contained solution which is based on the LLVM compiler framework. This solution provides an alternative approach to automatize the analysis and translation process 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.