Software Solutions for Floating Point to Fixed Point Source Code Conversion

S. Cherubin, G. Agosta

POLITECNICO MILANO 1863

HEAP LAB

{stefano.cherubin, giovanni.agosta}@polimi.it

An overview of the available solutions for floating point to fixed point code conversion. From the manual code conversion, through several levels of automatization, towards future directions and challenges.

MOTIVATION

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.

- Some HW units do not **support** floats
- Some HW units only implements **inefficient** floating point units
- **Vectorization** may be exploited more effectively with smaller data types

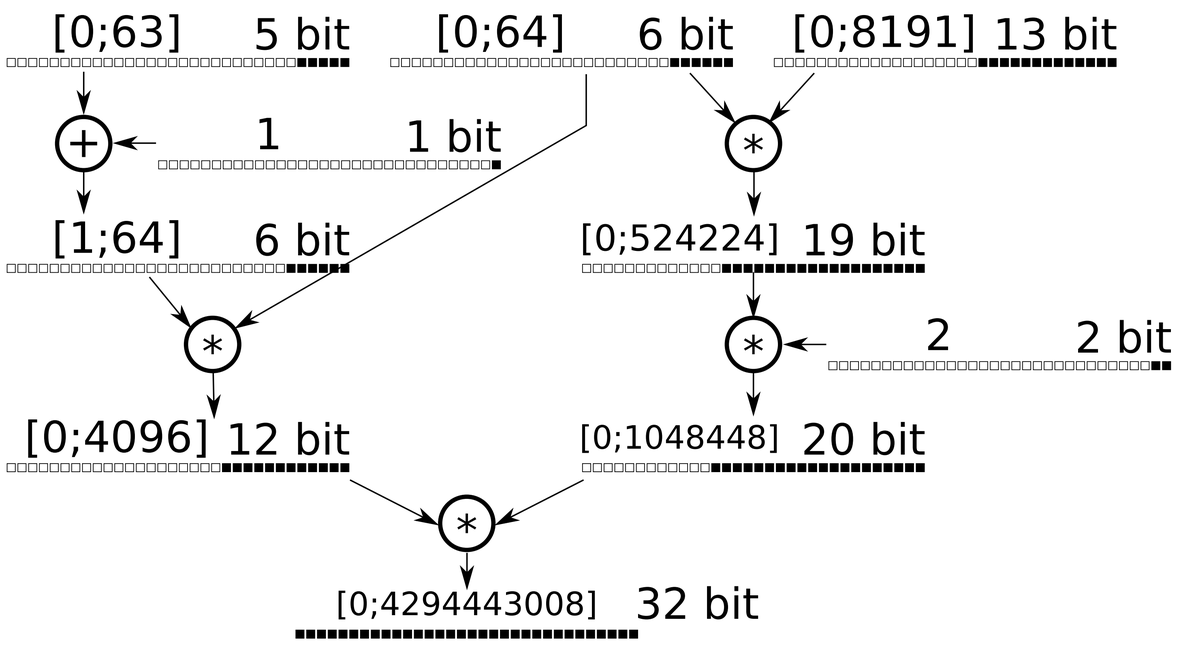
THE TRADITIONAL APPROACH

Programmers nowadays still perform manual conversion of floating point code into equivalent code which uses only integer arithmetic.
Although this approach requires a very high effort from the programmer, it allows to maximize the efficiency of the converted code.

Manual conversion requires the programmer to know the runtime bounds of each intermediate value in the algorithm in order to avoid any chance of overflow.
For large code bases this analysis becomes unfeasible by hand.

DATA FLOW ANALYSIS

Example of the data flow analysis to compute the minimum bit width of the values used by a real-time scheduling algorithm. Each node represents an arithmetic operation. For each node we report the range of values and the minimum data width required.



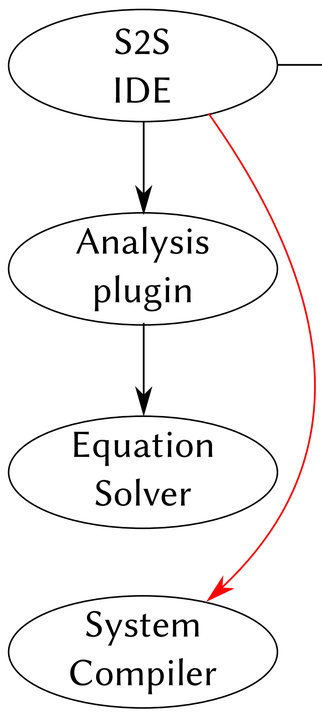
FIXED POINT C++ DATA TYPE

```
fixed_point_t<INT_BITS,FRAC_BITS> my_value;
```

Lightweight C++ header-only template class to represent a generic fixed point data type. From the programmer's point of view, this approach is very close to the manual conversion. In addition, it provides abstractions over the fixed point arithmetic operations and automatic data type conversion via static cast operations.

With respect to the traditional approach, this solution carries only an additional dependency, which is a compiler 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.

AD HOC S2S FRAMEWORKS

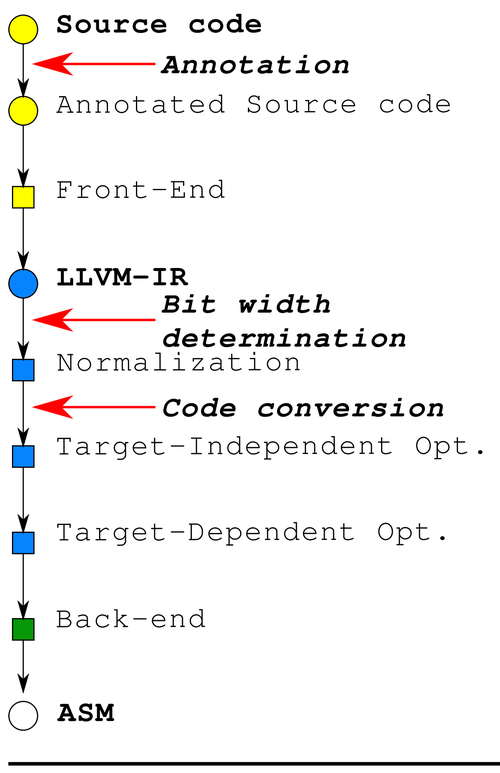


A common approach to automatizes the analysis and conversion process is to have a custom source-to-source framework. We take the example of the solution described in [1] where the GeCoS source-to-source compiler framework has been repurposed to determine the minimum data width to convert variables from floating point to fixed point representation.
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.

As can be seen from the above schema, 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.

SELF-CONTAINED LLVM PASS

A new approach described in [2] exploits the widely used LLVM compiler framework to perform the range analysis and data type conversion within the compiler.
By exploiting the clang plugin paradigm, this approach allows to limit the dependencies to the only standard LLVM front-end.



Our proposed transformation exploits only the compiler middle-end APIs which can be invoked through pluggable external modules called passes. The programmer **annotates the source** code via custom *annotate* attributes on floating point variables whose type they wish to convert to fixed point.
Attributes are parsed by the vanilla compiler front-end, which transfers them to the middle-end.
Our compiler pass **collects those annotations** and properly propagates them to intermediate values.
It **creates instructions** based on fixed point arithmetic equivalent to the original code.
We **compile** the converted code for the target architecture and we integrate the object code into the original build system.

ANNOTATION EXAMPLE

Variables and computations which are converted to fixed point.
Connections between them highlight the operations affected by the conversion.

It is possible to specify the size of the integer and fractional part of the representation.

```
float a __attribute((annotate("force_no_float")));
int b = 98;
a = b * 2.0;
a += 10.0;
float c __attribute((annotate("no_float 12 20")));
c = function1(b) + 3.5;
a += c * 2.0;
```

The `no_float` and `force_no_float` annotations indicate which variables have to be converted to fixed point. The conversion propagates to related computations in different ways: `force_no_float` entails the conversion of the dependencies, while `no_float` propagates only to intermediate values.

REFERENCES

[1] S. Cherubin. G. Agosta, I. Lasri. E. Rohou, O. Sentieys
Implications of Reduced-Precision Computations in HPC: Performance, Energy and Error
International Conference on Parallel Computing, Sep 2017

[2] D. Cattaneo, A. Di Bello, S. Cherubin, F. Terraneo, G. Agosta
Embedded Operating System Optimization through Floating to Fixed Point Compiler Transformation
Euromicro DSD 2018. Prague, Czech Republic. Aug 2018

ACKNOWLEDGEMENTS

This work is supported by the European Union's Horizon 2020 research and innovation programme, under grant agreement No 671623, FET-HPC ANTAREX.

ANTAREX 10^18