IWES 2018

Towards Efficient and Effective Fixed Point Support

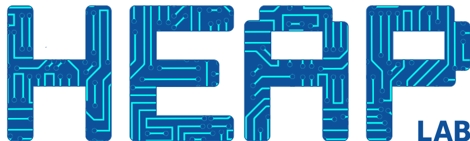
Stefano Cherubin, Giovanni Agosta
<name>.<surname>@{polimi.it}

Politecnico di Milano

14 September 2018

POLITECNICO
MILANO 1863

DIPARTIMENTO DI ELETTRONICA,
INFORMAZIONE E BIOINGEGNERIA



Section 1

1. Introduction
2. Manual Conversion
3. Abstract Data Type
4. Source-to-Source Compilers
5. Compiler Transformation
6. Conclusions

Motivation

Precision Tuning

Classic technique in Embedded Systems:

- Trade-off computation accuracy for performance/energy

Usually exploited among available floating point representations

32 bit	64 bit	128 bit

What if we want to use less bits?

- specialized hardware
- use integer data types
  - FIXED POINT representations

Motivation

Precision Tuning

Classic technique in Embedded Systems:

- Trade-off computation accuracy for performance/energy

Usually exploited among available floating point representations

32 bit	64 bit	128 bit

What if we want to use less bits?

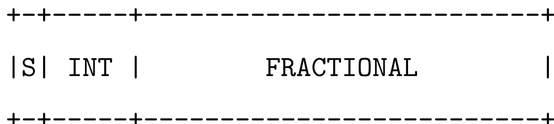
- specialized hardware
- use integer data types
  - FIXED POINT representations

Fixed Point Representations

Usually exploited when floating point unit is not available.

Programmable hardware can implement fixed point computation using custom width. We consider the general case of **x86-like** architecture.

- Keep fixed representation width (8, 16, 32, 64 bits)



Coding in Fixed Point

The *Embedded C* language allows programmers to use native fixed point data types. What about other programming languages? Let's take the example of ANSI C, C++ where the only way to represent reals is via floating point?

- Use native integer data types

HOW?

Section 2

1 Introduction

2 Manual Conversion

3 Abstract Data Type

4 Source-to-Source Compilers

5 Compiler Transformation

6 Conclusions

Manual Conversion

- Time consuming
- Error prone
- Unfeasible on large code base

Alternatives?

Several approaches have been proposed.
Let's discuss their benefits and drawbacks by examples.

Manual Conversion

- Time consuming
- Error prone
- Unfeasible on large code base

Alternatives?

Several approaches have been proposed.
Let's discuss their benefits and drawbacks by examples.

Section 3

1 Introduction

2 Manual Conversion

3 Abstract Data Type

4 Source-to-Source Compilers

5 Compiler Transformation

6 Conclusions

Template-based C++ class

Define a data type to automatize the most common operations

- ADT provide automatic scaling at every mul/div operation
- ADT provide conversion between representations
- ADT provide implicit static cast
- Programmer does the rest

ADT: Pro & Con

PRO

- Easy to use: just include an header file
- Portable

CON

- Force code standard change to C++
- Data format controlled by the programmer

Section 4

1 Introduction

2 Manual Conversion

3 Abstract Data Type

4 Source-to-Source Compilers

5 Compiler Transformation

6 Conclusions

S2S Approach

Change the source code to replace the floating point instruction with fixed point equivalents

- Programmer writes pragmas (or custom language)
- Tool performs pattern matching & rewrites code

- Requires custom environment

Example: ID.Fix & GeCoS

- Programmer annotates
- ID.Fix[1] propagates annotations
- GeCoS[2] source-to-source replaces floating point with fixed point
- Conversion utils are inserted before/after the given region

[1] `http://idfix.gforge.inria.fr`
[2] `http://gecos.gforge.inria.fr`

Annotations

- Programmer remarks variables that needs to be converted
- ID.Fix analysis returns dynamic range for annotated variable(s)

```
#pragma VARIABLE_TRACKING variable
for (int i = 0, i < 10, i++)
{
  variable = i;
}
```

```
Output:
variable_min = 0
variable_max = 9
```

Annotations

- Programmer remarks variables that needs to be converted
- ID.Fix analysis returns dynamic range for annotated variable(s)

```
#pragma VARIABLE_TRACKING variable
for (int i = 0, i < 10, i++)
{
  variable = i;
}
```

```
Output:
variable_min = 0
variable_max = 9
```

ID.Fix plugin

- Dynamic value range is propagated to all intermediate values
- For each variable we compute the minimun number of integer bits we need to represent it
- We leave the rest of the data size for the fractional part

Example:

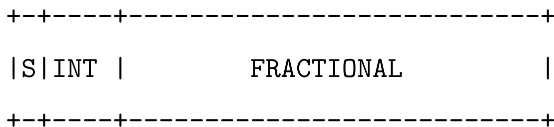
$$variable \in [0;9] \implies \begin{cases} INT_{variable} & \geq 4 \\ FRAC_{variable} & = 32 - INT_{variable} \end{cases}$$

ID.Fix plugin

- Dynamic value range is propagated to all intermediate values
- For each variable we compute the minimun number of integer bits we need to represent it
- We leave the rest of the data size for the fractional part

Example:

$$variable \in [0;9] \implies \begin{cases} INT_{variable} & \geq 4 \\ FRAC_{variable} & = 32 - INT_{variable} \end{cases}$$



Source-To-Source

The S2S compiler GeCoS takes the output of ID.Fix and converts the floating point code to fixed point code
Uses a C++ template-based fixed point library

```
double m[SIZE1][SIZE2];
FixedPoint<3,29> m_fixp[SIZE1][SIZE2];
convert2DtoFixP<double, SIZE1, SIZE2>(m, m_fixp);
```

GeCoS Infrastructure

ID.Fix is a plugin of the GeCoS source-to-source compiler

- GeCoS is a plugin of the Eclipse IDE
  - Requires to work with GUI
    - Requires JVM
- Requires an equation solver

Then, the code can be compiled using the system compiler.

S2S: Pro & Con

PRO

- Dynamically select the data types
- Fully automated

CON

- Source-language dependent
- Data format decision based on selected input test set
- Huge dependencies requirements
- Difficult to maintain

Section 5

1 Introduction

2 Manual Conversion

3 Abstract Data Type

4 Source-to-Source Compilers

5 Compiler Transformation

6 Conclusions

Compiler Transformation Approach

Perform analysis and code conversion within the compiler

- Provides the same features of the S2S compiler
- Static code analysis instead of dynamic profiling

Compiler-based Tuning Assistant

- Programmer specifies ranges of input values
- Compiler propagates ranges to intermediate values
- Compiler statically analyze the code
- Compiler automatically selects the best data type
- Compiler performs code conversion
- Compiler provides optimized code

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;
```

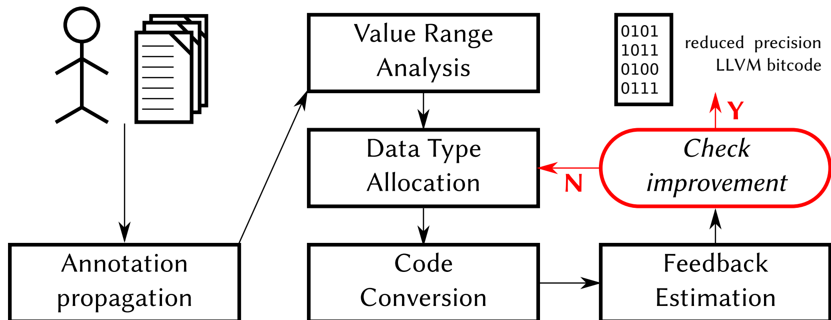
The no_float and force_no_float annotations indicate which variables have to be converted to fixed point. The conversion propagates to related computations in different ways: force_no_float entails the conversion of the dependencies, while no_float propagates only to intermediate values.

Example: Our Tool Infrastructure

Based on the LLVM compiler toolchain

- Annotations are natively supported by *clang*
  - Applies to a large variety of programming languages
- Packaged as self-contained clang plug-in
- *clang* can be used instead of the system compiler
  - or can be easily integrated with the target toolchain

Component Schema

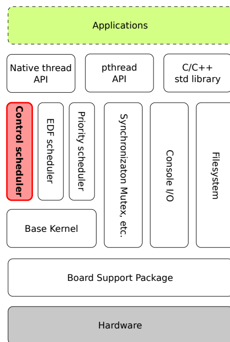


Benefits

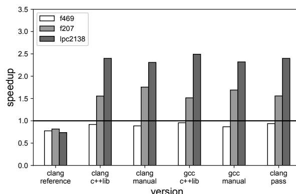
- Relies only on the well-known LLVM compiler framework
- Does not require any customization of the compiler
- Source language agnostic
- Easy to maintain

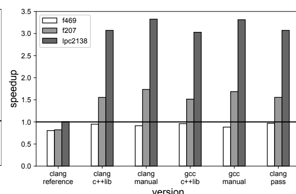
Success Story

We applied our tool to the Miosix embedded operating system



- Convert the Control-Theoretical scheduler from floating point to fixed point.
- Speedup achieved on real-time benchmarks (scheduling time)
- Work presented in Euromicro DSD 2018





Section 6

1 Introduction

2 Manual Conversion

3 Abstract Data Type

4 Source-to-Source Compilers

5 Compiler Transformation

6 Conclusions

Conclusions

- We presented different approaches to perform **reduced precision** computation with different **automation** levels
  - Exploit the **domain knowledge** of the programmer to perform **selective conversion** in a given source code
- We showed how to exploit **fixed point** for reduced precision computation in x86-like architectures

Conclusions

- We presented different approaches to perform **reduced precision** computation with different **automation** levels
  - Exploit the **domain knowledge** of the programmer to perform **selective conversion** in a given source code
- We showed how to exploit **fixed point** for reduced precision computation in x86-like architectures

Questions?

?

End

Thanks for your attention!