Code Transformation and Optimization Project, A. Y. 2013-2014

Marcello Pogliani
marcello.pogliani@mail.polimi.it

February 13, 2015

The project developed for the Code Transformation and Optimization course is a set of LLVM intraprocedural passes aimed at converting C functions from a floating point to a fixed point representation. The target fixed-poin representation is two's complement 64-bit fixed point, and the number of bits for the integer and decimal part is chosen by the algorithm and may be different for each function of the same compilation unit. The passes do not modify the return type of the function (the return value is converted back to floating point). Figure 1 reports the overall schema of the project.

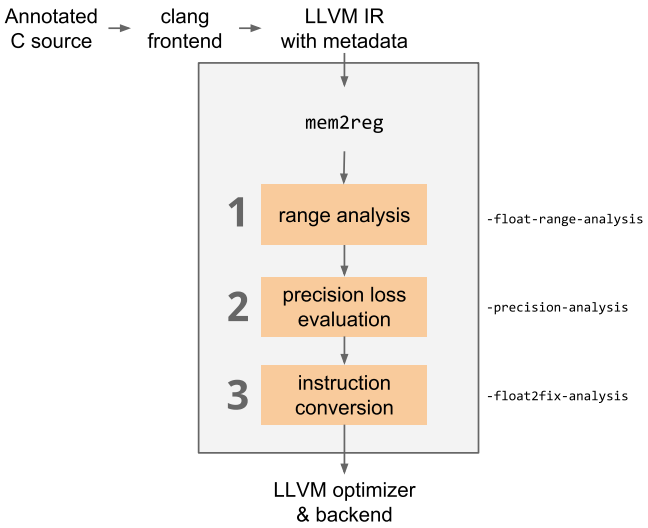

Figure 1: *Project structure*

1 Design notes

C source annotations To enable the range analysis, the value range of the function inputs should be known at compile time. For this purpose, the C functions to be converted must contain an annotation specifying the range of their parameters. The initial ranges are expressed as a pair of integers (unsigned longs). As an example, this snippet specifies that the variable *param* can range from -10 to $+10$:

```
double f(double param
         __attribute__((float_range(-10, 10))))
{
    /* body ... */
}
```

The attribute is expressed in the LLVM Intermediate Representation (IR) as a call to the intrinsic `llvm.float.range`. This intrinsic is not converted to any machine code, but used by the analysis passes to retrieve the range information.

Range analysis The `float-range-analysis` pass retrieves the initial range information from `llvm.float.range` intrinsics and forward-propagates it across the function. For each supported floating point operation, it propagates the range of the operands to the range of the result value according to the following rules:

$$
\begin{aligned}
&[a,b] + [c,d] = [a+b, c+d] \\
&[a,b] - [c,d] = [\min\{a-c, b-d\}, \max\{a-c, b-d\}] \\
&[a,b] \cdot [c,d] = [\min\{ac, bc, ad, bc\}, \max\{ac, bc, ad, bc\}] \\
&\frac{[a,b]}{[c,d]} = \left[\min\left\{\frac{a}{c}, \frac{b}{c}, \frac{a}{d}, \frac{b}{d}\right\}, \max\left\{\frac{a}{c}, \frac{b}{c}, \frac{a}{d}, \frac{b}{d}\right\}\right] \\
&\phi([a_1,b_1], \ldots, [a_n,b_n]) = [\min\{a_1, \ldots, a_n\}, \max\{b_1, \ldots, b_n\}]
\end{aligned}
$$

Every propagation operation with an operand having unbounded (`Range::Top`) range yields an unbounded result.

For a value having range $[x, y]$, the minimum number of bits required to store the the integer part without overflow is

$$\lceil \log_2 \max(\tilde{x}, \tilde{y}) \rceil + 1$$

where $\tilde{x} = x + 1$ if $x \geq 0$ and $\tilde{x} = |x|$ if $x < 0$. Globally, the minimum bit-width IBW required to store the integer part of each value is the maximum over the minimum integer bit-width of each value.

Precision analysis The `precision-analysis` pass builds on the results of the float range analysis to estimate the loss of precision due to the conversion to a fixed point representation.

Let IBW be the (minimum) number of bits for the integer part that avoids overflow (computed by the previous pass); for our target representation the number of bits DBW for the decimal part are computed as:

$$DBW = \lfloor (64 - IBW)/2 \rfloor$$

The most significant DBW bits are unused – this avoids to lose the most significant bits of the operands during some operations (multiplication and division).

The quantization (truncation) error due to the conversion from a floating point representation to a fixed point one is estimated as 2^{-DBW}. The pass computes the loss of precision $err(\cdot)$ according to the following rules:

$$
\begin{aligned}
err(a+b) &= err(a) + err(b) \\
err(a-b) &= err(a) + err(b) \\
err(a \cdot b) &= \max(|a|) \cdot err(b) + \max(|b|) \cdot err(a) + \\
&\quad + err(a) \cdot err(b) + 2^{-DBW} \\
err\left(\frac{a}{b}\right) &= \frac{\max(|a|)}{\max(b)^2} \cdot err(b) + \frac{1}{\max(|b|)} \cdot err(a) + 2^{-DBW}
\end{aligned}
$$

where $\max(\cdot)$ is the maximum absolute value of the value, computed from its range.

As a measure of the precision of the fixed point conversion, the pass uses the "equivalent" decimal bitwidths, i.e., the number of decimal bits so that the error occurring in the fixed point version of the functions is not greater than the quantization error when converting the floating point result to a fixed point representation using that decimal bit-width.

Conversion The conversion stage is implemented by the `float2fix` pass.

The pass supports the case where only some of the floating point instructions in a given function should be transformed to fixed point.

For each instruction that can be converted (and for its operands), the pass generates code to convert the floating point value into a fixed point one, and inserts those instructions immediately after the *definition* of the value. Pointers to converted instructions are inserted into a data structure for caching purposes. Then, the pass loops again on all the instructions, and checks whether the parameters of any non-converted instructions have been converted to fixed point. If this is the case, the pass creates a floating point version after the definition of the fixed point version.

To convert a value v to fixed point, the pass inserts a `fmul` instruction to compute $v \cdot 2^{DBW}$, followed by a cast to a 64 bit integer; the reverse happens to convert the value back to floating point.

Operations are implemented as the corresponding integer operations on signed 64-bit integers; in the case of multiplication, the operation is followed by a right shift (with sign extension) of DBW bits; a left shift of DBW bits is inserted before the divisions to avoid losing the decimal part during the operation.

The `float2fix` pass does not remove the original floating point instructions: the user is expected to run dead code elimination (`-dce`) afterwards.

Command line arguments The pass recognizes the following arguments:

- `precision-bitwidth` threshold to enable the conversion of a function, in terms of the minimum number of equivalent decimal bit-width as computed by the `precision-analysis` pass;
- `internal-bitwidth` overrides the result of the precision analysis pass and converts to fixed point all the instructions that are guaranteed not to overflow using the number of bits for the decimal part specified at command line, regardless of the precision loss.

Limitations

- Both the range analysis and the precision analysis perform propagation only in the case of simple loops where the trip-count can be computed statically at compile-time; furthermore, propagation occurs applying the transfer functions for the (statically known) number of loop iterations, thus in case of a large trip count the passes may be fairly inefficient; for loops with unknown trip-counts, an unbounded range is considered.
- During the range analysis, control dependencies are partly considered. When propagating the value range to an instruction i, the pass checks whether i is control dependent with respect to some condition concerning the operands of i. If this is the case, the ranges of the operands used for the propagation to the result of i are constrained according to the condition, in a way similar to the idea of *range refinement function* propsed in [6]. Despite this, the pass fails at considering some control-dependencies, especially in the *else* branch of *if* statements: complex conditionals (e.g., *if*$(x < 10.0$ *&&* $y > 4.0)$) are translated into the IR as multiple branches, thus the basic block for the *else* branch can have multiple predecessors, making it difficult for the algorithm to recognize the constraints.
- The precision analysis step may benefit from the use of affine arithmetic to propagate errors, as proposed in [5, 4].

2 How to compile the source code

The project requires to patch LLVM and clang with the implementation of the `llvm.float.range` intrinsics.

The project was developed and tested with LLVM 3.4 under Linux (in particular, it is known to work with CentOS 6.x using clang 3.4 compiled from sources using the stock gcc 4.4.x from the repositories).

The following instructions assume that

- `LLVM_SRC` is the directory where the llvm source code is located
- `LLVM_BUILD` is the llvm build directory (separated from the source tree)

Download LLVM 3.4 and clang:

```
$ wget http://llvm.org/releases/3.4/llvm-3.4.src.tar.gz
$ tar xvf llvm-3.4.src.tar.gz
$ mv llvm-3.4 $LLVM_SRC
$ rm llvm-3.4.src.tar.gz
$ cd $LLVM_SRC
$ wget http://llvm.org/releases/3.4/clang-3.4.src.tar.gz
$ tar xvf clang-3.4.src.tar.gz
$ mv clang-3.4 tools/clang
$ rm clang-3.4.src.tar.gz
```

Move the directory with the project source code to `projects/`:

```
$ mv /where/float-range/is/ $LLVM_SRC/projects/float-range
```

Apply the provided patch:

```
$ cd $LLVM_SRC
$ patch -p1 < projects/float-range/llvm-float-range.patch
$ cd tools/clang
$ patch -p1 < projects/float-range/clang-float-range.patch
```

Configure LLVM, clang and the project in `LLVM_BUILD`:

```
$ mkdir -p $LLVM_BUILD
$ cd $LLVM_BUILD
$ $LLVM_SRC/configure
$ mkdir -p $LLVM_BUILD/projects/float-range
$ cd $LLVM_BUILD/projects/float-range
$ $LLVM_SRC/projects/float-range/configure
```

Finally, configure and compile it all:

```
$ cd $LLVM_BUILD
$ make
```

3 Usage example

```
$ ./clang -emit-llvm file.c -c -o in.bc
$ ./opt -load /path/to/LLVMFloatRange.so -mem2reg -lcssa
        -float-range-analysis -precision-analysis -float2fix
        -dce -precision-bitwidth $PREC -S < in.bc > out.ll
```

References

[1] Clang – internals manual. `http://clang.llvm.org/docs/InternalsManual.html`.

[2] LLVM – how to write a pass. `http://llvm.org/docs/WritingAnLLVMPass.html`.

[3] LLVM programmer's manual. `http://llvm.org/docs/ProgrammersManual.html`.

[4] C.F. Fang, R.A. Rutenbar, and Tsuhan Chen. Fast, accurate static analysis for fixed-point finite-precision effects in dsp designs. In *Computer Aided Design, 2003. ICCAD-2003. International Conference on*, pages 275–282, Nov 2003.

[5] D. U. Lee, A. A. Gaffar, R. C.C. Cheung, O. Mencer, W. Luk, and G. A. Constantinides. Accuracy-guaranteed bit-width optimization. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 25(10):1990–2000, October 2006.

[6] M. Stephenson, J. Babb, and S. Amarasinghe. Bidwidth analysis with application to silicon compilation. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 108–120, New York, NY, USA, 2000. ACM.