Connecting LLVM with a WCET tool

Rick Veens
rickveens92@gmail.com

FOSDEM 2018 (4-2-2018)
10:35 – 11:05
“Connecting LLVM with a WCET tool”
talk outline

- What is WCET? WCET analysis?
- Why connect LLVM with a WCET tool?
- What tool to pick? (SWEET)
- Approach to connecting SWEET and LLVM
- WCET for the ARM Cortex-M3
- Conclusions
What is WCET?
What is WCET?

• Worst Case Execution Time: longest path in the code

• Example: inflation of an air bag

(by courtesy of Damir Isovic)
How to determine WCET?

- Running code, measuring?

- **No** There are too many paths in non-trivial code
  
e.g. `foo(unsigned a, unsigned b);`
  
  $2^{32} \times 2^{32}$ paths.

- **Static analysis** is the answer. Make use of
  *abstract interpretation*
WCET using static-analysis

![Diagram showing the relationship between probability, execution time, and WCET estimates.](image)

- **Unsafe execution time measurements**
- **Exact worst-case execution time**
- **Safe upper bound for worst-case execution time (aiT)**
- **WCET Estimates**
Why connect LLVM with a WCET tool?
Why connect LLVM with a WCET tool?

1) Provide WCET analysis alongside compilation
2) Re-use information about architecture (i.e. TableGen)

• Why not add WCET analysis to LLVM, instead of tool?
  - WCET analysis is not easy (Abstract Interpretation)
  - Why do the same work again?
  - To test if LLVM has enough info in TableGen.
What tool to pick?
(SWEET)
What tool to pick? (SWEET)

- **SWEdish Execution Time**
- Open-source tool
- Has interface language ALF
- Other tools considered: Bound-T, OTAWA, Hepatane, Absint aiT
How to use SWEET

- SWEET requires *cycle model* and *semantics*
  - Cycle Model: How many cycles is this basic-block?
  - Semantics: Register changes, control flow etc

![Diagram showing the process of using SWEET](image-url)
Example of ALF code

Example of addition-instruction of two registers (R0, R1)
Approach to connecting SWEET and LLVM
Approach to connecting SWEET and LLVM

- Output ALF from LLVM
- What ALF? Use info from TableGen
- Output from MI, just before conversion to MC. (addPreEmitPass of TargetPassConfig)
How to determine ALF per instruction

- **TableGen** back-end that generates ALF based on DAG-pattern
  - Generate function that determines ALF code for given `MachineInstr` object.
- **Most** instructions have a DAG-pattern
- Condition flags are assumed to be N, Z, C, V

```
llc (backend)
  MachineFunctionPass
  ALFWriter
  ARMALFWriter
  ...ALFWriter

llvm-tblgen
  .td file
  ALFWriterEmitter

.lc file
  .inc file
```
WCET for the ARM Cortex-M3
Generating for ARM Cortex-M3

- 52 of 86 Thumb1 covered (14 custom)
- Simple cycle model
- QEMU simulator used to test
- Discovered issues:
  1) Not all code available in LLVM: libgcc
  2) Globals are allocated by the linker, addr not known by LLVM
Conclusions
Conclusions

- Goal: Add WCET analysis with the SWEET tool
- SWEET runs on ALF code
- Using TableGen to generate ALF
  - Generated ALF for ARM Cortex-M3 for some programs
  - Condition flags not in TableGen
  - LLVM does not have all information (libgcc, globals)
Future work

- TableGen back-end uses fixed DAG-patterns e.g. (set .. (add … … ) )
- Instruction delay-slots not considered
- ARM Cortex-M3 instructions only partially finished
- Hand-write libgcc functions in ALF
Some links

• Code of LLVM with WCET additions
  https://github.com/rveens/LLVM-WCET-SWEET

• Vim syntax highlighting for ALF
  https://github.com/rveens/alf-vim

• SWEET homepage
  http://www.mrtc.mdh.se/projects/wcet/sweet/

• ALF spec
end of presentation
How to determine ALF per instruction

- Considered patterns:
  
  - Note: operators such as add, sub, ld etc are defined in TargetSelectionDAG.td

<table>
<thead>
<tr>
<th>DAG-pattern</th>
<th>Flags set</th>
</tr>
</thead>
<tbody>
<tr>
<td>(set $R (imm $L) )</td>
<td>N, Z</td>
</tr>
<tr>
<td>(set $R (add $L, $L) )</td>
<td>N, Z, C, V</td>
</tr>
<tr>
<td>(set $R (adde $L, $L) )</td>
<td>N, Z, C, V</td>
</tr>
<tr>
<td>(set $R (sub $L, $L) )</td>
<td>N, Z, C, V</td>
</tr>
<tr>
<td>(set $R (ld $L) )</td>
<td>N, Z</td>
</tr>
<tr>
<td>(set $R (shl $L $L) )</td>
<td>N, Z, V</td>
</tr>
<tr>
<td>(set $R (srl $L $L) )</td>
<td>N, Z, V</td>
</tr>
<tr>
<td>(set $R (mul $L $L) )</td>
<td>N, Z, V</td>
</tr>
<tr>
<td>(set $R (xor $L $L) )</td>
<td>N, Z, V</td>
</tr>
<tr>
<td>(set $R (or $L $L) )</td>
<td>N, Z, V</td>
</tr>
<tr>
<td>(set $R (and $L $L) )</td>
<td>N, Z, V</td>
</tr>
<tr>
<td>(set $R (sext_inreg $L $Type) )</td>
<td>N, Z</td>
</tr>
<tr>
<td>(st $L $L )</td>
<td>-</td>
</tr>
<tr>
<td>(br $BB )</td>
<td>-</td>
</tr>
</tbody>
</table>
LLVM TableGen example

```c
// A8.6.2
def tADC:
    T1sItDPEncode<0b0101, (outs tGPR:$Rdn), (ins tGPR:$Rn, tGPR:$Rm), IIC_iALUr,
            "adc", "\t$Rdn, $Rm",
            [(set tGPR:$Rdn, (adde tGPR:$Rn, tGPR:$Rm))],
            Sched<[WriteALU]>;

// Instruction InstTemplate InstThumb
def tADC {
    ….T1sItDPEncode Sched
    field bits<32> Inst = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
                           0, 0, 0, 0, 1, 0, 1, ... };  
    dag OutOperandList = (outs tGPR:$Rdn, s_cc_out:$s);
    dag InOperandList = (ins tGPR:$Rn, tGPR:$Rm, pred:$p);
    string AsmString = "adc${s}${p} $Rdn, $Rm";
    list<dag> Pattern = [(set tGPR:$Rdn, (adde tGPR:$Rn, tGPR:$Rm))];
    list<Register> Uses = [CPSR];
    list<Register> Defs = [];
```
SWEET Abstract Execution

(a) Example

```
i = INPUT; // i = [1..4]
while (i < 10) {
    // point p
    ...
    i = i + 2;
}
// point q
```

(b) Analysis

<table>
<thead>
<tr>
<th>iter</th>
<th>i at p</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[1..4]</td>
</tr>
<tr>
<td>2</td>
<td>[3..6]</td>
</tr>
<tr>
<td>3</td>
<td>[5..8]</td>
</tr>
<tr>
<td>4</td>
<td>[7..9]</td>
</tr>
<tr>
<td>5</td>
<td>[9..9]</td>
</tr>
<tr>
<td>6</td>
<td>impossible</td>
</tr>
</tbody>
</table>

(c) Result

min. #iter: 3
max. #iter: 5

Figure 1. Example of abstract execution