Automatic Binary Translator Generation from Instruction Set Description

Aleksandr Bezzubikov, Nikita Belov, Kirill Batuzov

ISP RAS

1 December 2017
Typical emulator scheme
Emulator Frontend

Decoder

```c
while(...) {
    opc0 = read()
    switch(opc0) {
    case0: {
       opc1 = read()
       switch(opc1) {
       ...
       }
    }
    case1:
    ...
    }
}
```

Translator to the IR

```c
void gen_mov_const(val) {
    TCGv tmp = tcg_temp_new_i32();
    tcg_gen_movi_i32(tmp, val);
    tcg_temp_free(tmp);
}
```
Motivation

• Writing a decoder
  • Time-consuming as performed manually
  • Lots of boilerplate code (nested switches)
  • Lots of auxiliary code
    Can be generated automatically

• Translating to the IR
  • Existing QEMU IR is too assembly-like to be convenient
  • That’s why API is inconvenient too
Related work

Common properties:
• High-level language for instruction specification
• Decoder and translator generation steps are splitted

Existing languages:
• nML – doesn’t support loops
• ISDL – describes instruction set in grammars, no working translator generator exists
• LISA – too detailed, describes CPU execution process
System overview

- Decoder and auxiliary code generator
- Instructions encoding specification
- Semantics language translator
- Instructions semantics specification
- Emulator sources
Decoder generation

Instruction set specification -> Parse tree generator -> Target architecture instruction parse tree -> C code generator -> Decoder in C
Target architecture specification

• In Python

• Instruction specification:
  • List of Instruction objects
  • Each Instruction is a list of fields:
    • Opcode(length, value)
    • Operand(length, name, blockNum=0)
    • Reserved(length, value=0)
  • Supports nested alternatives
    (different instructions with the same prefix)

```python
instruction_list = [
    Instruction('AND',
        Opcode(8, 0b00100110),
        Operand(4, 'A'),
        Operand(4, 'B')),
    Instruction('ADD',
        Opcode(8, 0b00000101),
        Operand(4, 'A'),
        Operand(4, 'B')),
    Instruction('BRK',
        Opcode(8, 0b00110101),
        Reserved(8))
]
```
Target architecture specification

• In Python

• CPU specification:
  1) Attributes
  2) Registers
  3) CPU state fields

```python
target_cpu = TargetCPU(
    'Moxie',
    Attribute('TARGET_LONG_BITS', 32),
    Attribute('TARGET_PAGE_BITS', 12),
    Attribute('NB_MMU_MODES', 1),
    StateField('uint32_t', 'flags'),
    StateField('uint32_t', 'pc'),
    RegisterGroup('gregs', regs=
      RegisterRange('gr', elem_size=32, end=16).regs),
    Register('cc_a', 32),
    Register('cc_b', 32)
  )
```
Generating parse tree

• Instruction encoding: a string of ‘0’, ‘1’ and ‘x’
• Recursive algorithm over a set of instruction encodings:
  • step 0: set contains all the instructions
  • step k:
    • size(current_set) = 1
      1) save a reference to the instruction
      2) return
    • size(current_set) > 1:
      1) find fragment of length $L$ starting from position $a$ consisting only of ‘0’ and ‘1’
      2) split current set by different values of the substring found
      3) for each subset perform a recursive call
Parse tree example
Semantics description language

• Convenient syntax
  • Based on C11

• DBT-specific
  • Added some QEMU-specific operators: signed/unsigned extension, signed division/remainder, cyclic shift and signed shift
  • Runtime values distinction
    1) Compile-time constants
    2) Resulting translator runtime values
    3) Guest code runtime values – HLTTemp
Language translator overview

Diagram:

- Instruction semantics description
- Preprocessor
- Translator
- Emulator sources
Semantics language translator: preprocessing

Transform into C11-compliant form:
• Define HLTTemp type
• Replace non-standard operators with API function calls
Semantics language translator: translation

- Parse input into an AST using clang
- Traverse AST and transform it with clang::Rewriter
- Transformation:
  - Statements independent of HLTTemp are copied ‘as-is’
  - Statement which depend on HLTTemp are translated into SSA form that is then used as translator IR
  - Translator IR operations are mapped to QEMU API calls, translator simply outputs these calls
  - For each HLTTemp variable declaration its allocation and deallocation are generated
Semantics language translation example

extern HLTTemp gregs[16];
static void inc_0(DisasContext *ctx, int r, int imm)
{
    if(imm == 1) {
        gregs[r]++;
    } else {
        gregs[r] += imm;
    }
}

extern TCGv cpu_gregs[16];
static void inc_0(DisasContext *ctx, int r, int imm)
{
    if(imm == 1) {
        TCGv _tmp_1 = tcg_temp_new();
        tcg_gen_mov_tl(_tmp_1, cpu_gregs[r]);
        tcg_gen_addi_tl(cpu_gregs[r],
                         cpu_gregs[r], 1);
        tcg_temp_free(_tmp_1);
    } else {
        tcg_gen_addi_tl(cpu_gregs[r],
                        cpu_gregs[r], imm);
    }
}
Evaluation

• Moxie as a target architecture

• Decoder generator evaluation
  1) Generate binaries with random instruction sequences
  2) Pass this binaries to generated decoder
  3) Compare generated decoder output to originally compiled instructions sequence

• Semantics translator evaluation
  • Synthetic tests

• Complex Moxie architecture test – compare CPU state after each instruction execution in generated translator and existing one
## Evaluation

<table>
<thead>
<tr>
<th>Component</th>
<th>QEMU API</th>
<th>Proposed language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auxiliary code</td>
<td>435 lines, 14656 characters</td>
<td>0 lines, 0 characters</td>
</tr>
<tr>
<td>CPU definition</td>
<td>37 lines, 1371 characters</td>
<td>16 lines, 639 characters</td>
</tr>
<tr>
<td>Instructions</td>
<td>53 lines, 1520 characters</td>
<td>21 lines, 430 characters</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>TCG API</th>
<th>Proposed language</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>1 line, 40 characters</td>
<td>1 line, 10 characters</td>
</tr>
<tr>
<td>BEQ</td>
<td>8 lines, 568 characters</td>
<td>6 lines, 194 characters</td>
</tr>
<tr>
<td>CMP</td>
<td>2 lines, 93 characters</td>
<td>2 lines, 26 characters</td>
</tr>
<tr>
<td>MOV</td>
<td>1 line, 53 characters</td>
<td>1 line, 27 characters</td>
</tr>
<tr>
<td>PUSH</td>
<td>5 lines, 242 characters</td>
<td>4 lines, 125 characters</td>
</tr>
<tr>
<td>UDIV</td>
<td>2 lines, 116 characters</td>
<td>2 lines, 52 characters</td>
</tr>
<tr>
<td>XOR</td>
<td>1 line, 40 characters</td>
<td>1 line, 10 characters</td>
</tr>
</tbody>
</table>
Conclusion

• We proposed the new system for automatic dynamic binary translator generation
• The system allows developers to use high-level language for instruction encoding and semantics description
• The system generates correct dynamic binary translator for described target architecture
• The system was tested on Moxie architecture
Thanks!