Automatic Generation of Assembly to IR Translators Using Compilers

Niranjan Hasabnis and R. Sekar

Stony Brook University, NY

8th Workshop on Architectural and Microarchitectural Support for Binary Translation (AMAS-BT)
7 February, 2015

Introduction

- CPU emulators, VMs

  ![bochs](bochs.png)

  ![EMU](emu.png)

- Binary translation, analysis, instrumentation systems,

Limitation

Rely on **manual development**

![Analysis and Transformation](diagram.png)

Assembly-to-IR translators are built manually.

Solution


---

1v3.10.0, 2014
Our approach

- Use modern compilers to build assembly-to-IR translators
  - Code generator = IR-to-assembly translator
  - Support many architectures
  - Support most (if not all) target instructions (GCC supports AVX, FMA4, SSE4.1.)

Our approach

Use code generator to build assembly-to-IR translator.

Approach details

Steps in our approach: (1) extract IR-to-assembly rules

<table>
<thead>
<tr>
<th>RTL instruction</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{set (mem:SI (pre_dec:SI (reg:SI esp))) (reg:SI ebp))})</td>
<td>(\text{pushl } %ebp})</td>
</tr>
<tr>
<td>(\text{set (reg:SI ebp) (reg:SI esp))}) (parallel (\text{set (reg:SI esp)}))</td>
<td>(\text{movl } %esp,%ebp})</td>
</tr>
<tr>
<td>(\text{set (reg:SI esp) (const \text{int } -20))})</td>
<td>(\text{subl } 20,%esp})</td>
</tr>
<tr>
<td>(\text{clobber (reg:CC eflags))})</td>
<td></td>
</tr>
</tbody>
</table>

OpenSSL

GCC

Compilation Logs

Approach details

Steps in our approach: (2) parameterize numeric values

1. **Identify** numeric parameters in IR \((P_i)\) and assembly \((P_a)\)
2. **Map** \(P_a\) to \(P_i\) \((f: P_a \rightarrow P_i)\)
   - \(f\) considered:
     - \(P_i = P_a\)
     - \(P_i = P_a + C\)
     - \(P_i = P_a - C\)
     - \(P_i = P_a \times C\)
     - \(P_i = P_a \div C\) \((C\) is a constant.)

<table>
<thead>
<tr>
<th>IR</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{set (reg:SI ax)}) (plus (reg:SI ax) (const_int = X)))</td>
<td>(\text{add } X,%eax)</td>
</tr>
<tr>
<td>(\text{clobber (reg:FLAGS))})</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IR</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{set (reg:SI ax)}) (plus (reg:SI ax) (const_int = 2)))</td>
<td>(\text{add } 2,%eax)</td>
</tr>
<tr>
<td>(\text{clobber (reg:FLAGS))})</td>
<td></td>
</tr>
</tbody>
</table>

Problem

Too many possible operand combinations

Solution

Parameterize IR-to-assembly rules.
### Approach details

#### Parameterization examples

**Concrete assembly and IR**

```
sub sp, sp, #8
(set (reg:SI sp) (plus:SI
  (reg:SI sp)
  (const_int -8)))
```

```
cmpl $3, 12(%esp)
(set (reg:CGC eflags)
  (compare:CGC mem:SI
    (plus:SI (reg:SI esp)
      (const_int 12)))
  (const_int 3)))
```

```
movl $8, 8(%esp)
(set (mem:SI (plus:SI
  (reg:SI esp) (const_int 8)))
  (const_int 8))
```

**Parameterized assembly and IR**

```
sub sp, sp, #X
(set (reg:SI sp) (plus:SI
  (reg:SI sp)
  (const_int -1 × X & X - 16)))
```

```
cmpl $X, Y(%esp)
(set (reg:CGC eflags)
  (compare:CGC mem:SI
    (plus:SI (reg:SI esp)
      (const_int =Y & X × 4)))
  (const_int =X & Y + 4)))
```

```
movl $X, Y(%esp)
(set (mem:SI (plus:SI
  (reg:SI esp) (const_int =X & =Y)))
  (const_int =X & =Y))
```

### IR ($I$) and assembly ($A$): mapping possibilities

1. $A_1 \rightarrow I$ and $A_2 \rightarrow I$
   - xor %eax, %eax, mov $0, %eax
   - Not really a challenge

2. $A \rightarrow I_1$ and $A \rightarrow I_2$
   - Confusion for assembly-to-IR translator
   - $I_1$ and $I_2$ should be semantically-equivalent
   - No cases found in testing

3. List of $A \rightarrow I$
   - **Challenge:** map single or multiple elements?
   - Max list size of 4 elements

4. List of $I \rightarrow A$
   - Handled normally

### Implementation

**Rule extraction**

- GCC plugin (70 lines of C) for rule extraction
- Integrates with `make` and `configure`
  - **Completely architecture-neutral**

**Parameterization**

- 900 lines of C++ code - **architecture-neutral**
- 70 lines of architecture-specific code (to parse log files)

### Evaluation: test setup

- **Test setup**
  - Used GCC-4.6 code generator
  - Compilation logs of openssl and binutils
  - Produced assembly-to-IR translators for x86, ARM, and AVR
  - For comparison purpose, used exact recall

- **Test criteria**
  1. Statistics of training data and learned rules
  2. Completeness
  3. Support for multiple architectures
  4. Compiler independence
  5. Translating advanced instructions
  6. Correctness
Training data and learned rules

<table>
<thead>
<tr>
<th>Arch</th>
<th>Parameter</th>
<th>Packages used for compilation</th>
<th>x86</th>
<th>ARM</th>
<th>AVR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># of unique concrete rules</td>
<td>openssl, binutils, openssl+binutils</td>
<td>21800</td>
<td>32400</td>
<td>300</td>
</tr>
<tr>
<td></td>
<td># of parameterized rules</td>
<td>openssl, binutils, openssl+binutils</td>
<td>6700</td>
<td>7300</td>
<td>170</td>
</tr>
<tr>
<td></td>
<td># of unique mnemonics</td>
<td>openssl, binutils, openssl+binutils</td>
<td>100</td>
<td>87</td>
<td>23</td>
</tr>
</tbody>
</table>

Figure: Details of training data used for learning purpose

Completeness

- Cross-testing mode

Other architectures, compilers, advanced instructions

- Support for multiple architectures

<table>
<thead>
<tr>
<th>Arch</th>
<th>ARM</th>
<th>AVR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training data</td>
<td>openssl+binutils GCC-4.6 cross compiler</td>
<td>openssl+binutils GCC-4.6 cross-compiler</td>
</tr>
<tr>
<td>Code generator</td>
<td>91% of coreutils</td>
<td>92% of coreutils</td>
</tr>
<tr>
<td>Instructions translated to IR</td>
<td>4 hrs</td>
<td>3 hrs</td>
</tr>
<tr>
<td>Time to build asm-to-IR translator</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Compiler independence
  - Test data: LLVM-3.3 compiled x86 coreutils binaries
  - Train data: GCC-compiled binutils+openssl
  - Translated 91% of instructions
  - 9% not in training data

- Translating advanced instructions
  - Training data: scientific packages (gimp)
  - Data covered 97% of advanced x86 instructions

Correctness

1. Self-testing and cross-testing
   - Check if $\forall (I, A) \in D_{test}: \text{translation}(A) = I$.

2. Semantic equivalence test: future work
   - Check if $\forall (I, A) \in D_{test}: \text{semantics}(A) = \text{semantics(translation}(A))$.
   - Takes care of multiple assemblies mapping to same IR

3. Loop-back test
   - Feed translated assembly back to compiler
   - Check if it produces same assembly
Related work

- Manually-building assembly-to-IR translator
  - QEMU, Valgrind
  - UQBT
- Relying on existing assembly-to-IR translators
  - BAP uses Valgrind.
- Building assembly-to-IR translators using compilers
  - Dagger
    - LLVM-specific
    - LLVM as a whitebox
    - Porting to other compilers is labor-intensive.
  - QEMU + LLVM
    - QEMU backend to produce LLVM IR
    - Limitations of QEMU limits applicability

Future work

- Comprehensive completeness evaluation
  - **Coverage**: lot of training data available
  - Evaluation across compilers
  - What about registers as parameters?
  - Building binary translation/instrumentation systems

Conclusion

- **Novel** and **automatic** approach to build assembly-to-IR translators
- **Reduces manual development efforts** considerably
- Evaluation demonstrates
  - architecture-neutrality
  - compiler-neutrality
  - reduction in manual efforts

Thank you.. Question?

nhasabni@cs.stonybrook.edu
http://seclab.cs.stonybrook.edu