Handling Multi-Versioning in LLVM: Code Tracking and Cloning

Alexandra Jimborean  Vincent Loechner  Philippe Clauss

INRIA - CAMUS Team
Université de Strasbourg

London - September, 2011
Outline

1. Motivation
2. State of the Art
3. Tracking code in LLVM IR using attached metadata
4. Interaction between high- and low-level IRs
5. Experiments
6. Conclusions
Why do we need multi-versioning?

Multi-versioning

- Sampling – Instrumentation
- Adaptive computing – Runtime version selection
- Dynamic optimization – Speculative parallelism

Multiple versions in different representations

- Each version in the most suitable IR
- Low-level IR for acquiring low-level information
- Higher level IR for performing code transformations
- Handled by a runtime system
Why do we need multi-versioning?

Multi-versioning

- Sampling – Instrumentation
- Adaptive computing – Runtime version selection
- Dynamic optimization – Speculative parallelism

Multiple versions in different representations

- Each version in the most suitable IR
- Low-level IR for acquiring low-level information
- Higher level IR for performing code transformations
- Handled by a runtime system
Outline

1 Motivation

2 State of the Art

3 Tracking code in LLVM IR using attached metadata

4 Interaction between high- and low-level IRs

5 Experiments

6 Conclusions
Related work

Tracking code through the optimization phase

- Extend debugging info and create bi-directional maps [Brooks et al.]
- Debug dynamically optimized code [Kumar et al.]

Interactive Compilation Interface

- Providing access to the internal functionalities of the compilers
- Generic cloning, instrumentation, control of individual optimization passes
- Multi-versioning available only at function level

http://ctuning.org/ici
LLVM features

Embedding high-level information in the IR
- Support for preserving the high-level information
- Annotate the code using metadata
  - No influence on the optimization passes, unless designed for this

Cloning utilities
- Copies of instructions, basic blocks or functions
- No correlation between original and cloned values
- Reserved only for some very specific situations
Outline

1. Motivation
2. State of the Art
3. Tracking code in LLVM IR using attached metadata
4. Interaction between high- and low-level IRs
5. Experiments
6. Conclusions
Code tracking in C/C++ source code

- Source code: pragma
  - Define new pragma to delimit the code regions of interest

```c
#pragma multi-version
{
    for(int i=0; i<N; i++)
        a[i] = 2 * i;
}
```

- Focus on loop nests
Extending the IR vs using annotations

Barriers:

```
#pragma new pragma {
C/C++ code
C/C++ code
}
```

```
Statement_out_1
LLVM_dummy_inst1
Statement_in_1
Statement_in_2
LLVM_dummy_inst2
Statement_out_2
```

```
Statement_out_1
LLVM_dummy_inst1
Statement_in_1
Statement_in_2
LLVM_dummy_inst2
Statement_out_2
```

Metadata:

```
#pragma new pragma {
C/C++ code
C/C++ code
}
```

```
Statement_out_1
Statement_in_1,
!metadata_info !0
Statement_in_2,
!metadata_info !0
Statement_out_2
```

```
Statement_out_1
Statement_in_1,
!metadata_info !0
Statement_in_2,
!metadata_info !0
Statement_out_2
```
Identify the region after applying optimizations

- Loop nest structure is significantly changed
  - Loop fusion, splitting, interchange etc.
- Metadata information may not be preserved
- Identify instructions that carry metadata information and consider the whole enclosing loop nest
  - Additional code might be included
  - All instructions marked for multiversioning are enclosed
A. Cloning
B. Rebuild control-flow-graph between clones
C. Extract versions in separate functions

Each version compiled independently into the most suitable IR
Challenges: Dominate all uses

Instruction does not dominate all uses!

%tmp = add i32 %a, %b
%aux_clone = add i32 %c, %tmp

Clone, replace uses in clones, reinsert, reconstruct the loop structure

%tmp = add i32 %a, %b
%aux = add i32 %c, %tmp
%tmp_clone = add i32 %a_clone, %b_clone
%aux_clone = add i32 %c_clone, %tmp_clone
Outline

1. Motivation
2. State of the Art
3. Tracking code in LLVM IR using attached metadata
4. Interaction between high- and low-level IRs
5. Experiments
6. Conclusions
Interaction between high- and low-level IRs

- Communication between code versions in distinct representations
- Control flow cannot enter or exit lower level representations
  - Inline assembly is expected to ‘fall through’ to the following code
- Handle the control flow graph in the low-level IR
- Minimally influence the behavior of the original code
Interaction between high- and low-level IRs

Handling jumps between LLVM IR and inline assembly

- **Generic callbacks - patched by the runtime system**
  - `mov $0x0,%rdi //address of the module`
  - `mov $0x0,%rsi //address of the function`

- **Labels**
  - Identify the address of the code to be patched
  - Target of the inline jumps

- **Jumps**

<table>
<thead>
<tr>
<th>Macro</th>
<th>Hexadecimal form</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>asm_jge8 TARGET</code></td>
<td><code>.byte 0X7D</code></td>
</tr>
<tr>
<td></td>
<td><code>.byte \TARGET \()-.-1</code></td>
</tr>
<tr>
<td><code>asm_jge32 TARGET</code></td>
<td><code>.byte 0X0F, 0X8D</code></td>
</tr>
<tr>
<td></td>
<td><code>.long \TARGET \()-.-4</code></td>
</tr>
</tbody>
</table>
Control flow graph rewritten in inline code
Control flow graph rewritten in inline code

Interaction between high- and low-level IRs

A. Jimborean, V. Loechner, Ph. Clauss

Handling Multi-Versioning in LLVM: Code Tracking and Cloning

London - September, 2011
Control flow graph rewritten in inline code
Challenges: Phi nodes

- Promote registers to memory
- `opt -reg2mem prg.bc`
Interaction between high- and low-level IRs

Eliminate Phi nodes to hack into the CFG

A. Jimborean, V. Loechner, Ph. Clauss

Handling Multi-Versioning in LLVM: Code Tracking and Cloning

London - September, 2011
Interaction between high- and low-level IRs

**Toy example**
Interaction between high- and low-level IRs

SPEC CPU 2006 bzip

CFG of a simple loop from bzip2 SPEC CPU 2006
Promote registers to memory

Loop indices must be either defined or used outside the loop, otherwise they are not sent as parameters when extracting the loops in new functions.
Promote registers to memory

- *Inline assembly defining labels must come before the phi instructions*
Interaction between high- and low-level IRs

Promote registers to memory

- More memory accesses
- Restricted optimizations
- Negative impact on performance
Challenges: Inline assembly

- Prevent optimizations from duplicating, reordering, deleting the inlined code
  - Create a new BasicBlock containing only the asm code
  - Connect it in the CFG using indirect branches
  - Insert metadata to prevent optimizations
- Minimally influence the optimization passes to maintain performance
Outline

1. Motivation
2. State of the Art
3. Tracking code in LLVM IR using attached metadata
4. Interaction between high- and low-level IRs
5. Experiments
6. Conclusions
Loop Instrumentation by sampling

Decision block:

Original version of the loop

Instrumented version of the loop

End of the loop

Following code
Challenges: Multiple exit loops

- Extract each loop in a new function
- Unique exit: returning point of the function
Challenges: Instrumentation instructions

- In x86_64 assembly: after register allocation

- In LLVM IR
  - Requires type conversions
  - Instrumenting all LLVM loads and stores → negative impact on the performance

A. Jimborean, V. Loechner, Ph. Clauss

Handling Multi-Versioning in LLVM: Code Tracking and Cloning

London - September, 2011
## Results SPEC CPU 2006

<table>
<thead>
<tr>
<th>Program</th>
<th>Runtime overhead (-O0)</th>
<th>Runtime overhead (-O3)</th>
<th># linear m.a.</th>
<th># instrumented m.a.</th>
<th>Percentage of linear m.a.</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip2</td>
<td>0.24%</td>
<td>12.31%</td>
<td>608</td>
<td>1,053</td>
<td>57.73%</td>
</tr>
<tr>
<td>mcf</td>
<td>20.76%</td>
<td>17.23%</td>
<td>2,848,598</td>
<td>4,054,863</td>
<td>70.25%</td>
</tr>
<tr>
<td>milc</td>
<td>0.081%</td>
<td>3.61%</td>
<td>1,988,256,000</td>
<td>1,988,256,195</td>
<td>99.99%</td>
</tr>
<tr>
<td>hmmer</td>
<td>0.062%</td>
<td>0.76%</td>
<td>845</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>sjeng</td>
<td>182%</td>
<td>11.13%</td>
<td>1,032,148,267</td>
<td>1,155,459,440</td>
<td>89.32%</td>
</tr>
<tr>
<td>libquantum</td>
<td>3.88%</td>
<td>2.76%</td>
<td>203,078</td>
<td>203,581</td>
<td>99.75%</td>
</tr>
<tr>
<td>h264ref</td>
<td>0.49%</td>
<td>4.59%</td>
<td>30,707,102</td>
<td>32,452,013</td>
<td>94.62%</td>
</tr>
<tr>
<td>lbm</td>
<td>0%</td>
<td>0.93%</td>
<td>358</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>sphinx3</td>
<td>172%</td>
<td>27.62%</td>
<td>51,566,707</td>
<td>78,473,958</td>
<td>65.71%</td>
</tr>
</tbody>
</table>
Measurements on SPEC CPU 2006: -O0 vs -O3
## Results

### Pointer-Intensive benchmark suite

<table>
<thead>
<tr>
<th>Program</th>
<th>Runtime overhead</th>
<th># linear m.a.</th>
<th># instrumented m.a.</th>
<th>Percentage of linear m.a.</th>
</tr>
</thead>
<tbody>
<tr>
<td>anagram</td>
<td>-5.37%</td>
<td>134</td>
<td>159</td>
<td>84.27%</td>
</tr>
<tr>
<td>bc</td>
<td>183%</td>
<td>243,785</td>
<td>302,034</td>
<td>80.71%</td>
</tr>
<tr>
<td>ft</td>
<td>-8.46%</td>
<td>22</td>
<td>36</td>
<td>61.11%</td>
</tr>
<tr>
<td>ks</td>
<td>29.7%</td>
<td>29,524</td>
<td>42,298</td>
<td>69.79%</td>
</tr>
</tbody>
</table>
Outline

1. Motivation
2. State of the Art
3. Tracking code in LLVM IR using attached metadata
4. Interaction between high- and low-level IRs
5. Experiments
6. Conclusions
Open questions

- Promoting registers to memory (Phi node elimination)
- Maintain LLVM branches and jumps in inline assembly
- Type conversions
Perspectives

- Speculative code parallelization on the fly using multi-versioning
- Develop an easy-to-use API to extend the framework
Thank you.