A Mapping Approach Between IR and Binary CFGs dealing with Aggressive Compiler Optimizations for Performance Estimation

Omayma Matoussi
Frédéric Pétrot

TIMA Laboratory – 46 Avenue Félix Viallet, 38031, Grenoble, France

01/24/2018
Introduction

- MPSoCs are getting more software-centric.
- SW has an impact on the performance of MPSoCs.
- Accurate feedback on SW performance is necessary during early phases of MPSoC design.

⇒ Instruction Interpretation Approaches (ISS, DBT, etc.):
  - target instructions transformed into host instructions,
  - accurate,
  - very slow.
Introduction

- MPSoCs are getting more software-centric.
- SW has an impact on the performance of MPSoCs.
- Accurate feedback on SW performance is necessary during early phases of MPSoC design.

⇒ Instruction Interpretation Approaches (ISS, DBT, etc.):
  - target instructions transformed into host instructions,
  - accurate,
  - very slow.

⇒ Native Simulation (a.k.a. host-compiled simulation):
  - SW compiled and executed on the host machine,
  - abstraction of low-level architectural details,
  - fast.
Overview of a Native simulation platform

Execution Unit (EU) implements:

- Hardware Abstraction Layer (HAL) API.
Introduction

Lack of performance information in Native Simulation

- Originally developed for *purely functional verification* of SW on top of a virtual platform,
- Absence of non-functional information (e.g. execution time).

⇒ How to obtain *performance estimates* using Native Simulation?
1. Introduction
2. Software Back-annotation
3. The Proposed Mapping Approach
4. Experimentation
5. Conclusion
Outline

1. Introduction
2. Software Back-annotation
3. The Proposed Mapping Approach
4. Experimentation
5. Conclusion
Non-functional information (e.g. timing properties) is computed using low-level analysis and is inserted into the functional model (SW).

```c
1: for (i = 0; i < n; i++) {
2: nb_instr+=6;
3: a = b + c;
4: T[i] = (i+1)*a;
}
```

Source code annotation
Software Back-annotation

- How to compute non-functional information? (target binary analysis + modeling micro-architectural components)

- How to introduce target-specific performance metrics into the functional model (SW)?
  - Which software representation (source code, compiler Intermediate Representation-IR or target binary code) to opt for?
  - How to find correspondences between target binary control flow graph (CFG) and high-level code CFG when:
    - compiler optimizations, even the aggressive ones, are enabled (e.g. gcc -O3)?
Software Back-annotation

- How to compute non-functional information? (target binary analysis + modeling micro-architectural components)

- How to introduce target-specific performance metrics into the functional model (SW)?
  - Which software representation (source code, compiler Intermediate Representation-IR or target binary code) to opt for?
  - How to find correspondences between target binary control flow graph (CFG) and high-level code CFG when:
    - compiler optimizations, even the aggressive ones, are enabled (e.g. gcc -O3)?
Outline

1. Introduction
2. Software Back-annotation
3. The Proposed Mapping Approach
4. Experimentation
5. Conclusion
IR-Level Annotation Framework

(source code) → cross compiler → IR-CFG

IR-CFG → Gimple-CFG-To-C → compilable IR-CFG

compilable IR-CFG → time analysis and annot insertion → annotated compilable IR-CFG

annotated compilable IR-CFG → CFG mapping → mapping data base

mapping data base → BB data base

BB data base → native sim platform

source code → Extract CFG → target binary

target binary → Extract BB Info → target binary CFG
IR-Level Annotation Framework

- IR-CFG
  - Gimple-CFG-To-C
  - time analysis and annot insertion
    - annotated compilable IR-CFG
  - CFG mapping
    - mapping data base
- source code
  - cross compiler
  - target binary
- target binary CFG
  - Extract CFG
  - Extract BB Info
    - BB data base
  - native sim platform
IR-Level Annotation Framework

source code → cross compiler → IR-CFG

IR-CFG → Gimple-CFG-To-C → compilable IR-CFG

compilable IR-CFG → time analysis and annot insertion → native sim platform

annotated compilable IR-CFG

mapping data base

mapping data base

BB data base

target binary

target binary CFG

Extract CFG

Extract BB Info
IR-Level Annotation Framework

- **Source Code**
  - Cross compiler
  - IR-CFG
  - Gimple-CFG-To-C
  - Compilable IR-CFG
  - Time analysis and annot insertion
  - Annotated compilable IR-CFG
  - Native sim platform

- **Target Binary**
  - Extract CFG
  - CFG mapping
  - Mapping data base
  - BB data base

- **Target Binary CFG**
  - Extract BB Info
IR-Level Annotation Framework

source code → cross compiler → IR-CFG

IR-CFG → Gimple-CFG-To-C → compilable IR-CFG

compilable IR-CFG → CFG mapping → mapping data base

mapping data base → time analysis and annot insertion → annotated compilable IR-CFG

annotated compilable IR-CFG → native sim platform

target binary → Extract CFG → target binary CFG

target binary CFG → Extract BB Info → BB data base
IR-Level Annotation Framework

1. Source code
2. Cross compiler
3. IR-CFG
4. Gimple-CFG-To-C
5. Compilable IR-CFG
6. Time analysis and annot insertion
7. Annotated compilable IR-CFG
8. CFG mapping
9. Mapping data base
10. BB data base
11. Target binary CFG
12. Extract BB Info
13. Target binary
14. Extract CFG
15. Native sim platform
Choice of the Software Representation

- How to accurately place non-functional information into the functional model?
  - **Choice of the Intermediate Representation (IR),**
  - Accurate mapping of the functional model to the target binary code.

GCC’s intermediate representations
IR and binary CFGs Are Not Always Identical

```c
int i=0
i < n
exit
a[i] = b[i] + c[i]
```

Source Code CFG

IR CFG (gcc -O3)

Binary CFG (gcc -O3)
Existing Mapping Approach

IR CFG (gcc -O3)

Binary CFG (gcc -O3)

Existing mapping algorithm is efficient with O2 optimization level

Existing Mapping Approach

IR CFG (gcc -O3)

Binary CFG (gcc -O3)

SCC: Strongly Connected Component

Existing Mapping Approach

IR CFG (gcc -O3)

Binary CFG (gcc -O3)

Loop block contraction

Existing Mapping Approach

- Entry SCCs are fixed-points,
- Loop blocks are fixed-points,
- fixed-points are propagated using \( PRED(SCC) \) and \( SUCC(SCC) \).

---

Existing Mapping Approach

- Entry SCCs are fixed-points,
- Loop blocks are fixed-points,
- fixed-points are propagated using $\text{PRED}(\text{SCC})$ and $\text{SUCC}(\text{SCC})$.

Existing Mapping Approach

IR CFG (gcc -O3)

Binary CFG (gcc -O3)

\[ bb_{bin}^{gcc}, bb_{bin}^{gcc}, ..., bb_{bin}^{gcc} \] have no match in the IR

---

Loop Unrolling replicates the loop body **UF** (Unrolling Factor) times.
case1 :

- The loop trip count is **known** at compile time
  - The trip count is a **multiple** of $(U F + 1)$

(a) IR loop (max itr bound=20)

```plaintext
loop_body(i)
cnt++
if (cnt <= 2) {
    inst_count+=nb_inst}
i++
i<20
```

(b) Unrolled binary loop (max itr bound=2, UF=9)

```plaintext
loop_body(i)
loop_body(i+1)
...
loop_body(i+9)
i+=10
i<20
```
case2:

- The loop trip count is **known** at compile time
  - The trip count is **NOT a multiple** of \((U F + 1)\)

(a) IR loop (max itr bound=23)

```
...  
  inst_count+=nb_inst

  loop_body(i)
  cnt++
  if (cnt <= 2)
  {inst_count+=nb_inst}
  i++

  i<23
```

(b) Unrolled binary loop (max itr bound=2, UF=10, first itr peeled)

```
...  
  loop_body(0)

  loop_body(i)
  loop_body(i+1)
  ...
  loop_body(i+10)
  i+=11

  i<23
```
Mapping Scheme for Aggressive Compiler Optimizations

**case3**:

- The loop trip count is **unknown** at compile time (gcc)

![Diagram](image)

- Only the innermost loop is unrolled.
- GCC adds a prologue.
- Number of tests depends on the UF.

**Partially Unrolled binary loop (gcc)**
case3:

- The loop trip count is **unknown** at compile time (gcc)

```c
// loop body(i)
int cnt = 0;
if (cnt <= (n/4)) {
    inst_count+=nb_inst
} i++
```

Adding a prologue with if statements to the IR

Partially Unrolled binary loop (gcc)
case3:

- The loop trip count is **unknown** at compile time (gcc)

```plaintext
Partially Unrolled binary loop (gcc)

```

```plaintext
if (n mod 4) == 0
    inst_count += nb_inst

if (n mod 4) == 1
    inst_count += nb_inst

if (n mod 4) == 2
    inst_count += nb_inst

if (n mod 4) == 3
    inst_count += nb_inst

loop_body(i)
    cnt++
    if (cnt <= (n/4))
        { inst_count += nb_inst }
    i++

i < n
```

```plaintext
Adding a prologue with if statements to the IR

```
Mapping Scheme for Aggressive Compiler Optimizations

- Miscellaneous Optimizations

(a) while loop

(b) do-while loop

Loop Inversion
Mapping Scheme for Aggressive Compiler Optimizations

- Miscellaneous Optimizations

(a) If-then-else in the IR

(b) If-conversion in the binary

If conversion
Outline

1 Introduction
2 Software Back-annotation
3 The Proposed Mapping Approach
4 Experimentation
5 Conclusion
Experimentation

- Target architecture: Kalray k1 core,
- Host Machine: Intel x86-64 core,
- Native simulation platform:
  - Based on KVM,
  - The HW components are modeled with SystemC-TLM,
- ISS provided by Kalray.
### Table 1: A sample of the used benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Polybench</td>
<td></td>
</tr>
<tr>
<td>covar</td>
<td>Covariance Computation</td>
</tr>
<tr>
<td>atax</td>
<td>Matrix Transpose and Vector Multiplication</td>
</tr>
<tr>
<td>reg-detect</td>
<td>2-D Image processing</td>
</tr>
<tr>
<td>trmm</td>
<td>Triangular matrix-multiply</td>
</tr>
<tr>
<td>other</td>
<td></td>
</tr>
<tr>
<td>matmult</td>
<td>1 Matrix Multiplication</td>
</tr>
<tr>
<td>bubbleSort</td>
<td>Bubble Sort</td>
</tr>
<tr>
<td>blowfish</td>
<td>Symmetric-key block cipher</td>
</tr>
</tbody>
</table>
Table 2: Comparison of the simulation time

<table>
<thead>
<tr>
<th>sim_time(s)</th>
<th>matmult</th>
<th>bubbleSort</th>
<th>covar</th>
<th>atax</th>
<th>reg-detect</th>
<th>trmm</th>
<th>gemver</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISS</td>
<td>0.624</td>
<td>2.863</td>
<td>9.006</td>
<td>2.020</td>
<td>1.396</td>
<td>38.086</td>
<td>4.208</td>
</tr>
<tr>
<td>ILS+O3Map</td>
<td>0.180</td>
<td>0.184</td>
<td>0.284</td>
<td>0.196</td>
<td>0.180</td>
<td>0.348</td>
<td>0.196</td>
</tr>
<tr>
<td>speedup_O3Map</td>
<td>3.47</td>
<td>15.56</td>
<td>31.71</td>
<td>10.31</td>
<td>7.76</td>
<td>109.44</td>
<td>21.47</td>
</tr>
<tr>
<td>ILS+O2Map</td>
<td>0.170</td>
<td>0.180</td>
<td>0.282</td>
<td>0.192</td>
<td>0.172</td>
<td>0.348</td>
<td>0.188</td>
</tr>
<tr>
<td>speedup_O2Map</td>
<td>3.67</td>
<td>15.91</td>
<td>31.94</td>
<td>10.52</td>
<td>8.12</td>
<td>109.44</td>
<td>22.38</td>
</tr>
<tr>
<td>ILS+O2Map+</td>
<td>0.176</td>
<td>0.180</td>
<td>0.282</td>
<td>0.194</td>
<td>0.176</td>
<td>0.348</td>
<td>0.188</td>
</tr>
<tr>
<td>speedup_O2Map+</td>
<td>3.56</td>
<td>15.91</td>
<td>31.94</td>
<td>10.41</td>
<td>7.93</td>
<td>109.44</td>
<td>22.38</td>
</tr>
</tbody>
</table>

\[
\text{speedup}(\text{ILS} + \text{O}\times\text{Map}) = \frac{\text{sim\_time}(\text{ISS})}{\text{sim\_time}(\text{ILS} + \text{O}\times\text{Map})}.
\]
## Table 3: Comparison of the instruction count

<table>
<thead>
<tr>
<th>instr_count</th>
<th>matmult</th>
<th>bubbleSort</th>
<th>covar</th>
<th>atax</th>
<th>reg-detect</th>
<th>trmm</th>
<th>gemver</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISS</td>
<td>155993</td>
<td>2646028</td>
<td>151302</td>
<td>25748</td>
<td>9892</td>
<td>136033</td>
<td>40556</td>
</tr>
<tr>
<td>ILS+O3Map</td>
<td>155993</td>
<td>2656128</td>
<td>154561</td>
<td>25684</td>
<td>10011</td>
<td>136321</td>
<td>40809</td>
</tr>
<tr>
<td>error_O3Map</td>
<td>+0.0%</td>
<td>+0.38%</td>
<td>+2.15%</td>
<td>-0.25%</td>
<td>+1.2%</td>
<td>+0.21%</td>
<td>+0.62%</td>
</tr>
<tr>
<td>ILS+O2Map</td>
<td>954293</td>
<td>10498510</td>
<td>902115</td>
<td>109985</td>
<td>18213</td>
<td>862273</td>
<td>176398</td>
</tr>
<tr>
<td>error_O2Map</td>
<td>+512%</td>
<td>+297%</td>
<td>+496%</td>
<td>+327%</td>
<td>+84%</td>
<td>+534%</td>
<td>+335%</td>
</tr>
<tr>
<td>ILS+O2Map+</td>
<td>102893</td>
<td>3600010</td>
<td>98327</td>
<td>14625</td>
<td>5741</td>
<td>88129</td>
<td>29686</td>
</tr>
<tr>
<td>error_O2Map+</td>
<td>-34%</td>
<td>+36%</td>
<td>-35%</td>
<td>-43%</td>
<td>-42%</td>
<td>-35%</td>
<td>-27%</td>
</tr>
</tbody>
</table>

\[
\text{error(\%)} = \frac{|\text{nb\_exec\_instrs}(\text{ILS + OxMap}) - \text{nb\_exec\_instrs}(\text{ISS})|}{\text{nb\_exec\_instrs}(\text{ISS})} \times 100.
\]
Outline

1. Introduction
2. Software Back-annotation
3. The Proposed Mapping Approach
4. Experimentation
5. Conclusion
Conclusion

- We proposed a mapping approach between IR and binary CFGs, when aggressive compiler optimizations (gcc -O3) are enabled.
- We modify the IR CFG without changing its functional behavior.
- Experiments show considerable speedup yet high accuracy in instruction count.
A Mapping Approach Between IR and Binary CFGs dealing with Aggressive Compiler Optimizations for Performance Estimation

Omayma Matoussi
Frédéric Pétrot

TIMA Laboratory – 46 Avenue Félix Viallet, 38031, Grenoble, France

01/24/2018