Precise Simulation of Interrupts using a Rollback Mechanism

Florian Brandner

Vienna University of Technology
Institute of Computer Languages
Christian Doppler Laboratory - Compilation Techniques for Embedded Systems
Outline

- Introduction
  - Architecture description language
  - Instruction set simulation

- Simulation of interrupts
  - Simple approaches
  - Optimized by a rollback mechanism

- Evaluation
  - Simulation performance
  - Rollback behavior

- Conclusion
Introduction

- Architecture description language
  - Meta-Information
  - Structural processor models
  - Implicit definition of instruction set
    - Parallel instructions/Interrupts

- Generator backends
  - Compiler backend
  - Instruction set simulator
  - XML report
  - VHDL model (prototype)
Example: MIPS Model

Instructions discovered along paths:
- FETCH→CACHE→DECODE→ALU→CACHE→COMMIT
- FETCH→CACHE→DECODE→ALU→CACHE
- FETCH→CACHE→DECODE→ALU→COMMIT

Load operations
- Store operations
- Arithmetic and Branch

Parallel Instructions/Interrupts
Instruction Set Simulation

- Cycle-accurate emulation
- Mixed-mode simulation
  - Simple interpreter
    - Simulate each pipeline stage separately
  - Hot linear code
    - Compiled into basic block functions using LLVM
    - Executed as native code
  - Hot basic blocks
    - Compiled into region functions
    - May contain arbitrary control flow
    - Including loops
- Automatically derived from architecture models
Example: Simulation

<table>
<thead>
<tr>
<th>Basic Blocks:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compile linear code into machine code of the host machine</td>
</tr>
<tr>
<td>Execute complete basic blocks one by one</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Regions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Combine multiple basic blocks</td>
</tr>
<tr>
<td>Compile to machine code</td>
</tr>
<tr>
<td>Simulate regions one by one</td>
</tr>
</tbody>
</table>

Example: Simulation

- **Region 1 (BB1):**
  - `xor` `a0,a1,a0` (WB)
  - `andi` `a0,a0,0xff` (MEM)
  - `sll` `a0,a0,0x2` (MEM)
  - `lui` `v0,0x8003` (EX)
  - `beq` `v0, BB3` (RO)
  - `nop` (DE)
  - `srl` `a1,a1,0x8` (BB2)

- **Region 2 (BB3):**
  - `bne` `a1, BB1` (BB3)
  - `xor` `v0,a1,v0` (BB3)

- **Regions:**
  - Combine multiple basic blocks
  - Simulate regions one by one
Simple Interrupt Handling

- Emit code for interrupt check and dispatch on every cycle

- Problems
  - Interrupts are rare events!
  - Increased compile time and code size
    - Due to the interrupt check and dispatch
    - Most of the code is useless though
  - Low simulation speed
    - Interrupt check always needs to be executed
    - Optimizations are impeded by interrupt code
Simple Interrupt Handling

Cycle Simulation-Action

<table>
<thead>
<tr>
<th>N</th>
<th>(a) increment time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(b) simulate instruction</td>
</tr>
<tr>
<td></td>
<td>(c) if (interrupt) jump exit</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>N+1</th>
<th>(a) increment time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(b) simulate instruction</td>
</tr>
<tr>
<td></td>
<td>(c) if (interrupt) jump exit</td>
</tr>
</tbody>
</table>

...  

exit: store state

- Cycle-accurate interrupt simulation is thus **avoided**
  - Instead only done at certain points
  - Example: basic block boundaries
Using Rollbacks

- Assume interrupts do not occur while executing compiled code
  - On exit verify this assumption
  - Rollback to previous `restore-point` when interrupt missed
    - Restart simulation using interpreter
    - Interpreter models interrupts faithfully

- Advantages
  - Reduced code size
  - Reduced compilation time
  - Improved simulation speed
### Using Rollbacks (2)

<table>
<thead>
<tr>
<th>Cycle Simulation-Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>( N ) simulate instruction</td>
</tr>
<tr>
<td>( N+1 ) simulate instruction</td>
</tr>
<tr>
<td>( N+2 ) simulate instruction</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>( N+n ) simulate instruction</td>
</tr>
<tr>
<td>increment time by ( n )</td>
</tr>
<tr>
<td>if (interrupt-missed) jump rollback</td>
</tr>
<tr>
<td>exit: store state</td>
</tr>
<tr>
<td>rollback: revert state</td>
</tr>
</tbody>
</table>

*Still rarely taken*

*Optimizations possible*

*Restore-point*
Implementing Rollbacks

- Shadow architecture state in local variables
  - Restore-points
    - Program point that last updated the global state
    - Usually at end of compiled code
    - Special care within regions
  - On rollback simply discard values and return
    - Otherwise copy the local to the global state

- Memories can not be shadowed!
  - Establish restore-points around store operations
Evaluation

- 3GHz Xeon CPU, 24 GB RAM, Linux 2.6.18
- Large subset of MiBench
  - Compiled using GCC 4.2 for mips-elf target
  - With standard optimizations
- 3 Configurations of MIPS-r2000
  - Simple approach (check for interrupts on every cycle)
  - Optimized using rollback mechanism
  - No interrupt support (no overhead)
- Periodic interrupt every 20,000 cycles
- Initially determine optimal thresholds for each configuration
### Relative Speedup

<table>
<thead>
<tr>
<th>Function</th>
<th>Simple</th>
<th>Rollback</th>
<th>No-interrupt</th>
</tr>
</thead>
<tbody>
<tr>
<td>adpcm</td>
<td>1.69</td>
<td>2.1</td>
<td>1.69</td>
</tr>
<tr>
<td>bitcount</td>
<td>1.64</td>
<td>1.2</td>
<td>1.64</td>
</tr>
<tr>
<td>blowfish</td>
<td>1.63</td>
<td>1.43</td>
<td>1.63</td>
</tr>
<tr>
<td>crc32</td>
<td>2.09</td>
<td>1.46</td>
<td>2.09</td>
</tr>
<tr>
<td>dijkstra</td>
<td>1.75</td>
<td>2.1</td>
<td>1.75</td>
</tr>
<tr>
<td>gsm</td>
<td>2.78</td>
<td>1.84</td>
<td>2.78</td>
</tr>
<tr>
<td>jpeg</td>
<td>2.15</td>
<td>2.81</td>
<td>2.15</td>
</tr>
<tr>
<td>prime</td>
<td>2.95</td>
<td>3.86</td>
<td>2.95</td>
</tr>
<tr>
<td>rijndael</td>
<td>3.86</td>
<td>3.86</td>
<td>3.86</td>
</tr>
<tr>
<td>sha</td>
<td>2.25</td>
<td>1.3</td>
<td>2.25</td>
</tr>
<tr>
<td>stringsearch</td>
<td>1.68</td>
<td>1.3</td>
<td>1.68</td>
</tr>
<tr>
<td>average</td>
<td>2.25</td>
<td>2.25</td>
<td>2.25</td>
</tr>
</tbody>
</table>

**Speedups up to 2.95x, only two cases show slowdowns!**

**Similar results for code size and compilation time.**
### Cycles reverted per rollback

<table>
<thead>
<tr>
<th>benchmark</th>
<th>avg</th>
<th>min</th>
<th>max</th>
<th>Std. dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>adpcm</td>
<td>5</td>
<td>1</td>
<td>11</td>
<td>2.74</td>
</tr>
<tr>
<td>bitcount</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>blowfish</td>
<td>396</td>
<td>1</td>
<td>422</td>
<td>100.41</td>
</tr>
<tr>
<td>crc32</td>
<td>8</td>
<td>1</td>
<td>12</td>
<td>3.7</td>
</tr>
<tr>
<td>dijkstra</td>
<td>7</td>
<td>1</td>
<td>16</td>
<td>4.17</td>
</tr>
<tr>
<td>gsm</td>
<td>275</td>
<td>1</td>
<td>477</td>
<td>228.64</td>
</tr>
<tr>
<td>jpeg</td>
<td>17</td>
<td>1</td>
<td>160</td>
<td>27.3</td>
</tr>
<tr>
<td>prime</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>1.33</td>
</tr>
<tr>
<td>rijndael</td>
<td>185</td>
<td>1</td>
<td>538</td>
<td>235.47</td>
</tr>
<tr>
<td>sha</td>
<td>19</td>
<td>1</td>
<td>32</td>
<td>7.21</td>
</tr>
<tr>
<td>stringsearch</td>
<td>4</td>
<td>1</td>
<td>9</td>
<td>1.84</td>
</tr>
</tbody>
</table>

The number of cycles reverted by a rollback is small. The fraction of reverted cycles is very low and never exceeds 1.67%.
Conclusion

- Rollbacks are very efficient for interrupt simulation
  - Reduced code size
  - Reduced compilation time
    - Even for lower thresholds
  - Improved simulation speed, up to 2.95x
    - Due to reduced overhead
    - More effective optimizations

- May also be beneficial for other rare events
  - Examples: Branch predictors, Caches
Compilation Time

Compile time reduced to 34% in the best case, despite lower compilation thresholds.