Using Binary Translation in Event Driven Simulation for Fast and Flexible MPSoC Simulation

Marius Gligor, Nicolas Fournel and Frédéric Pétrot

TIMA Laboratory - SLS Group, INP-UJF-CNRS
Grenoble, France

October 18, 2010
Outline

- **Introduction**
  - Context & Motivations

- Using binary translation in event driven simulation

- Experimental results

- Conclusion and Perspectives
MPSoC Perspectives

ITRS Roadmap for the number of cores in consumer devices

Characteristics of future SoCs architecture

- More and more homogeneous
- Massively parallel
- Programmability is a major concern
Introduction -> Context & Motivations

Problems

MPSoCs based on GPPs

▶ Great flexibility, short time-to-market
▶ Great reusability
  ◆ Use GPP and legacy code (e.g. OS)

System simulation

▶ Architecture validation and early software coding
▶ Optimize hardware/software performances and consumption through architecture exploration
▶ Speed/accuracy trade-off required
  ◆ Several abstraction levels
Abstraction Levels

System

Virtual architecture

Transaction accurate

Cycle approximate / accurate
Motivations

Classical TA solution: interpretive ISS

- Pros: flexible and precise
- Cons: low simulation speed

Binary translation based ISS

- Pros: fast
- Cons: no time notion for the simulated platform

Solution: Transform a binary translation based ISS into an accurate component that can be used in an event-driven simulation

Possible candidates

- Binary translation based ISS: QEMU, Bochs, ...
- Event driven simulator: SystemC is the current solution for MPSoC simulation
Outline

- Introduction

- Using binary translation in event driven simulation
  - Background technologies
  - Multiprocessor modeling
  - Time modeling
  - Frequency and energy modeling

- Experimental results

- Conclusion and Perspectives
QEMU / SystemC TLM

QEMU

- Fast and portable emulator
- Emulates multiple target architectures (e.g. x86, ARM, SPARC) on multiple host architectures
- Based on dynamic binary translation and dynamic code generation
- 5 to 20 times slower than native code execution

SystemC TLM

- Use of transactions for data exchange
- Discrete event-driven simulation
  - Concurrency of processes
Using binary translation in event driven simulation

---

QEMU Emulation Process

**Process**

1. **PC already seen?**
   - Yes
   - No

   - **Fetch**
     - Binary Translation
     - **Decode**
       - Branch?
         - No
         - **Execute**
       - Yes

   - **Target binary code (.elf)**
     - Micro-operations built-in
     - Translation Cache (host binary code)

   - **Instruction**
     - micro-ops identifiers buffer
     - Tiny code generator

---

M. Gligor - Using Binary Translation in Event Driven Simulation for Fast and Flexible MPSoC Simulation
QEMU Emulation Process

Process

PC already seen? Yes | No

Fetch

Binary Translation

Decode

Branch?

Execute

micro-ops identifiers buffer

Tiny code generator

TB Cache Entry

Target binary code (.elf)

Instruction

Translation Cache (host binary code)

Micro-operations built-in

Code generation example

18 targetInstrX
QEMU Emulation Process

Process

PC already seen?

Yes

No

Fetch

Binary Translation

No

Branch?

Yes

Execute

No

Decode

micro-ops
identifiers buffer

Tiny code
generator

Micro-operations built-in

Translation Cache
(host binary code)

TB Cache Entry

Target binary code (.elf)

Instruction

Code generation example

18 target_instrX

micro-op1_instrX

micro-op2_instrX
QEMU Emulation Process

**Process**

1. **PC already seen?**
   - Yes: **Instruction**
     - Target binary code (.elf)
   - No: **Fetch**
     - Binary Translation
     - **Fetch**
       - **Decode**
     - **Branch?**
       - Yes: **Execute**
       - No: **No**
         - **Micro-operations built-in**
         - **Tiny code generator**
         - **Translation Cache**
           - (host binary code)
         - **TB Cache Entry**

**Code generation example**

18  target_instrX

<table>
<thead>
<tr>
<th>micro-op1_instrX</th>
<th>host_instr1_micro-op1_instrX</th>
</tr>
</thead>
<tbody>
<tr>
<td>host_instr2_micro-op1_instrX</td>
<td></td>
</tr>
<tr>
<td>host_instr3_micro-op1_instrX</td>
<td></td>
</tr>
<tr>
<td>micro-op2_instrX</td>
<td>host_instr1_micro-op2_instrX</td>
</tr>
</tbody>
</table>
ISS Wrapping and Connection

ISS SystemC wrapper

- Simulates independently under the SystemC control in the context of a SystemC thread
- Connected to interconnect
- Implements instruction and data caches
- A SystemC thread - interface between SystemC interrupt signals and ISS

ISS group

- Groups the processors that may share the same translation cache
- Identical processors

SystemC timed TLM components

- Traffic generator, timers, main memory, spinlocks, interconnect, RAMDAC, TTYs
ISS Wrapper/SystemC Synchronization

ISS component must synchronize with SystemC

- Consume the time corresponding to the simulated cycles

Synchronization points

- Cache misses (instruction and data caches)
- I/O operations
- Target synchronization instructions (e.g. load and store exclusive)
- Predefined period without synchronization

Interrupts forwarding

- Generated by hardware components during the SystemC activities of the processors
- Interrupt treated after the current SystemC activity ends, at the beginning of the next translation block
Using binary translation in event driven simulation - Time modeling

**Code Annotation**

**Motivation**

- Generated code offers no time information about the target execution
- Accurate time modeling of the ISS

**Insert micro-operations**

- To increment the number of simulated cycles
  - Inserted before the micro-operations of each target instructions
  - Use the target processor datasheet
  - Take into account registers dependencies, branch prediction
- To emulate target caches (instruction and data) and write buffer

**Annotation example**

<table>
<thead>
<tr>
<th>Instr address</th>
<th>Target code</th>
<th>Original translation</th>
<th>Annotated translation</th>
<th>Annotated generated code</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr_instrX</td>
<td>target_instrX</td>
<td>micro-op1_instrX</td>
<td>micro-op1_instrX</td>
<td>host_instr1_micro-op1_instrX</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>host_instr2_micro-op1_instrX</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>host_instr3_micro-op1_instrX</td>
</tr>
<tr>
<td>micro-op2_instrX</td>
<td></td>
<td>micro-op_annotation</td>
<td></td>
<td>host_instr1_micro-op_annotation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>host_instr2_micro-op_annotation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>micro-op2_instrX</td>
<td>host_instr1_micro-op2_instrX</td>
</tr>
</tbody>
</table>
Using binary translation in event driven simulation -> Time modeling

**Code Annotation: Details & Example**

### Instruction Cache

- **Where?**
  - At the beginning of each translation block
  - At the beginning of each cache line
- **What?**
  - Synchronize simulated cycles
  - Request over the interconnect

### Data cache

- **Where?**
  - At each data access (read and write)
- **What?**
  - On read miss: synchronize and fill the cache line using the interconnect
  - On write hit: update the value in cache
  - On write: update the value in memory through interconnect (write through policy)

<table>
<thead>
<tr>
<th>Instr address</th>
<th>Target code</th>
<th>Original translation</th>
<th>Annotated translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>start_tb: 18</td>
<td>instr1_reg_operation</td>
<td>micro-op1_for_instr1</td>
<td>instr_cache_verify (18);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>micro-opN1_for_instr1</td>
<td>nb_cycles += cpu_datasheet [instr1];</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>micro-op1_for_instr1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>micro-opN1_for_instr1</td>
</tr>
<tr>
<td>1C</td>
<td>instr2_load_from_1000</td>
<td>micro-op1_for_instr2</td>
<td>nb_cycles += cpu_datasheet [instr2];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>micro-opN2_for_instr2</td>
<td>data_cache_verify (1000);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>micro-op1_for_instr2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>micro-opN2_for_instr2</td>
</tr>
<tr>
<td>20</td>
<td>instr3_store_5_to_2000</td>
<td>micro-op1_for_instr3</td>
<td>instr_cache_verify (20);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>micro-opN3_for_instr3</td>
<td>nb_cycles += cpu_datasheet [instr3];</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>write_access (2000, 5);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>micro-op1_for_instr3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>micro-opN3_for_instr3</td>
</tr>
</tbody>
</table>
Precision Levels

- Simulation speed/accuracy trade-off
- Caches full
  - Search data and instructions over the interconnect
  - Ignore instructions from cache (instructions are simulated from QEMU translation cache)

![Diagram of CPU, ICACHE, DCACHE, RAM, and Other peripherals connected through Interconnect]

Figure: Cache full
Precision Levels

- Simulation speed/accuracy trade-off
- Caches full
  - Search data and instructions over the interconnect
  - Ignore instructions from cache (instructions are simulated from QEMU translation cache)
- No caches (0 memory access time)

Figure: No cache
Precision Levels

- Simulation speed/accuracy trade-off
- Caches full
  - Search data and instructions over the interconnect
  - Ignore instructions from cache (instructions are simulated from QEMU translation cache)
- No caches (0 memory access time)
- Caches as pure directories
  - Precomputed time consumed
    - Cache wait: when the miss occurs
    - Cache late: at the next synchronization

Figure: Cache wait & late
Frequency and Energy Modeling

Frequency modeling

- Each processor can be simulated at a different frequency
- Compute time corresponding to the number of cycles to synchronize \( t = \frac{nb\_cycles}{fq} \)
- A processor can change the frequency of other processors

Energy modeling  (in collaboration with Nicolas Fournel)

- Power consumed by a processor at a given moment depends on the current activity (e.g. instruction execution, IDLE state) and the current frequency and voltage of the processor
  \[ E_t^{CPU} = \sum (P_{CPU}[Activity_i][f, v] \times t_i), \text{ where } \sum t_i = t \]
- Power consumed by a hardware component depends on the current running mode (display device: idle, display): \( E_t^{HW} = P_{HW} \times t \)
  - For each event \( Event_i \) (display device: read, write), energy \( E_{Event_i}^{HW} \) is added
Outline

- Introduction
- Using binary translation in event driven simulation
- Experimental results
- Conclusion and Perspectives
Motion-JPEG Application

Software stack

- Motion-JPEG decoding application
- Mutek operating system
  - POSIX compliant
  - SMP version

Hardware platform

- Processors
- Caches
- Interconnect
- Memories
- Timers
- DVFSes
- ...

Experimental results ->
Experimental Results - Accuracy

Virtual Platforms

- Transaction level: modeled with QEMU/SystemC
- CABA reference: modeled with ISS/SoClib/SystemCASS

Table: Monoprocessor results

<table>
<thead>
<tr>
<th></th>
<th>No cache</th>
<th>Cache late</th>
<th>Cache wait</th>
<th>Cache full</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction cycles error</td>
<td>0.00 %</td>
<td>0.00 %</td>
<td>0.00 %</td>
<td>0.00 %</td>
</tr>
<tr>
<td>Uncached accesses error</td>
<td>0.00 %</td>
<td>0.00 %</td>
<td>0.00 %</td>
<td>0.00 %</td>
</tr>
<tr>
<td>Simulated time error (accuracy error)</td>
<td>-36.70 %</td>
<td>-0.04 %</td>
<td>-0.04 %</td>
<td>-0.04 %</td>
</tr>
<tr>
<td>Sim. speedup (vs. Cache full)</td>
<td>19.38 X</td>
<td>12.48 X</td>
<td>1.94 X</td>
<td>1</td>
</tr>
<tr>
<td>Sim. speedup (vs. CABA reference)</td>
<td>553.10 X</td>
<td>356.13 X</td>
<td>55.39 X</td>
<td>28.55 X</td>
</tr>
</tbody>
</table>

Table: Multiprocessor results

<table>
<thead>
<tr>
<th></th>
<th>No cache</th>
<th>Cache late</th>
<th>Cache wait</th>
<th>Cache full</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction cycles error</td>
<td>35.13%</td>
<td>22.31%</td>
<td>5.24 %</td>
<td>2.56 %</td>
</tr>
<tr>
<td>Uncached accesses error</td>
<td>-44.54 %</td>
<td>-9.80 %</td>
<td>2.01 %</td>
<td>8.19 %</td>
</tr>
<tr>
<td>Simulated time error (accuracy error)</td>
<td>-21.07 %</td>
<td>1.34 %</td>
<td>-8.44 %</td>
<td>-3.42 %</td>
</tr>
<tr>
<td>Sim. speedup (vs. Cache full)</td>
<td>21.45 X</td>
<td>13.87 X</td>
<td>2.03 X</td>
<td>1</td>
</tr>
<tr>
<td>Sim. speedup (vs. CABA reference)</td>
<td>381.01 X</td>
<td>246.38 X</td>
<td>35.97 X</td>
<td>17.76 X</td>
</tr>
</tbody>
</table>
Experimental Results - Simulation Speed

Virtual Platforms

- Transaction level: modeled with QEMU/SystemC
- Transaction level reference
  - Interpretative SoClib ISS
  - Rest of the hardware components from our simulation platform
- Results: "Cache full" configuration is 2 times faster than the reference platform

Other simulation speed results ("No cache" configuration)

- Linux boot: 31s (against 3.3h estimated for CABA)
- H264 frame decoding: 2s (against 12.7min estimated for CABA)
Limitations

- The processors pipeline is not modeled
- Usual drawback of TLM - Time modeling is rough
Outline

- Introduction
- Using binary translation in event driven simulation
- Experimental results
- Conclusion and Perspectives
Conclusion and Perspectives

Simulation strategy implemented at the TA abstraction level

- Based on the existing untimed binary translation ISSes
- Allows a simulation speed/accuracy trade-off
- Supports runtime change of the processors frequency

Short term future works

- Integrate the ISS with cycle accurate hardware models of SoCLib

Long term future works

- Improve the accuracy (timing, power) modeling of the processors
Thank you!

{marius.gligor@imag.fr}
Conclusion and Perspectives

Binary Translation / SystemC Synchronization

Simulation at CA abstraction level
- Synchronization at least each clock edge

Binary translation based ISS (TLM)
- Synchronization after one or several simulated cycles
QEMU/SystemC Implementation Details