Using Hardware-Assisted Virtualization for Native Simulation of MPSoC

Frédéric Pétrot, Hao Shen, Mian M. Hamayun

System-Level Synthesis Group
TIMA Laboratory
46, Av Félix Viallet, 38031 Grenoble, France

July 18th, 2013
1 Introduction
   - Motivations/Issues
   - Memory Representations
   - Chip Memory Mappings vs. SystemC Memory Mapping

2 Proposed Solution – Native Simulation using HAV
   - Hardware Assisted Virtualization Using KVM
   - Address Translation using Memory Virtualization
   - Using HAV in Event Driven Simulation

3 Experiments and Results
   - Computation and I/O Speed Comparisons
   - Performance Estimation Results

4 Conclusions and Future Works
Native Software Execution In MPSoC Simulation

A *Technology that can be used for Rapid Architecture Exploration and Design of MPSoC Systems*

**Key Ideas of this Talk**
- Simulation of hardware/software systems
- Focusing on the fast simulation of software
- Within the model of a hardware environment (SystemC/TLM)
- Support for performance estimation/annotations

**Clarification**

- **Host**: machine on which the simulator is executed (e.g. x86)
- **Host code**: code directly executable on the host
- **Native code**: code executable on the host once linked with a simulator
- **Target**: machine which is simulated (e.g. ARM, MIPS)
**Motivations/Issues**

### Why native simulation?
- Software is Compiled for Host ⇒ Fastest Functional Simulations
- Software is Executed Natively ⇒ No need for ISS Development
- Software Executes in *Zero Time* w.r.t H/W ⇒ Functional Verification

### Key Issue
- H/W models use target addresses whereas S/W uses host addresses
  - Chip Memory Mapping
  - SystemC Memory Mapping

### Requirement: MPSoC platform for realistic native simulation
- Maximize the source code reusability
- Keep low level hardware details
Motivations/Issues

Why native simulation?
- Software is Compiled for Host ⇒ Fastest Functional Simulations
- Software is Executed Natively ⇒ No need for ISS Development
- Software Executes in Zero Time w.r.t H/W ⇒ Functional Verification

Key Issue
- H/W models use target addresses whereas S/W uses host addresses
  - Chip Memory Mapping
  - SystemC Memory Mapping

Requirement: MPSoC platform for realistic native simulation
- Maximize the source code reusability
- Keep low level hardware details
**Motivations/Issues**

**Why native simulation?**
- Software is Compiled for Host ⇒ Fastest Functional Simulations
- Software is Executed Natively ⇒ No need for ISS Development
- Software Executes in *Zero Time* w.r.t H/W ⇒ Functional Verification

**Key Issue**
- H/W models use target addresses whereas S/W uses host addresses
  - Chip Memory Mapping
  - SystemC Memory Mapping

**Requirement:** MPSoC platform for realistic native simulation
- Maximize the source code reusability
- Keep low level hardware details
Memory Representations

Real System or CABA Platforms
- SW binary loaded in memory
- HW address decoder uses physical chip memory mapping
- SW uses real chip memory mapping

Transaction Accurate Simulation
- SW dynamically loaded and executed on the Host
- HW address decoder uses physical chip memory mapping
- SW uses SystemC process memory mapping
Memory Representations

**Real System or CABA Platforms**
- SW binary loaded in memory
- HW address decoder uses physical chip memory mapping
- SW uses real chip memory mapping

**Transaction Accurate Simulation**
- SW dynamically loaded and executed on the Host
- HW address decoder uses physical chip memory mapping
- SW uses SystemC process memory mapping

Cross-compiled software binary in memory

ISS

MEMORY

Interconnect

ADC

DMA
Memory Representations

Real System or CABA Platforms
- SW binary loaded in memory
- HW address decoder uses physical chip memory mapping
- SW uses real chip memory mapping

Transaction Accurate Simulation
- SW dynamically loaded and executed on the Host
- HW address decoder uses physical chip memory mapping
- SW uses SystemC process memory mapping
Memory Representations

Real System or CABA Platforms
- SW binary loaded in memory
- HW address decoder uses physical chip memory mapping
- SW uses real chip memory mapping

Transaction Accurate Simulation
- SW dynamically loaded and executed on the Host
- HW address decoder uses physical chip memory mapping
- SW uses SystemC process memory mapping
Memory Representations

Real System or CABA Platforms
- SW binary loaded in memory
- HW address decoder uses physical chip memory mapping
- SW uses real chip memory mapping

Transaction Accurate Simulation
- SW dynamically loaded and executed on the Host
- HW address decoder uses physical chip memory mapping
- SW uses SystemC process memory mapping
Memory Representations

Real System or CABA Platforms
- SW binary loaded in memory
- HW address decoder uses physical chip memory mapping
- SW uses real chip memory mapping

Transaction Accurate Simulation
- SW dynamically loaded and executed on the Host
- HW address decoder uses physical chip memory mapping
- SW uses SystemC process memory mapping
Memory Representations

Real System or CABA Platforms
- SW binary loaded in memory
- HW address decoder uses physical chip memory mapping
- SW uses real chip memory mapping

Transaction Accurate Simulation
- SW dynamically loaded and executed on the Host
- HW address decoder uses physical chip memory mapping
- SW uses SystemC process memory mapping
Two uncorrelated memory mappings have to be considered

**Chip Memory Mapping (1)**
- Defined by the HW designers
- Used by the address decoder at simulation time

**SystemC Memory Mapping (2)**
- Shared by the SW stack
- Host machine dependent
- Contains standard sections
  - Program in `.text`
  - Initialized data in `.data`
  - Uninitialized data in `.bss`
Two uncorrelated memory mappings have to be considered

**Chip Memory Mapping (1)**
- Defined by the HW designers
- Used by the address decoder at simulation time

**SystemC Memory Mapping (2)**
- Shared by the SW stack
- Host machine dependent
- Contains standard sections
  - Program in `.text`
  - Initialized data in `.data`
  - Uninitialized data in `.bss`
Two uncorrelated memory mappings have to be considered

**Chip Memory Mapping (1)**
- Defined by the HW designers
- Used by the address decoder at simulation time

**SystemC Memory Mapping (2)**
- Shared by the SW stack
- Host machine dependent
- Contains standard sections
  - Program in `.text`
  - Initialized data in `.data`
  - Uninitialized data in `.bss`
Mixing both memory mappings?

### DMA Transfer Example
- Read data in the ADC
- Write data into memory

### Source in Chip Mapping (1)
- Source address is valid in address decoder
- So DMA can access source address

### Destination in SystemC Mapping (2)
- Destination address is valid in SW
- But *Invalid* in the address decoder
- So DMA cannot access the destination address
Mixing both memory mappings?

**DMA Transfer Example**
- Read data in the ADC
- Write data into memory

**Source in Chip Mapping (1)**
- Source address is valid in address decoder
- So DMA can access source address

**Destination in SystemC Mapping (2)**
- Destination address is valid in SW
- But *Invalid* in the address decoder
- So DMA cannot access the destination address
Mixing both memory mappings?

**DMA Transfer Example**
- Read data in the ADC
- Write data into memory

**Source in Chip Mapping (1)**
- Source address is valid in address decoder
- So DMA can access source address

**Destination in SystemC Mapping (2)**
- Destination address is valid in SW
- But *Invalid* in the address decoder
- So DMA cannot access the destination address
Mixing both memory mappings?

**DMA Transfer Example**
- Read data in the ADC
- Write data into memory

**Source in Chip Mapping (1)**
- Source address is valid in address decoder
- So DMA can access source address

**Destination in SystemC Mapping (2)**
- Destination address is valid in SW
- But *Invalid* in the address decoder
- So DMA cannot access the destination address
HW Support for CPU Virtualization

Saved by virtualization’s needs in the software industry!

**New Guest Operating Mode**
- Hardware Support for *Root* and *Non-Root* operations
  - Root Operations for VMM and Non-Root for Guest System
- Hardware Based Guest <-> Host Mode Switching
- Guest Mode Exit Reason Reporting
  - PMIO, MMIO, Signal Pending, Shutdown?

Available in most popular CPUs since the mid 2000’s:
- x86 (Intel, AMD), Power, Cortex A15 (ARM), Sparc, ...

**New Transitions**
- VM Entry and VM Exit
- Swapping of Registers and Address Space in one Atomic Operation

**VM Control Structure (VMCS)**
- Controlled by software
- Keeps track of Guest OS State
- Controls when VM Exits occur
**HW Support for Memory Virtualization**

**HAV based Memory Virtualization**

- Software is Compiled as a *Static* binary and executes in Target Address-Space
- SystemC models simulate target addresses and remain un-modified
- Translation layer provides bidirectional accesses between SW and HW components.

**Hard-coded addresses in SW**

Even hard-coded addresses can be used in Software i.e.

```
((uint8_t *) 0x0A000010) = 0x33;
```

**Memory Virtualization != Virtual Memory**
Proposed Solution – Native Simulation using HAV
Address Translation using Memory Virtualization

HW Support for Memory Virtualization

HAV based Memory Virtualization

- Software is Compiled as a Static binary and executes in Target Address-Space
- SystemC models simulate target addresses and remain un-modified
- Translation layer provides bidirectional accesses between SW and HW components.

Hard-coded addresses in SW

Even hard-coded addresses can be used in Software i.e. `((uint8_t *) 0x0A000010) = 0x33;`

Memory Virtualization != Virtual Memory
Proposed Solution – Native Simulation using HAV

Address Translation using Memory Virtualization

HW Support for Memory Virtualization

HAV based Memory Virtualization

- Software is Compiled as a *Static* binary and executes in Target Address-Space
- SystemC models simulate target addresses and remain un-modified
- Translation layer provides bidirectional accesses between SW and HW components.

Hard-coded addresses in SW

Even hard-coded addresses can be used in Software i.e.

```c
((uint8_t *) 0xA000010) = 0x33;
```

Memory Virtualization != Virtual Memory
Using HAV in Event Driven Simulation

**Transparent Memory Accesses**
- Using the Virtual MMU implementation in KVM
- Full address space available in Guest mode
- Transparent access i.e. No mode switch on memory accesses

**I/O Emulation**
- I/O (MMIO & PMIO) accesses force VM Exits
- We exploit PMIO exits for providing Semi-hosting support. e.g. Annotations
## Experiments and Results

### Computation and I/O Speed Comparisons

#### Computation Speed Comparisons

<table>
<thead>
<tr>
<th>Application</th>
<th>QEMU</th>
<th>Native</th>
<th>KVM</th>
<th>Host</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>SusanX5</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>QsortX100</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
</tr>
<tr>
<td>DijkstraX10</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>Dijkstra*10</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>PatriciaX5</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>Patricia*X5</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>BlowfishX5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>RijndaelX5</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>ShaX20</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>CRC32X5</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>BitCountX5</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>CjpegX5</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
<td>250</td>
</tr>
<tr>
<td>DjpegX10</td>
<td>1K</td>
<td>1K</td>
<td>1K</td>
<td>1K</td>
<td>1K</td>
</tr>
<tr>
<td>StringSearchX250</td>
<td>50K</td>
<td>50K</td>
<td>50K</td>
<td>50K</td>
<td>50K</td>
</tr>
</tbody>
</table>

#### Pi Accuracy (Decimal Digits)

<table>
<thead>
<tr>
<th>Simulation Platform</th>
<th>Rijndael</th>
<th>Djpeg</th>
<th>All Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>QEMU</td>
<td>18.081s</td>
<td>1.033s</td>
<td>104.099s</td>
</tr>
<tr>
<td>KVM</td>
<td>0.376s</td>
<td>0.148s</td>
<td>5.814s</td>
</tr>
<tr>
<td>Speedup/Slowdown</td>
<td>48.10X</td>
<td>6.96X</td>
<td>17.91X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Simulation Platform</th>
<th>Dijkstra</th>
<th>Rijndael</th>
<th>All Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native</td>
<td>1.185s</td>
<td>0.246s</td>
<td>6.104s</td>
</tr>
<tr>
<td>KVM</td>
<td>0.846s</td>
<td>0.376s</td>
<td>5.814s</td>
</tr>
<tr>
<td>Speedup/Slowdown</td>
<td>1.40X</td>
<td>0.66X</td>
<td>1.05X</td>
</tr>
</tbody>
</table>

Frédéric Pétrot (TIMA Lab)

MPSoC’13

July 18th, 2013
Experiments and Results  

Computation and I/O Speed Comparisons

I/O Speed Comparisons & Slowdown w.r.t Direct Host

Simulation Platform | Best-Case | Worst-Case | Total Time
--- | --- | --- | ---
QEMU | 314.190s | 155.247s | 916.112s
KVM | 35.768s | 385.659s | 708.708s

Speedup/Slowdown | 8.78X | 0.40X | 1.29X

Computation Time | Host | QEMU | Native | KVM
--- | --- | --- | --- | ---
4.127s | 104.099s | 6.104s | 5.814s

Comp. Slowdown | 1X | 25.23X | 1.48X | 1.41X

I/O Time | 3.528s | 916.112s | 545.136s | 708.708s

I/O Slowdown | 1X | 259.69X | 154.53X | 200.90X

Total Time | 7.654s | 1020.211s | 551.240s | 714.522s

Total Slowdown | 1X | 133.28X | 72.02X | 93.35X

Frédéric Pétrot (TIMA Lab)
Simulation Performance with S/W Annotations

<table>
<thead>
<tr>
<th>Buffer Size</th>
<th>Best-Case</th>
<th>Worst-Case</th>
<th>Avg. Slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>StringSearch</td>
<td>BitCount</td>
<td>All Apps</td>
</tr>
<tr>
<td>0</td>
<td>10.38X</td>
<td>1862.38X</td>
<td>84.56X</td>
</tr>
<tr>
<td>64</td>
<td>1.20X</td>
<td>39.69X</td>
<td>2.83X</td>
</tr>
<tr>
<td>256</td>
<td>1.08X</td>
<td>17.00X</td>
<td>1.77X</td>
</tr>
<tr>
<td>1024</td>
<td>1.04X</td>
<td>11.12X</td>
<td>1.49X</td>
</tr>
</tbody>
</table>
Accuracy of S/W Annotations in Native Simulation

Experiments and Results
Performance Estimation Results

<table>
<thead>
<tr>
<th>Error Type</th>
<th>Best-Case</th>
<th>Worst-Case</th>
<th>Average Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rijndael</td>
<td>-0.02%</td>
<td>-16.60%</td>
<td>-3.95%</td>
</tr>
<tr>
<td>StringSearch</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Error Instructions</td>
<td>-0.02%</td>
<td>-16.60%</td>
<td>-3.95%</td>
</tr>
<tr>
<td>Error CPU-Cycles</td>
<td>+0.55%</td>
<td>-14.06%</td>
<td>-2.77%</td>
</tr>
<tr>
<td>Abs. Error Instructions</td>
<td>0.02%</td>
<td>16.60%</td>
<td>5.80%</td>
</tr>
<tr>
<td>Abs. Error CPU-Cycles</td>
<td>0.55%</td>
<td>14.06%</td>
<td>5.36%</td>
</tr>
</tbody>
</table>
Conclusions

- HAV provides a novel way of implementing native simulation
- Memory virtualization solves the conflicting address-spaces issue
- Support for complex MPSoC architectures and legacy software is possible as shared memories can be modeled transparently
- Simulation performance very close to previous native solutions

Limitations

- I/O performance is a bottle-neck, as all I/O requests must trap
- S/W debugging support is currently not available in KVM

Future Directions

- Finding a solution to minimize the Guest-to-Host transitions
- Devising a translation scheme for complex architecture simulation e.g. VLIW Machines
- Improving the annotation technique to increase estimation accuracy


Questions & Answers

Thanks for Your Attention

Questions ?