PQEMU: A Parallel System Emulator Based on QEMU

Jiun-Hung Ding, Po-Chun Chang, Wei-Chung Hsu, Yeh-Ching Chung

Department of C.S., National Tsing Hua University, National Chiao Tung University, Hsinchu, Taiwan, ROC

18 March 2011
Outline

- Motivation
- How to parallelize QEMU
- Experimental results
- Conclusion and future works
Motivation

- QEMU is a fast system emulator which uses DBT technique to achieve emulation efficiency
- While multi-core architectures are getting more important, is current QEMU suitable for emulating multi-core machines?
Computer System emulated by the current QEMU

Emulation thread

CPU
- CPU Idle
- DBT Engine
- Code Cache
- Soft MMU
- Exception/Interrupt Check

Memory
- SDRAM
  - RAM Block
- FLASH
  - RAM Block

I/O
- Interrupt notification
- I/O Device Model

I/O thread

- Keystroke Event
- Screen Update
- Timer Alarm

Exemption/Interrupt Check
- DBT Engine
- Code Cache
- Soft MMU
- Exception/Interrupt Check

CPU
- CPU Idle
- DBT Engine
- Code Cache
- Soft MMU
- Exception/Interrupt Check

Memory
- SDRAM
  - RAM Block
- FLASH
  - RAM Block

I/O
- Interrupt notification
- I/O Device Model

I/O thread

- Keystroke Event
- Screen Update
- Timer Alarm
Serial emulation model

- Guest processor
- Thread on host machine
- Physical core

- QEMU
- Host OS scheduler

- Round-Robin

- P0
- P1
- P2
- P3

- T0

- P0
- P1
- P2
- P3

- Idle
Is QEMU still fast? No!!

1. Emulation slow down!!
2. Incorrect multi-threading application evaluation
Issues

- Current QEMU cannot take advantage of the parallelism available in the underlying hardware
- It may output misleading performance results of multi-threading applications running on the emulated multi-core machine

These issues motivate us to propose a parallel emulation model in QEMU
Computer System in PQEMU

CPU
- CPU Idle
- Parallel DBT Engine
- Code Cache
- Soft MMU
- Exception/Interrupt Check

Memory
- SDRAM
  - RAM Block
- FLASH
  - RAM Block

I/O
- Interrupt notification
- I/O Device Model

Emulation thread
- Emulation thread
- Emulation thread

I/O thread
- Keystroke Event
- Screen Update
- Timer Alarm
Parallel Emulation 1/2

- Guest processor
- Thread on host machine
- Physical core

QEMU

Host OS scheduler

G0 → T0 → P0
G1 → T1 → P1
G2 → T2 → P2
G3 → T3 → P3
Parallel Emulation 2/2

1. Multiple emulation threads can run simultaneously
2. Each virtual CPU is 1-to-1 mapped to each emulation thread
3. We assume # of physical cores is greater and equal than # of guest cores
4. All emulation threads are scheduled by the host OS scheduler
5. Easily enable the parallel emulation by compiling current QEMU with parallel configuration
Outline

- Motivation
- **How to parallelize QEMU?**
- Experimental results
- Conclusion and future works
How to parallelize QEMU?

Trace QEMU source codes to analyze all sharing resources between VCPUs

Propose synchronization models to parallelize the DBT engine of QEMU
QEMU CPU Emulation Flow

- **CPU Idle**
- **Find Fast**
  - **Hit**
  - **Miss**
- **Chain**
- **Execute**
  - **Hit**
  - **SMC**
  - **Interrupt**
- **Check Interrupt**
  - **No**
  - **Yes**
- **Find Slow**
  - **Hit**
  - **Miss**
  - **Done**
  - **Full**
- **Build**
  - **Flush**
  - **Invalidate**
  - **Unchain**
  - **Restore**
- **TBD**
- **TBDA**
- **TBHT**
- **TCG Context**
- **CC**
- **MPD**
Resource dependence graph

- Restore
- Execute
- Flush
- Invalidate
- TCG
- CC
- TBD
- TBDA
- TBHT
- MPD
- Chain
- Unchain
- Build
- Find Slow
Unified Code Cache (UCC) design

- Synchronized
- Dependent, but intrinsically synchronized
- Independent

<table>
<thead>
<tr>
<th>UCC</th>
<th>B</th>
<th>R</th>
<th>C</th>
<th>U</th>
<th>F</th>
<th>I</th>
<th>S</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Build</td>
<td>S</td>
<td>S</td>
<td>D</td>
<td>D</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>D</td>
</tr>
<tr>
<td>Restore</td>
<td>S</td>
<td>S</td>
<td>I</td>
<td>I</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>I</td>
</tr>
<tr>
<td>Chain</td>
<td>D</td>
<td>I</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>I</td>
<td>D</td>
</tr>
<tr>
<td>Unchain</td>
<td>D</td>
<td>I</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>I</td>
<td>D</td>
</tr>
<tr>
<td>Flush</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>Invalidate</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>FindSlow</td>
<td>S</td>
<td>S</td>
<td>I</td>
<td>I</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>I</td>
</tr>
<tr>
<td>Execute</td>
<td>D</td>
<td>I</td>
<td>D</td>
<td>D</td>
<td>S</td>
<td>S</td>
<td>I</td>
<td>I</td>
</tr>
</tbody>
</table>
Synchronization Locks for UCC

Write Lock
- Flush
- Invalidate

Read Lock
- Build
- Chain
- Execute

TB Lock
- Find Slow
- Restore
- Unchain

Chain Lock
Lock Deployment in UCC Design

**CPU Idle**
- **Wait**
  - Read lock E
  - Find Fast (Hit)
  - Lock C
  - Chain
  - Unlock C
  - Execute
  - Check unchain
  - Read unlock E
  - Unlock C
- **Read lock E**
- **Write unlock E**
- **Write lock E**
- **Read unlock E**
- **SMC**
- **Check Interrupt**
  - No Halt?
  - Yes

**Lock B**
- Find Slow
  - Unlock B (Miss)
  - Lock B
  - Build
  - Unlock B
- **Unchain**
  - Try-lock C
  - Lock B
  - Restore
  - Unlock B

**Read lock E**
- **Write unlock E**
- **Write lock E**
- Read unlock E

**Uncompute**
- **Flush**
SCC Model

- Unified Code Cache (UCC) design try to share all resources as possible
- However, this would cause complex implementation and be error-prone
- Hence we consider another design, called Shared Code Cache (SCC), which would duplicate resources for each VCPU to reduce synchronization points
Separate Code Cache (SCC) design

- Separate Code Cache (SCC) design
  - Duplicate all shared resources except MPD
    - MPD is a linked-list array for quick SMC detection

<table>
<thead>
<tr>
<th>SCC</th>
<th>B</th>
<th>R</th>
<th>C</th>
<th>U</th>
<th>F</th>
<th>I</th>
<th>S</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Build</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>Restore</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>Chain</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>Unchain</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>Flush</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>Invalidate</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>Find Slow</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>Execute</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S</td>
<td></td>
</tr>
</tbody>
</table>
Synchronization Locks for SCC

Write Lock
- Invalidate

Read Lock
- Build
- Find Slow
- Restore
- Flush
- Execute
- Chain
- Unchain
Other works

• Support target atomic instruction correctly
• Reduce the communication overhead between I/O thread and emulation threads
Outline

- Motivation
- How to parallelize QEMU?
- Experimental results
- Conclusion and future works
## Experimental Environment

<table>
<thead>
<tr>
<th>Experimental Parameters</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Benchmark</strong></td>
<td>Splash-2 programs for ARM v6 ISA</td>
</tr>
<tr>
<td><strong>Guest OS</strong></td>
<td>Linux 2.6.27</td>
</tr>
<tr>
<td><strong>Guest HW</strong></td>
<td>ARM 11 MPCore architecture (x4 ARM 11 processor)</td>
</tr>
<tr>
<td><strong>Emulator</strong></td>
<td>QEMU 0.12.1 with parallel emulation model (UCC &amp; SCC)</td>
</tr>
<tr>
<td><strong>Host OS</strong></td>
<td>x86_64 Fedora 12 Linux (2.6.31.12)</td>
</tr>
<tr>
<td><strong>Host Machine</strong></td>
<td>Intel Core i7 Quad Cores (4 cores, 8 SMT)</td>
</tr>
</tbody>
</table>
Experimental Result

**Single Core**

ARM11 MPcore X1

**Quad-Core**

ARM11 MPcore X4
Scalable Evaluation

P-QEMU-UCC UCC

P-QEMU-SCC SCC
Comparison: UCC and SCC

- The implementation of UCC is more complex than SCC.
- The overhead of handling “Invalidate” event will be greater for SCC, because it will spend more time to invalidate many duplicated data structures.
- The UCC implementation will incur more cache coherence traffic while the SCC implementation tends to have less coherence traffic.
- The SCC implementation requires much more memory for private code caches.
Conclusion and Future Works

- We have designed and implemented a parallel emulation model based on the current QEMU 0.12.1
- Both UCC and SCC design can improve QEMU to emulate multi-core architecture with efficiently
- Our future works
  - We will use this model to emulate X86 multi-core target with up to 256 processors
  - Propose a VCPU scheduling mechanism for the case which # of emulated cores is larger than # of host cores