COREMU: a Portable and Scalable Parallel Full-system Emulator

Haibo Chen
Parallel Processing Institute
Fudan University
http://ppi.fudan.edu.cn/haibo_chen
Full-System Emulator

• **Useful** tool for multicore
  – Debugging
  – Statistics
  – Trace generation
  – Various instrumentation
  – ...

• **Difficult** to build a **mature emulator on multicore**
  – Hard to be **portable** across platforms
  – Different architectures, OSes, ...
Current Full-System Emulator on CMP systems

- **Poor scalability**
  - Due to the sequential, round-robin emulation of multi/many cores

- **Emulation fidelity**
  - Due to the *coarse emulation granularity*.
  - An example comes soon
Poor Scalability: An Example

- **Word-count** (10MB input)
  - Linear performance slowdown even on a 16 core machine
Emulation Fidelity
- A Simple Example

<table>
<thead>
<tr>
<th>CPU 1</th>
<th>CPU 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>INC [%eax]</td>
<td>INC [%eax]</td>
</tr>
</tbody>
</table>

Data race?

No data race presents if scheduling emulation at basic block / instruction granularity!

Race happens at micro-instruction level
COREMU Design

• Design Principle: Decouples building sequential emulators from parallelizing them

• Cluster mature sequential emulators to build a parallel one.
  – Each emulator efficiently emulates 1 core.

• Use a thin layer for synchronization, communication and scheduling.
  – Decouple the complexity to support parallel emulation
COREMU Architecture

Operating system

Seq. Emulator
Seq. Emulator
Seq. Emulator
Seq. Emulator

Coremu library

Host Operating System

Solve technical problems here
Challenges

• Scalable and portable multiprocessor emulation synchronization

• Efficient and scalable communication

• Scalable code cache management

• Flexible and efficient scheduling
Multiprocessor Synchronization

- This problem is reduced to the problem of atomic instruction emulation
  - Atomic instruction only locks the memory region to access

- On X86
  - Instruction with “lock” prefix
  - Atomically access to a memory location
Naïve Sol#1: Protection with Shadow Locks

Emulated Space

mutex mu1 = ...
mutex mu2 = ...
...
Lock(mu1);
...
Unlock(mu1);

Host Space

mu1
mu2
sh_mu1
sh_mu2

Emulated Code

atomic_cmpxchg (mu, 0, 1);

Lock(mu1);
Unlock(sh_mu1);

Guest Code
Weak Atomicity with Shadow Locks

```c
void lock(mutex_t *mu) {
    for (;;) {
        uint64_t cur = atomic_cmpxchg(mu, 0, 1);
        if (cur == 0)
            break;
        thread_yield();
    }
}

void unlock(mutex_t *mu) {
    *mu = 0;
}
```

Race here causes deadlock!
Naïve Sol#2: Full Locks

• Protect all memory accesses with locks
  – Modify every memory access in sequential emulation

  – Not portable when sequential emulator upgrades

  – Prohibitive performance overhead
    • All memory accesses need locks!
COREMU: use memory transactions

• Observations
  – Few real concurrent updates to atomic variables

• Idea: Use transaction memory
  – Translation of INC is simply as:

```c
Begin_tx();
// Reuse the code generation of sequential emulator
...
End_tx();
```
COREMU: lightweight memory transactions

• Problems
  – No transaction memory support in most commodity processors

• Observations: Instruction has well-defined semantics
  – Stateless: one memory state transition
    • Just redo the failed instruction on conflicts
    • No state recording

  – Different host-target word size
    • Emulate atomic INC64 on 32 bit machine
    • Use well-known CASN algorithm
    • Compare and Swap N words
Emulating Atomic Insts using Txn

**Sequential Emulation**
- **Load**: Reg = [Addr]
- **Inc**: Reg = Reg + 1
- **Store**: [Addr] = Reg

**Parallel Emulation**
- **REDO**:
  - **Load**: old = [Addr]
  - **Inc**: new = old + 1
  - **CAS**(Addr, old, new)
    - IF(success)
      - goto next instr;
    - ELSE
      - goto REDO;
Communications between Cores

• Nonblocking data structures + Real time signals
  – For example, cores can enqueue and dequeue of interrupt simultaneously

• Problem
  – Signal is expensive
  – Excessive signals can cause prohibitive performance overhead
  – Delay of interrupts will cause abnormal behavior

• Optimization
  – Adaptive interrupt control
Adaptive Signal Control

Core1

Sender

Enqueue

Interrupt Queue

#Interrupts Exceed a Threshold?

Threshold Dynamically Adjusted

Notify Receiver to Poll for Interrupt

Receiver

Poll for interrupts and Do Batch Processing

Core2

dequeue
Scalable **Code Cache Management**

- **Code cache**: avoid retranslating existing code

- Scalable code cache scheme
  - Thread-private cache to avoid contentions

- Problems: self/cross-modifying code and code page reused as data page
  - Write to a code page must synchronize with code cache in other emulated cores
    - to invalidate all possible cached translation
Handling SMC/CMC

• Key observation
  – invalidated code pages rarely re-executed later

• Solution
  – Postpone invalidation until re-execution of stale cached code
  – on a code page write
    • all cached translation blocks returns a value to indicate if code eviction needed
    • If re-executed, remove the cached block from each code cache, re-translates and executes it
Scheduling: Lock-holder Preemption

- **vcpu1**
  - Acquire Lock1 Failed
  - Spin-Waiting

- **vcpu2**

- **vcpu3**
  - Holding Lock1

- **vcpu4**
  - Lock Holder
  - Unlock Lock1

- **Physical Core1**

- **Physical Core2**

- **Execution**

- **Not Running**
- **Work Time**
- **Spin Waiting**

- vcpu4 is preempted here
Scheduling: Lock-Aware Preemption

- vcpu1
  - Spin-Waiting
  - Holding Lock1
- vcpu2
  - Not Running
  - Work Time
- vcpu3
  - Not Running
  - Work Time
  - Spin Waiting
- vcpu4
  - Holding Lock1
  - Lock Holder

Execution:
- Only preempt vcpu4 if lock1 was released
- Lock-holder executes till lock is released
Implementation

• Based on QEMU checked out from the git repository on May 4th, 2010
  – 2700 LOC of the COREMU library
  – 2500 LOC modification to QEMU

• On X64 host
  – Support X86, X64 emulation, up to 255 cores
    • xapic has 255 processor emulation limit
  – ARM MPcore architecture, up to 4 cores limited by specification
Data-parallel applications with COREMU

- MapReduce Word-Count 100MB

COREMU has good scalability
Performance for ARM

- Word count 10MB and matrix multiply 800x800
Where can you get COREMU

- COREMU is open sourced
  - On sourceforge: http://sf.net/p/coremu
- Project web site: http://ppi.fudan.edu.cn/coremu
- Mailing list: https://lists.sourceforge.net/lists/listinfo/coremu-list-devel
power management:

processor : 31
vendor_id  : AuthenticAMD
cpu family : 6
model      : 2
model name : QEMU Virtual CPU version 0.12.50
stepping   : 3
cpu MHz    : 4797.855
cache size: 512 KB
fpu        : yes
fpu_exception : yes
cpuid level: 4
wp         : yes
flags      : fpu pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse
            36 clflush mmx fxsr sse sse2 syscall nx lm pni cx16 popcnt hypervisor lahf_lm svm
            abm sse4a
bogomips   : 9599.07
TLB size   : 1024 4K pages
clflush size: 64
cache_alignment : 64
address sizes : 40 bits physical, 48 bits virtual
power management:

debian:~# _
Tasks: 47 total, 22 running
Load average: 2.58 0.55 0.17
Uptime: 00:02:42
Conclusion

• **Decouple** complexity of parallel emulation
  – Clustering multiple sequential emulators

• **Negligible** uniprocessor emulation overhead
  – Within 1%

• **Scale** much better than sequential emulator
  – Significant speedup emulating small scale multi/many cores
Future Work

• Deterministic replay

• Support more architecture

• More debugging and instrumentation support
Thanks!

http://ppi.fudan.edu.cn/coremu
Backup Slides
Uniprocessor Emulation

- **SPECINT2000** - Hardware relative slowdown

Negligible overhead, within 1%
Handling large working set

- Canneal from PARSEC
File System and I/O performance

- Dbench

![Diagram showing throughput (MB/s) vs. number of emulated cores for COREMU, QEMU, and Native.]
Complex work load

- Kernel build: 2.6.33.1 kernel, compact configuration

![Graph showing execution time vs. number of emulated cores for kernel build. The graph compares COREMU and Native approaches. The execution time decreases as the number of emulated cores increases, with COREMU significantly outperforming Native at higher core counts.]
Performance for ARM

- Word count 10MB and matrix multiply 800x800
SMC/CMC

Translated Cache: Core1

... 
Add %eax,%ebx
Mov %ebx, [%ecx]
...

Translated Cache: Core2

... 
Add %eax,%ebx
Mov %ebx, [%ecx]
...

eip