MARSS: Micro Architectural Systems Simulator

Organizers & Presenters

SUNY Binghamton
- Avadh Patel
- Furat Afram
- Brendan Fitzgerald
- Kanad Ghose

Rambus
- Hongzhong Zheng
- James Tringali

marss86.org
MARSS Background

Kanad Ghose
SUNY Binghamton
ghose@cs.binghamton.edu

marss86.org
Simulation Framework for Architects

- Often need to capture microarchitectural details
- Growing need to have:
  - Realistic and accurate models for the memory hierarchy
  - System level component models
- Must be able to simulate execution of wide variety of workloads, not just a handful of benchmarks
  - Need to simulate part of all of the software stack under the application level: OS, libraries, network stack etc.
- High accuracy that can be validated
- High simulation speed is critical
- Easy to use!
MARSS: Microarchitectural and System Simulator

- Meets all of the desired features for current end emerging simulation needs for architects
- Functionally, MARSS is:
  - A full-system simulator: simulates multiple processing cores, coherent caches, on-chip interconnections, DRAM, chipset, I/O + full unmodified binaries of software stack (including OS and libraries)
  - Cycle-accurate: CPU datapath (in-order and out-of-order), memory hierarchy, on-chip interconnections, system bus, DRAM and controllers can be simulated cycle-by-cycle. Functional models exist for chipset, disk, NIC etc.
  - A simulation framework that switches seamlessly between cycle-accurate simulation and fast emulation on the native hardware.
MARSS: Accuracy and Speed

- Results for X86 platforms/cores can be validated against existing hardware
  - Almost impossible to do for most of the simulators we use today that use defunct or non-pervasive ISAs
  - Toolchain support for many of these ISAs is poor
- MARSS with X86 core models is fast
  - Hand crafted datapath models that exploit existing caches well in simulation, use media extensions where they help, employ tuned assist functions...
  - Switches seamlessly to fast emulation on the native X86 platform when cycle-accurate simulation is not needed
- Permits fast deployment and use
  - Unmodified binaries can be run, no cross-compilation, library incompatibilities
  - No need to modify system-level components (such as hypervisors)
  - Can use existing and widely used toolchain (compilers, debuggers, profilers etc.)
MARSS Development History – 2005-2009

- X86 Core Design: started by Matt Yourst in 2005
  - Extensive reverse engineering using a Transmeta SBC to validate X86 to uop decomposition
  - Requires modified Xen hypervisor, modified binaries for code to be simulated
  - *ISPASS 2007* paper has details

- SMT extensions designed by Hui Zeng and Matt Yourst in 2007

- Initial Multicore version with coherent caches (called MPTLsim) was designed by Hui Zeng
  - Had many of the inherent limitations of PTLsim
  - *DAC 2010* paper has details
MARSS Development History – 2010-2012

- **Avadh Patel**, assisted by **Furat Afram** addressed the limitations of MPTLsim to realize the MARSS prototype by:
  - Switching to a QEMU-based environment, completely redesigning simulation framework
  - Extensively modifying the PTLsim OOO core model to correct known issues
  - Completing the ISA support including MMX and by adding many additional features
  - Adding Modularization and User-Friendly Configuration support
  - **DAC 2011** paper has details
- MARSSx86 was deployed publicly at [www.marss86.org](http://www.marss86.org) on February 11, 2010.
MARSS Development – Community Support

- Significant extensions and bug fixes were made to MARSS after the initial deployment:
  - Adnan Khaleel of CRAY/SGI provided extensive feedback and bug fixes on the early version
  - Paul Rosenfeld of UMD ported DRAMSim2 to MARSS
  - Hongzhong Zheng and James Tringali of Rambus performed bug fixes and extensive validation to make MARSS more robust and friendlier for users
  - The McPAT power and area estimation tool was integrated with MARSS by Tyler Stachecki at Binghamton – an initial version was available in May 2012.
  - Last, but not the least, the MARSS user community
This Tutorial

- Will describe the design philosophy of MARSS
- Present the internals of MARSS and relevant source code
- Will show you how to set up and use MARSS
- Present important extensions and case studies using MARSS
- Incorporate demos extensively

So, on to the tutorial…..
MARSSS Introduction

Avadh Patel
SUNY Binghamton
apatel@cs.binghamton.edu

marss86.org
Architectural Simulation Challenges

- **Large-scale Computing:**
  - **Multicore** servers: need to simulate wide variety of workloads with large footprints
  - Need to simulate complex memory system architectures, coherent caches
  - Need to simulate IO bound software applications with impact of systems software components
  - Need to evaluate energy-performance tradeoffs

- **Mobile/embedded Space needs:**
  - Hardware-Software **co-design** support
  - Support SoC with diverse **modules**
  - Support energy-performance tradeoff studies
Ideal Architecture Tools in Our Mind

- State of art on-chip and off-chip component models
- Full software stack
- Client-server Simulations

- Complete
- Fast
- Open

- Simulation Speed
- Minimum setup time
- Rapid development

- Open-sourced
- Collaborative Community
- Transparent to hardware

marss86.org
QEMU – Quick Emulator

- **Complete**: Full system emulation of
  - x86, ARM, etc.
  - Unmodified software stack
- **Fast** Emulation
  - Modularized
- **Open**-source
  - Active community and growing
MARSS - Micro-ARChitectural System Simulator

- Integration of QEMU and modified PTLsim
- A full-system simulation framework with
  - Detailed simulation of CPUs, Caches and Memory
  - Emulation/simulation of IO devices

![MARSS Diagram]

- Guest Applications
- Guest OS Kernel
- Emulation Engine
- Simulation Engine
- Emulated Devices
- Disk, NIC, GPU, USB
- Host Linux Kernel
- Host x86-64 Hardware
MARSS – QEMU + Modified PTLsim

- Framework includes
  - Cycle accurate simulation models for CPUs, cache, interconnect, DRAM controller etc.
  - Emulation models for CPU, Disk, NIC, etc. from QEMU
- Simulate unmodified software stack
- Runs on top of unmodified Linux Kernel and x86-64 hardware
MARSS and QEMU

- MARSS is hosted within the QEMU environment
- Extensively use emulation models of QEMU
  - Complex opcode helpers (for instructions like fxsave, fxrstor, syscall, sysret for the X86 ISA) within QEMU are used to avoid complexities within the cycle-accurate simulation
- Leverages the extensive effort of the large QEMU development community
Key Features

- Full System Cycle-Accurate Simulation of
  - Multicore x86 systems
  - Multiple core and cache models
  - Coherent caches and Network-on-chip
- Boot and Simulate unmodified OS/libraries/Applications
  - Supports unmodified pthreads
- Plugin interface for custom/proprietary modules
  - DRAMSim2, etc.
Advanced Features

- Checkpoint, Fast-Forward and Simpoint Support
  - Enable simulation of represented snapshots of applications
- Power Estimation using *McPAT* tool from HP
- Hierarchical dynamic statistics framework
  - captures separate user and kernel statistics
- Batch simulation runs using support scripts
- Built-in optimized library for basic architectural modules
  - Queues, Associative Arrays, Bit-vectors etc.
Performance & Accuracy Summary

Average simulation speed of SPEC CPU2006 benchmarks

SPEC 2006

Parsec 2.1
Project Goals

- Full System Simulation Framework
- Client-server Simulations
- Fast and scalable Simulation speeds
- Modularized plugin based framework
- Unmodified OS & applications
- Open sourced under GPL-v2
- Collaboration from Academic and Industry

marss86.org
External Tools

- DRAMSim2
  - https://wiki.umd.edu/DRAMSim2/
- HybridSim
  - Intel
- PCI_SSD
  - https://github.com/jimstevens2001/PCI_SSD
MARSS Performance and Accuracy

Hongzhong Zheng
Rambus Labs
hzheng@rambus.com
The Most Important Features For a Simulation Tool In Our Mind

- Can evaluate real application on main stream computer architecture in full system environment
- Can produce solid and repeatable simulation results
- Has reasonable accuracy against real hardware
- Is fast enough to explore design space in reasonable time
Benchmarks And Applications Used: Summary

- SPEC CPU 2006 INT/FP
  - CPU intensive benchmarks
- PARSEC suite
  - Emerging workloads for shared-memory parallel programs
- LMBench and STREAM
  - Computing system latency and bandwidth benchmarks
- Pgbench
  - PostgreSQL benchmarks for SQL database
- SPECJbb
  - JVM based benchmarks
- CloudSuite (Working in progress)
  - Big data applications: Map-reduced workloads, Web search, Web serving and Media streaming etc.
Solid Simulation Results from MARSS
Full System Simulation

- Small simulation result variability for Parsec suite
  - IPC variation: Max/Min IPC from 10 runs for same configuration
- Parallel workloads
  - Eight active benchmark threads and native input
- Eight-core OOO CPU, 3-level MESI coherent cache and simple main memory model
- Fast-forward 50 billion instructions from Parsec benchmark region of interest, and Simulate 500 million instructions

<table>
<thead>
<tr>
<th>Workloads</th>
<th>IPC variation</th>
</tr>
</thead>
<tbody>
<tr>
<td>blackscholes</td>
<td>0.0%</td>
</tr>
<tr>
<td>canneal</td>
<td>0.0%</td>
</tr>
<tr>
<td>swaptions</td>
<td>0.1%</td>
</tr>
<tr>
<td>streamcluster</td>
<td>1.3%</td>
</tr>
<tr>
<td>facesim</td>
<td>0.1%</td>
</tr>
<tr>
<td>raytrace</td>
<td>0.1%</td>
</tr>
<tr>
<td>fluidanimate</td>
<td>0.1%</td>
</tr>
<tr>
<td>dedup</td>
<td>1.7%</td>
</tr>
<tr>
<td>bodytrack</td>
<td>3.4%</td>
</tr>
<tr>
<td>vips</td>
<td>1.8%</td>
</tr>
<tr>
<td>ferret</td>
<td>0.3%</td>
</tr>
<tr>
<td>freqmine</td>
<td>0.0%</td>
</tr>
<tr>
<td>x264</td>
<td>1.2%</td>
</tr>
<tr>
<td>AVG</td>
<td>0.78%</td>
</tr>
<tr>
<td>MAX</td>
<td>3.40%</td>
</tr>
</tbody>
</table>
Solid Simulation Results from MARSS
Full System Simulation

- Small simulation result variability for SPEC CPU2006 suite
  - IPC variation: Max/Min IPC from 10 runs for same configuration
  - Multi-program workloads
    - Eight active instance and reference input
  - Eight-core OOO CPU, 3-level MESI coherent cache and simple main memory model
  - Simulate 500 million instructions from single Simpoint

<table>
<thead>
<tr>
<th>Workloads</th>
<th>IPC variation</th>
<th>Workloads</th>
<th>IPC variation</th>
</tr>
</thead>
<tbody>
<tr>
<td>astar</td>
<td>0.2%</td>
<td>GemsFDTD</td>
<td>0.2%</td>
</tr>
<tr>
<td>bzip2</td>
<td>0.0%</td>
<td>gromacs</td>
<td>0.1%</td>
</tr>
<tr>
<td>gcc</td>
<td>0.1%</td>
<td>ibm</td>
<td>0.1%</td>
</tr>
<tr>
<td>gobmk</td>
<td>0.0%</td>
<td>leslie3d</td>
<td>0.4%</td>
</tr>
<tr>
<td>hmmer</td>
<td>0.0%</td>
<td>milc</td>
<td>0.2%</td>
</tr>
<tr>
<td>mcf</td>
<td>0.2%</td>
<td>namd</td>
<td>0.0%</td>
</tr>
<tr>
<td>sjeng</td>
<td>0.0%</td>
<td>soplex</td>
<td>0.1%</td>
</tr>
<tr>
<td>bwaves</td>
<td>0.0%</td>
<td>tonto</td>
<td>0.1%</td>
</tr>
<tr>
<td>cactusADM</td>
<td>0.2%</td>
<td>zeusmp</td>
<td>0.0%</td>
</tr>
<tr>
<td>AVG</td>
<td>0.1%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAX</td>
<td>0.4%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
MARSS Hardware Correlation

- Target system: Use kernel PCM API to gather IPC etc. stats

- Simulated system: Use MARSS to gather IPC etc. stats

SPEC CPU 2006
Perf (PCM)
Linux Kernel 2.6
Target x86-64 Hardware

SPEC CPU 2006
Guest Linux Kernel 2.6

- Emulation Engine
- Simulation Engine
- Emulated Devices: Disk, NIC, GPU, USB

Host Linux Kernel
Host x86-64 Hardware

CPUs
MEM

Sanity Checking Model

correlation result
Sanity Checking Methodology

- Three steps for hardware correlation
  - To minimize impact of microarchitecture unknown

Step 1: CPU function unit latency correlation

Step 2: Memory hierarchy latency and memory bandwidth correlation
  - LMBench and performance counter

Step 3: Application IPC correlation
  - Use application Simpoints for simulation results

<table>
<thead>
<tr>
<th>Target computing system</th>
<th>Simulated computing system</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS</td>
<td>OS</td>
</tr>
<tr>
<td>Linux 2.6.39-1 64bit</td>
<td>Linux 2.6.31-4 64bit</td>
</tr>
<tr>
<td>Processor</td>
<td>Processor</td>
</tr>
<tr>
<td>Intel Xeon E5620 @2.4GHz</td>
<td>4-core, out of order processor, 2.4GHz</td>
</tr>
<tr>
<td></td>
<td>Fetch/Commit width: 4, Issue width: 5</td>
</tr>
<tr>
<td></td>
<td>LD/ST FU: 1/1, ITLB/DTLB: 32/32</td>
</tr>
<tr>
<td></td>
<td>frontend_stages: 4, max_branch_in_flight: 24</td>
</tr>
<tr>
<td>cache</td>
<td>cache</td>
</tr>
<tr>
<td>IL1:32KB, 4-way; DL1:32KB, 8-way</td>
<td>IL1:32KB/core, 4-way, 2cc; DL1:32KB/core, 8-way, 4cc</td>
</tr>
<tr>
<td>L2:256KB,8-way; L3: 12MB,16-way</td>
<td>L2:256KB/core, 8-way, 6cc</td>
</tr>
<tr>
<td>Main memory</td>
<td>Main memory</td>
</tr>
<tr>
<td>3 channel, DDR3-1333, 10-10-10</td>
<td>Fix latency per access: 55ns</td>
</tr>
<tr>
<td>1DIMM/CH, 2Rank/DIMM</td>
<td>(MC overhead+DRAM access+conflicts)</td>
</tr>
<tr>
<td>cache line interleaving</td>
<td>64 banks, cache line interleaving; Queue: 128</td>
</tr>
</tbody>
</table>
MARSS Memory Hierarchy Latency Correlation

<table>
<thead>
<tr>
<th>memory hierarchy latency</th>
<th>Target system (round trip latency, ns)</th>
<th>Simulated system (round trip latency, ns)</th>
<th>difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>1.668</td>
<td>1.686</td>
<td>1%</td>
</tr>
<tr>
<td>L2</td>
<td>4.178</td>
<td>4.184</td>
<td>0%</td>
</tr>
<tr>
<td>L3</td>
<td>18.19</td>
<td>18.182</td>
<td>0%</td>
</tr>
<tr>
<td>Main memory</td>
<td>75.63</td>
<td>76.06</td>
<td>1%</td>
</tr>
</tbody>
</table>

- Memory hierarchy latency correlate very well
  - ‘lat_mem’ of LMBench suite
MARSS Main Memory Bandwidth Correlation

<table>
<thead>
<tr>
<th>Single instance</th>
<th>Target system</th>
<th>Simulated system</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single instance</td>
<td>6823 MB/s</td>
<td>8717 MB/s</td>
</tr>
<tr>
<td>Four instance</td>
<td>14572 MB/s</td>
<td>19294 MB/s</td>
</tr>
</tbody>
</table>

Notes: statistics from ‘bw_mem’ of LMBench

- Main memory bandwidth not correlate well
  - Due to LLC misses traffic difference from undisclosed micro architecture
  - ‘bw_mem’ of LMBench suite
  - Four core system + dual channel DDR3-1333 memory system
### CPU Function Unit Latency Correlation

<table>
<thead>
<tr>
<th>INT function unit</th>
<th>Target system (ns)</th>
<th>Simulated system (ns)</th>
<th>difference</th>
<th>FP function unit</th>
<th>Target system (ns)</th>
<th>Simulated system (ns)</th>
<th>difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer bit</td>
<td>0.42</td>
<td>0.34</td>
<td>19%</td>
<td>float add</td>
<td>1.25</td>
<td>1.22</td>
<td>0.02%</td>
</tr>
<tr>
<td>integer add</td>
<td>0.21</td>
<td>0.17</td>
<td>19%</td>
<td>float mul</td>
<td>1.67</td>
<td>1.65</td>
<td>0.01%</td>
</tr>
<tr>
<td>integer mul</td>
<td>0.13</td>
<td>0.14</td>
<td>-8%</td>
<td>float div</td>
<td>6.21</td>
<td>6.19</td>
<td>0.03%</td>
</tr>
<tr>
<td>integer div</td>
<td>10.03</td>
<td>10.41</td>
<td>-4%</td>
<td>double add</td>
<td>1.25</td>
<td>1.22</td>
<td>0.02%</td>
</tr>
<tr>
<td>integer mod</td>
<td>9.57</td>
<td>10.55</td>
<td>-10%</td>
<td>double mul</td>
<td>2.08</td>
<td>2.48</td>
<td>-19%</td>
</tr>
<tr>
<td>int64 bit</td>
<td>0.42</td>
<td>0.41</td>
<td>0.2%</td>
<td>double div</td>
<td>9.55</td>
<td>9.1</td>
<td>0.04%</td>
</tr>
<tr>
<td>uint64 add</td>
<td>0.21</td>
<td>0.21</td>
<td>0%</td>
<td>float bogomflops</td>
<td>5.84</td>
<td>11.15</td>
<td>-91%</td>
</tr>
<tr>
<td>int64 mul</td>
<td>0.13</td>
<td>0.13</td>
<td>0%</td>
<td>double bogomflops</td>
<td>9.18</td>
<td>14.79</td>
<td>-61%</td>
</tr>
<tr>
<td>int64 div</td>
<td>18.04</td>
<td>19.58</td>
<td>-8%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>int64 mod</td>
<td>18.97</td>
<td>19.64</td>
<td>-3%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **CPU function unit latency difference due to undisclosed microarchitecture detail**
  - E.g. decode optimization, function unit and uop optimization
  - Function unit latency measured by ‘lat_ops’ of LMBench suite
### SPEC CPU2006 IPC Correlation

<table>
<thead>
<tr>
<th>SPEC CPU2006 INT IPC correlation</th>
<th>SPEC CPU2006 FP IPC correlation</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip2 -28%</td>
<td>bwaves -35%</td>
</tr>
<tr>
<td>gcc -13%</td>
<td>gamess N/A</td>
</tr>
<tr>
<td>MCF 13%</td>
<td>milc -16%</td>
</tr>
<tr>
<td>gobmk -7%</td>
<td>zeusmp 23%</td>
</tr>
<tr>
<td>hmmmer -23%</td>
<td>gromacs -11%</td>
</tr>
<tr>
<td>sjeng -10%</td>
<td>cactusADM 28%</td>
</tr>
<tr>
<td>libquantum N/A</td>
<td>leslie3d -24%</td>
</tr>
<tr>
<td>H264 N/A</td>
<td>namd -3%</td>
</tr>
<tr>
<td>omnetpp -15%</td>
<td>dealll N/A</td>
</tr>
<tr>
<td>astar 15%</td>
<td>soplex N/A</td>
</tr>
<tr>
<td>xalancbmk N/A</td>
<td>povray -26%</td>
</tr>
<tr>
<td>(calculix -60%</td>
<td>GemsFDTD -35%</td>
</tr>
<tr>
<td>tonto -15%</td>
<td>lbm 23%</td>
</tr>
<tr>
<td>wrf 12%</td>
<td>sphinx3 N/A</td>
</tr>
<tr>
<td>calculix -60%</td>
<td>GemsFDTD -35%</td>
</tr>
<tr>
<td>lbm 23%</td>
<td>sphinx3 N/A</td>
</tr>
</tbody>
</table>

- **Reasonable IPC difference due to undisclosed microarchitecture detail**
  - IPC of target system from Performance counter
  - IPC of simulated system from weighted multiple Simpoints IPC results
  - Due to uncorrelated CPU function unit latency, more uops decoded per instruction, etc.
MARSS Simulation Speed

- Scalable simulation speed with number of cores to simulate
  - $> 160K$ instructions per second on average for single core configuration among all SPEC CPU2006 benchmarks
  - $< 8$ min to simulate 100 million instructions
  - Host machine: Intel Xeon CPU E5520 @ 2.27GHz, 8MB LLC

**Simulated computing system**

| OS | Linux 2.6.31-4 64bit |
| Processor | 1/4/8/32-core, out of order processor, 2.4GHz |
| ROB: | 128; LDQ: 48; STQ: 32; FetchQ: 48; IQ: 36 |
| Fetch/Commit width: | 4, Issue width: 5 |
| LD/ST FU: | 1/1, ITLB/DTLB: 32/32 |
| frontend_stages: | 4, max_branch_in_flight: 24 |
| Cache | IL1:32KB/core, 4-way, 2cc; DL1:32KB/core, 8-way, 4cc |
| L2:256KB/core, 8-way, 6cc; |
| L3: 2MB/core, shared, 16-way, 27cc |
| Queue: | 128; cache coherence: MESI |

**Average simulation speed of SPEC CPU2006 benchmarks**

- Thousand simulated instructions per second

- Workloads: astar, hmmer, lbm, mcf, omnetpp, calculix, gcc, GemsFDTD, gobmk, gromacs, bwaves, bzip2, leslie3d, cactusADM, milc, namd, sjeng, soplex, tonto, zeusmp

- Single core
- Four core
- Eight core
- Thirty-two core
Parallel MARSS Simulation to Speedup Experiments

- Serialized simulation of all cores for each cycle
  - Cycle through all cores for each cycle

- Parallel simulation of each core by ‘pthread’
  - Synchronize machine status each cycle

---

Flowchart:

- **Simulate one cycle for a CPU core**
  - **Simulate events (MEM, I/O, etc.)**
  - **Cycle start**
  - **Simulate one cycle for a CPU core**
  - **Simulate events (MEM, I/O, etc.)**
  - **Cycle start**
  - **Simulate one cycle for a CPU core**
  - **Simulate one cycle for a CPU core**
  - **Synchronize machine status**
  - **Cycle end**

---

- **Cycle through all cores?**
  - **No**
  - **Simulate one cycle for a CPU core**
  - **Simulate events (MEM, I/O, etc.)**
  - **Cycle start**
  - **Simulate one cycle for a CPU core**
  - **Simulate events (MEM, I/O, etc.)**
  - **Cycle start**
  - **Simulate one cycle for a CPU core**
  - **Simulate one cycle for a CPU core**
  - **Synchronize machine status**
  - **Cycle end**

---

**Independent instructions of all cores can be simulated at same time**
Parallel MARSS Simulation Speed

- Significant speedup by multi-threaded cycle accurate MARSS simulation
  - > x2 simulation speed improvement on average
  - < 0.4% IPC variance among different pthread configurations

- Eight active benchmark threads and native input
- Eight-core OOO CPU, 3-level MESI coherent cache and simple main memory model
- Fast-forward 50 billion instructions from Parsec benchmark region of interest, and Simulate 500 million instructions
Summary

- MARSS is robust, fast and cycle accurate architecture full system simulation framework
- MARSS can be used to evaluate a broad range of unmodified x86 based applications
- MARSS is reasonably accurate against real hardware
MICRO-2012

Roadmap

James Tringali
Rambus Labs
jamestr@rambus.com
Current state

- Many options available for simulation, emulation and benchmarking.
- Few are well supported.
  - Documentation
  - Continued development.
- At time zero ....
  - Survey existing tools
  - happy = eval (features, buggy-ness, support)
  - if (not happy) { Create a new one! }

- Energy focused on results not correlation and robustness
- Spotty industry support for correlating architectural models against state of the art implementations
Ideal state

- Highly capable embedded systems
  - Capable of running an OS

- Scale-out architectures for service providers
  - High degree of parallelism

- Fungible infrastructure capable of spanning both classes

- Well documented and modular infrastructure

L. A. Barroso and U. Hölzle, “The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines"
MARSS closes the gap

- System level simulation using VMM
- Software models are complete enough to boot modern OS
- Speed and accuracy by tightly coupling functional and cycle driven models
- Only publically available implementation of cycle “approximate” x86 core
- Completely open source implementation
  - QEMU substrate
  - PTLSim x86 model
Roadmap

PHASE 1:
NURTURE

PHASE 2:
EVANGELIZE

+ Client-Server
+ ARM
+ GPU
+ Disksim
+ NIC
+ Memory
+ Logic

Enables SOC
+ ... 

Mobile clients
Data centers
HPC
Novel Arch

Distributed systems

Distributed computation

ISCA 2012
MICRO 2012

RMBS or others
SUNY

HPC
Novel Arch

Mobile clients
Data centers

HPC
Novel Arch

SUNY
RMBS or others

Distributed systems

Enables SOC

+ Memory + Logic

+ GPU

+ Disksim

Distributed computation
Join us!
MARSS Architecture

Avadh Patel
SUNY Binghamton
apatel@cs.binghamton.edu
MARSS – Major Components

- MARSS framework includes
  - Emulation models for CPU, Disk, NIC, etc. from QEMU
  - Cycle accurate simulation model for CPUs, cache, interconnect, DRAM controller etc.
  - Runs on top of unmodified Linux Kernel and x86-64 hardware
MARSS – Execution Flow

- Seamless switching between emulation and simulation engines
- Emulation and Simulation engines share:
  - Guest CPU contexts
  - Guest Physical memory
  - Guest I/O devices
MARSS – Modularized Simulation Components

Components of Simulation Engine

- **Machine Manager**
  - Clock
  - Modules
  - Builder

- **Builder**
  - OoO
  - Coherent Cache
  - Bus
  - DRAM Cont.

**Components of Simulation Engine**

- **Cores**
  - OoO
  - Atom
  - DRAM Cont.
  - Directory

- **NoC**
  - Bus
  - Switch
  - Coherent
  - Write-Through
  - Write-Back

- **Caches**
  - Coherent
  - Bus
  - Switch

- **Plugins**
  - Directory
  - DRAMsim2 etc.

- **x86 Decode**
  - Decoder
  - uOp

- **Statistics**
  - statsBuilder
  - stats

- **Support Libraries**
  - superstl
  - logic
  - YAML
  - gTest
  - x86 Decode

- **Emulation Engine**
  - CPU Context
  - Guest Memory
  - IO Devices
  - GPU
  - NIC

- **Build machine from components**

Can be extended for timing simulation

marss86.org
Simulation Components – CPU Context

- Maintains per CPU registers, flags, PC address, etc.
- Shared between Emulation & Simulation engines
- File: qemu/target-i386/cpu.h

```c
typedef struct CPUX86State {
    /* standard registers */
    target_ulong regs[CPU_NB_REGS];
    target_ulong eip;
    target_ulong eflags;
    /* emulator internal eflags handling */
    target_ulong cc_src;
    target_ulong cc_dst;
    /* segments */
    SegmentCache segs[6]; /* selector values */
    /* Control Registers */
    target_ulong cr[8];
    /* FPU Status Registers */
    unsigned int fpstt; /* top of stack index */
    uint16_t fpus;
    uint16_t fpuc;
    uint8_t fptags[8]; /* 0 = valid, 1 = empty */
    FPReg fpregs[8];
    ...
};
```
MARSS – Hybrid Simulation Methodology

- Discrete-event simulation
  - Modules with fewer per cycle simulation
  - Examples: Caches, Simple DRAM Controllers, IO Devices
  - Use ‘Signal’ to add event:
    - `marss_add_event(..)`

- Cycle-by-cycle simulation
  - Modules with detailed per cycle activities
  - Example: Core Models, Detailed DRAM Controllers
  - Register Callback Interface:
    - `marss_register_per_cycle_event(..)`

```c
/* Execute cycle callbacks */
for each (clock-callback) {
  (*clock-callback());
}
cycle++;

/* Execute event Signals */
while (signal->cycle == cycle) {
  signal->emit();
  signal = next_signal();
}
```
MARSS – Functional Simulation

- In the simulation mode, functional correctness is provided by 'simulation engine'
  - Validated by QEMU test suite
- Each x86 opcode is divided into one or more micro-ops (uop)
- Currently 129 uops are implemented in ‘ptlsim/x86/uopimpl.cpp’

**MARSS uops**

- **x86 Opcode**
- **add rax,(rcx)**
- **ld tmp0,(rcx)**
- **add rax, tmp0**

**MARSS uops**

- **Generate Result**
- **Update CPU Context**
MARSS – Timing Simulation

- Cycle-by-cycle execution of each stage

- Generate Cache Access and wait for “Complete” Event

- Detailed Memory hierarchy timing simulation with queuing delays using Events

- Load

- Store

- Detailed Memory hierarchy timing simulation with queuing delays using Events
MARSS – Simulate everything except micro-code

- User and kernel activities are simulated in detail
- Exceptions/Interrupts/System calls are simulated
- Micro-code for CPU context setup emulated by QEMU API functions
  - No timing simulation for this
  - File: qemu/target-i386/helper.c
Simulate Everything – Example: Page Fault

- On TLB miss
  - Detailed simulation of 4 level page walking
  - Cache access for each level of page table entry
- On a page fault (no valid page table entry)
  - Set OS Page Fault Handler Context
  - Simulate OS page fault handler instructions
Maintaining Guest Machine’s Clock

- QEMU (Emulation) does not maintain ‘clock’
  - Returns host machine’s clock value on `rdtsc`
  - File: `qemu/qemu-timer.{h,c}`
- Simulation engine keeps track of number of executed cycles
  - Global variable `sim_cycle`
  - Save host clock value as `offset` on switch from emulation mode
  - Return `sim_cycle + offset` on `rdtsc`
MARSS Core Models

Avadh Patel
SUNY Binghamton
apatel@cs.binghamton.edu

marss86.org
Core Module

- Current support
  - Out-of-order core
  - Atom-like in-order core
- Supports
  - Multi-core
  - Heterogeneous cores
  - SMT
- Files: ptlsim/core/*
Core Module View From 5000 Miles

- Core Module independent x86 decoder
- CPU context contains the architectural state
- Core Module maintains the micro-architectural state
- Core Modules responsible of both timing and function simulation

---

**x86 Decoder**

- Basic Block buffer
- x86 to MARSS uop translator
- VM physical memory

**Core Module**

- independent x86 decoder
- CPU context contains the architectural state
- Core Module maintains the micro-architectural state
- Core Modules responsible of both timing and function simulation

---

**CPU context**

- Fetch
- Decode
- Rename
- Dispatch
- issue
- Execute
- Complete
- Broadcast
- Write back
- Commit

---

**Out-of-order pipeline**

- Load
- Store

---

**Update CPU context**

---

marss86.org
MARSS Micro-Operation (uop)

- Models break down of the x86 instruction into RISC-like MARSS uops
- MARSS uop has 3 source registers and 1 destination register
- MARSS uop can be
  - Register operation
  - Memory operation (load/store)
  - Control operation
MARSS Micro-Ops Format

- **Opcode:**
  - MARSS opcode
  - All opcodes are defined in ptsim/x86/ptlhwdem.h
  - Every opcode has to be assigned to function unit/group
- **Destination:** destination register or memory location
- **Sources:** up to 3 source operand, RC used only for store operation
- **Size:** 8, 16, 32 or 64 bit operation
- **Optional arguments:**
  - Immediate values
  - Flags
x86 to uop Example (PUSH cs)

```c
if operand size = 16
    then temp <-- SRC
    else temp <-- zeroextend(src)
fi
if operand size = 64
    then
        RSP <-- RSP - 8
        Memory[RSP] <-- temp
    else
        RSP <-- RSP - 2
        Memory[RSP] <-- temp
fi
```

TransOp(OP_ld, r, REG_ctx, REG_imm, REG_zero, size,
        offsetof_t(Context, segs[seg_reg].selector));
TransOp(OP_st, REG_mem, REG_rsp, REG_imm, r, sizeshift, -size);
TransOp(OP_sub, REG_rsp, REG_rsp, REG_imm, REG_zero, 3, size);

Intel® 64 and IA-32 Architectures Software Developer’s Manual
4-426 Vol. 2B

MARSS
x86 Decoder

- x86 ISA compatible
- Break down every x86 instruction into Micro-operations (uops)
- Buffer decoded uops to speedup simulation
  - Software optimization
Out-of-Order Pipeline

- Static front end delay for decoding
  - Configure using the `front_end_delay` variable
- Configurable stage widths
- Pipeline stages communicate using software

**ROB entry**

CPU context

x86 to uop Decode

Fetch → Decode → Rename → Dispatch → issue → Execute → Complete → Transfer → Write back → Commit

ROB Entry

marss86.org
Flow Chart of Instruction In OOO Pipeline

- **Fetch**
  - Timing: icache latency

- **Decode**
  - Timing: front_end_delay

- **Rename**
  - Timing: rob full, no physical register available

- **Issue**
  - Timing: availability of operands & Function unit

- **Execution**
  - Timing: Function unit latency / cache latency

- **Commit**
  - Timing information: head of the ROB, all the uops that belong to the same x86 instruction are ready to commit

---

- **Operands available**
  - yes

- **PC**

- **Fetch**

- **Decode**
  - Uops buffer hit/miss

- **Create new ROB entry**
  - hit
  - Translate basic block

- **Rename operands**

- **Operands available**
  - no

- **Execute operation**
  - Probe the cache

- **Update physical registers**

- **Update rename table and CPU context**

- **Free resources**

- **Timing information**: head of the ROB, all the uops that belong to the same x86 instruction are ready to commit

---

**marss86.org**
Multi-Threaded Out-of-Order Structures

*Per Core Structures
- Shared among core threads

*Per Thread Structures

Pipeline Stage Functions

### OooCore
- ThreadContext threads[];
- IssueQueue issueq;
- PhysicalRegisterFiles rf;
- W32 fu_map;
- bool fetch();
- void frontend();
- int dispatch();
- void rename();
- int issue();
- int complete();
- int transfer();
- int writeback();
- int commit();

### ThreadContext
- Queue<FetchBufferEntry, N> fetchq;
- Queue<ReorderBufferEntry, N> ROB;
- Queue<LoadStoreQueueEntry, N> LSQ;
- RegisterRenameTable specrrt;
- RegisterRenameTable commitrrt;
- CPUContext ctx;

### FetchBufferEntry
- TransOp uop;
- uopimpl_func_t syntop;

### ReorderBufferEntry
- FetchBufferEntry uop;
- int cycles_left;
- StateList *current_state_list;

### IssueQueue
- int uopids;
- int tags[];
- bitvec valid;

### PhysicalRegisterFile
- array<PhysicalRegister, N> registers;

### LoadStoreQueueEntry
- W64 address;
- W64 data;
- ReorderBufferEntry *rob;

### RegisterRenameTable
- array<PhysicalRegister*> regs;
Reorder Buffer

- ROB tracks all the uops in flight
- ROB entry state describes the uop progress in the pipeline
- ReorderBufferEntry structure contains
  - uop information
  - ROB state
  - Pointers to the physical registers
  - Pointer to the LSQ entry (memory operations only)

ReorderBufferEntry

FetchBufferEntry uop;
W16s cycles_left;
StateList *current_state_list;
PhysicalRegister* physreg;
PhysicalRegister* operands[MAX_OPERANDS];
LoadStoreQueueEntry* lsq;
W16s idx;
W16s forward_cycle;
W16s ifrqslot;
W16s iqslot;
W16s executable_on_cluster_mask;
W8s cluster;
W8 coreid;
W8 threadid;
...

marss86.org
### ROB Entry State Transition

- Each pipeline-stage maintains a list of ROB entries in same state.
- Timing delay is captured by ROB entry state transition.
- ROB entry state transition follows the path of instruction in pipeline.

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROB free</td>
<td>ROB entry allocated in rename stage</td>
</tr>
<tr>
<td>ROB Frontend</td>
<td>Frontend delay</td>
</tr>
<tr>
<td>Rob ready to dispatch</td>
<td>Dispatch stage</td>
</tr>
<tr>
<td>ROB ready to issue</td>
<td>Issue stage</td>
</tr>
<tr>
<td>ROB ready to load</td>
<td>Complete stage</td>
</tr>
<tr>
<td>ROB issued</td>
<td>Forward stage</td>
</tr>
<tr>
<td>ROB completed</td>
<td>Write back stage</td>
</tr>
<tr>
<td>ROB ready to writeback</td>
<td>Commit stage</td>
</tr>
</tbody>
</table>

Diagram:
- ROB Frontend
- Rob ready to dispatch
- ROB ready to store
- ROB ready to issue
- ROB ready to load
- ROB issued
- ROB completed
- ROB ready to writeback
- ROB ready to commit

---

Copyright 
Binghamton University State University of New York
Thomas J. Watson School of Engineering and Applied Science

[marss86.org](http://marss86.org)
Load Store Queue (LSQ)

- LoadStoreQueueEntry structure
  - Data
  - Memory address
  - Operation type
    - Load/store, MMIO, and Memory Fence
- Load Store Queue
  - Resolve Memory Dependency/Aliasing
  - Data Forwarding/Merge
  - Split phase Store

### LoadStoreQueueEntry

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>W64</td>
<td></td>
</tr>
<tr>
<td>data</td>
<td>W64</td>
<td></td>
</tr>
<tr>
<td>ReorderBufferEntry *rob;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>idx</td>
<td>W16</td>
<td></td>
</tr>
<tr>
<td>core</td>
<td>OooCore*</td>
<td></td>
</tr>
<tr>
<td>store:1, ifence:1, sfence:1, entry_valid:1, mmio:1;</td>
<td>W8</td>
<td></td>
</tr>
<tr>
<td>sfr_data;</td>
<td>W64</td>
<td></td>
</tr>
<tr>
<td>sft_bytemask</td>
<td>W8</td>
<td></td>
</tr>
</tbody>
</table>
Physical Register

- Represented by PhysicalRegister structure
- Physical register holds the actual data
- Physical register allocated per uop including branch and store
  - Branch: PR stores the target RIP
  - Store: PR stores the merged data
- Done to keep the design simple
- Arch

```c
struct PhysicalRegister: public selfqueueLink {
  ReorderBufferEntry* rob;
  W64 data;
  W16 flags;
  W64 idx;
  W8 coreid;
  OoOCore* core;
  W8 rfid;
  W8 state;
  W8 archreg;
  W8 all_consumers_sourced_from_bypass:1;
  W16s refcount;
  W8 threadid;
}
```

Physical Register states
- None
- Free
- Waiting
- Bypass
- Written
- Pending free
Physical Register Files

- Supports both separate or unified FP/INT register files
- RegisterRenameTable structure manages the rename information
  - SpecRRT points to the registers with speculative states (that is non-committed)—Future File
  - CommitRRT points to the registers that carry the architectural (committed) values (Architectural Register File)

```
MOV R1,1
MOV RB,1
MOV RC,RA,RB
ADD RC,RA,RB <- current rip
ADD RB,RC,RA
SUB RB,RC,RB
ADD RA,EA,RA
```
Function Units

- Configurable number of function units
- Similar function units can be grouped
- FunctionalUnitInfo structure holds opcode to function unit map and the opcode latency

```c
struct FunctionalUnitInfo {
    byte opcode;
    byte latency;
    W16 fu;
};
```
Translation Lookaside Buffer

- Single level TLB
- Separate iTLB and dTLB
- Simulate four level page walk on TLB miss
Atom-Like In-Order Pipeline

- Early Intel® Atom architecture has CISC design
- MARSS In-Order core merge multiple MARSS uops of one instruction into one AtomOp
  - Reuse x86 decode logic and functional execution
Multi-core Model

- Create multiple instances of Core Models to simulate multi-core
  - Can use different core models for heterogeneous architecture
- Cores can have share/private caches
- All cores share a global clock
  - Variable: \textit{sim\_cycle}
- Ongoing features:
  - DVFS (common to all cores) – exists internally, Per-Core DVFS
  - Intel TSX – Transactional memory support
MARSS Memory Hierarchy Module

Brendan Fitzgerald
SUNY Binghamton
fitz@cs.binghamton.edu

marss86.org
Memory Hierarchy

- Components of multilevel caches, main memory and I/O
  - Controller + Storage
  - Components are connected by interconnections ("interconnect")
- Interface of CPU cores
  - access_cache, core_wakeup, clock
- Components defined in “machine” configuration, parameters can be reconfigured at runtime
- File: ptlsim/cache/*.h,cpp

![Diagram of memory hierarchy with CPU, cache component, and main memory component connected by interconnect module.](image)
Memory Hierarchy Simulation

- Event driven simulation
  - A “MemoryRequest” is created in the CPU that issues a memory reference
    - Goes through all level memory hierarchy if needed
  - One request per issued memory operation – easy to track request through all components
  - A “Message” including “MemoryRequest” is used for inter-component communication
  - The event and callback function are used to simulate delay
  - File: ptlsim/cache/memoryRequest.{h,cpp}
Communication using Messages

- Modules communicate using “Message”:
  - A short lived object having all routing information
  - File: ptlsim/cache/controller.h
Signal Interface

- Each module exposes a ‘Signal’ object
  - Overrides ‘Signal’ callback function to handle messages
- Other modules send messages via the exposed ‘Signal’ object
Controller Module

- **Component**
  - Controller + Storage
  - Module to control storage
    - Cache controller, directory controller, main memory controller, I/O controller
  - Controllers are connected by ‘Interconnect’
  - Controllers communicate using ‘Message’
- **File:** ptlsim/cache/controller.h
  - **Interface Signal:** handle_interconnect
Cache Controllers

- Multi-connected cache controller
  - Two variants: coherent cache controller and simple cache controller
  - Use “Signal” to communicate with lower and upper level interconnections
  - File: ptlsim/cache/{cacheController,coherentCache}.{h,cpp}

Receive message into module’s “Signal” - handle_interconnect

Send message to Interconnect’s “Signal” interface (Access using get_controller_request_signal())

Attach interconnection:
- register_interconnect [type: upper, lower]

outstanding request queue
Simple Cache Controller Module

- Write back and write through cache models
  - Simple cache without coherency
  - Used in building Single Core Caches or Shared Caches
  - File: ptlsim/cache/cacheController.{h,cpp}

Cache Storage (cache lines):
- set
- way
- linesize
- latency
- readports
- writeports
- LRU

Queue and dependency check:
- pendingRequests
- find_dependency
- find_match

Request handling using events:
- cacheAccess
- cacheHit
- cacheMiss
- cacheInsert
- clearEntry
Simple Cache Access: Processing Flow

- **Request from Upper Interconnect**
  - Queue Available?
    - Yes: Add to outstanding queue
    - No: Queue Available?
      - Yes: Add to outstanding queue
      - No: Port free?
        - Yes: (cacheAccess,1)
        - No: Port conflict delay

- Hit/
  - Miss?
    - Yes: Hit/
      - Miss?
        - Yes: Hit/ Miss?
          - No: (cacheAccess,1)
          - Yes: (cacheHit, latency)
            - Remove request from queue
              - Send response to Upper Interconnect
            - (cacheMiss, latency)
              - Send to Lower Interconnect

*Message including request
*latency: cache storage access latency
Simple Cache Response: Processing Flow

- Request from Lower Interconnect
- Is response?
  - No: Ignore Request
    - (cacheInsert,1)
      - Update Cache
      - (clearEntry, latency)
      - Remove request from queue
  - Yes: Wakeup request from queue
    - (waitInterconnect,1)
      - Send to Upper Interconnect

*Message including request
*latency: cache storage access latency

Internal cache housekeeping

Response to upper level
Coherent Cache Controller Module

- Coherence Logic on the top “Simple Cache model”
  - MESI and MOESI
  - File: ptlsim/cache/coherentCache.{h,cpp}, coherenceLogic.h, mesiLogic.{h,cpp}, moesiLogic.{h,cpp}

Coherence Logic API:
- handle_local_hit
- handle_local_miss
- handle_interconn_hit
- handle_interconn_miss

Cache Coherence Module

1. Request from Upper Interconnect
   - No → Queue Available?
     - Yes → Add to outstanding queue
     - No → (cacheAccess,1)
2. (cacheAccess,1)
   - Port free?
     - Yes → Hit/ Miss?
       - Yes → Coherence: handle_local_hit
       - No → Coherence: handle_local_miss
   - Miss → Coherence: handle_local_miss
3. Coherence Miss?
   - No → Remove request from queue
     → Send response to Upper Interconnect
   - Yes → Send to Lower Interconnect

Queueing delay
Port conflict delay
Cache storage access delay

(cacheHit, latency)
(cacheMiss, latency)
(cacheAccess, 1)
Coherent Cache Response/Remote Access: Processing Flow

1. Request from Lower Interconnect
2. Is response?
   - Yes: Wakeup request from queue
     - (waitIInterconnect, 1)
     - Send to Upper Interconnect
   - No: Is response?
     - Yes: Update Cache
       - (clearEntry, latency)
       - Remove request from queue
     - No: (cacheAccess, 1)
3. Port free?
   - Yes: Hit/Miss?
     - Hit: Coherence: handle_remote_hit
       - Prepare response with data
       - Send to Lower Interconnect
     - Miss: Coherence: handle_remote_miss
       - Prepare response without data
       - Send to Lower Interconnect
   - No: (cacheAccess, 1)
Main Memory Controller Module

- Simple memory controller model
  - Outstanding request queue, multiple banks, and cache line interleaving address mapping
  - Simulates bank contention and has a fixed bank access latency
- File: ptlsim/cache/memoryController.{h,cpp}
MARSS Interconnection Module

Brendan Fitzgerald
SUNY Binghamton
fitz@cs.binghamton.edu

marss86.org
Interconnection Module

- Connection between two or more controller modules
  - Connection Fabric & Queuing Logic
- On-chip and Off-chip Interconnections
  - Point-2-Point, Split-phase Bus, Crossbar Switch
- Event-driven timing simulation

![Diagram showing interconnection module with controller nodes and connection fabric](image-url)
Interconnect Module - APIs

Connect controller:
- `register_controller(Controller*)`

Receive message into module's "Signal" - `controller_request`

Send message to Controller’s Signal Interface (Access using `controller->get_controller_request_signal()`)

Controller -> Connection Fabric

Controller
Interconnect Module – Point-2-Point (P2P)

- Used to connect only two controllers
- Simulate two sets of wires
  - No resource contention
  - Does not buffer any requests
  - Simple design - currently no delay
- Files: ptlsim/cache/p2p.{h,cpp}
Interconnect Module – Split Phase Bus

- Connect two or more controllers
- Split address and data transactions
- Supports Snooping/Directory Coherence Protocol
- Round-robin arbitration
- Queue for each connected controller
- Files: ptlsim/cache/splitPhaseBus.{h,cpp}

Simulated Timing Events:
- broadcast+arbitration
- broadcastCompleted
- dataBroadcast
- dataBroadcastCompleted

Simulated Delay:
- Broadcast: default – 6 cyc
- Arbitration: default – 1 cyc
Split Phase Bus: Processing Flow

Message from Controller

Controller’s Queue Available?

Yes

No

Yes

(broadcast,1)

Arbitrate & initiate broadcast

(broadcastComplete, latency)

Send message to all Controllers

Add to outstanding request queue

No

Address Bus available?

Yes

No

Data Bus available?

Yes

No

Response received from all Controllers

Initiate data broadcast

Send message to all Controllers

Remove from outstanding request queue

*Message including request latency: bus broadcast latency

Arbitration and address broadcast delay

Data broadcast delay
Interconnect Module – Crossbar Switch

- Connect multiple controllers
  - Using NxN crossbar
- Per controller incoming buffer
- Simulated timing events
  - send: Resolve conflicts, 3 cycles
  - sendComplete: Send message with simulate crossbar latency
- Files: ptlsim/cache/switch.{h,cpp}
Building the Simulation Machines

Avadh Patel
SUNY Binghamton
apatel@cs.binghamton.edu
General overview of machine building

- Each module contains a ‘Builder’
  - to create one or more module objects
- Three types of builders
  - Core builder
  - Controller builder
  - Interconnect builder
- Each builder has a unique name
  - used in ‘Machine Configuration’ files to create module instances
- File: config/* .conf
Machine Builders - Example

**ControllerBuilder**

virtual Controller* get_new_module(...)

**WbCacheController Builder**

const char* name = "wbcache"

Controller* get_new_module(...) 

**WriteThroughCache ControllerBuilder**

const char* name = "wtcache"

Controller* get_new_module(...) 

**CacheController**

write-back = True;

**CacheController**

write-back = False;
Machine Configuration Files

- Static configuration of simulation machines
  - Instances of Core, Controller and Interconnect modules
  - Connection between modules
  - Configuration parameters
- Support multiple configurations
  - Select one at runtime using ‘-machine’
- Four major sections: core, cache, memory and machine
- Configuration files in YAML format

```yaml
machine:
  sample_machine:
    description: 2 wide OoO Machine
    min_context: 1
    max_context: 1
    cores:
      - type: ooo2w
        name_prefix: ooo_2w_
        insts: 1
        option:
          threads: 1
    caches:
      - type: l1_64k_wb
        name_prefix: L1_I_
        insts: 1
      - type: l1_64k_wb
        name_prefix: L1_D_
        insts: 1
      - type: l2_256k_wb
        name_prefix: L2_
        insts: 1
    memory:
      - type: dram_cont
        name_prefix: MEM_
        insts: 1
    interconnects:
      - type: p2p
        connections:
          - core_1:
            L1_I_0: UPPER
          - core_2:
            L1_D_0: LOWER
          - L1_I_0: LOWER
            L2_0: UPPER
          - L1_D_0: LOWER
            L2_0: UPPER
          - L2_0: LOWER
            MEM_0: UPPER
```
Machine Configuration - Cores

- **Core configuration** defines a new core with specific configuration parameters
- Current models include
  - OoO
  - In-order, like Atom
- Default parameter values in: ‘ptlsim/core/*/const.h’

### Core Configuration Grammar

```
core: 
  <core_name>: 
    base: <core_builder_name> 
    params: 
      <param_name> : <value> 
```

### 2-wide OoO Core Configuration

```
core: 
  ooo2w: 
    base: ooo
    params: 
      ISSUE_WIDTH: 2
      FETCH_WIDTH: 2
      COMMIT_WIDTH: 2
```
Machine Configuration - Caches

- **Cache configuration** defines a cache model with various cache-storage parameters
- Current models include
  - wb_cache
  - wt_cache
  - moesi_cache
  - mesi_cache

### Cache Configuration Grammar

```
cache:
  <cache_name>:
    base: <cache_builder_name>
    params:
      <param_name> : <value>
```

### 2MB MESI L2 Cache Configuration

```
cache:
  l2_2M_mesi:
    base: mesi_cache
    params:
      SIZE: 2M
      LINE_SIZE: 64
      ASSOC: 8
      LATENCY: 5
```
Machine Configuration - Memory

- **Memory configuration** defines a controller model for memory hierarchy beyond on-chip caches.

- Current models include:
  - `simple_dram_cont`
  - `global_directory` (for coherence)

### Memory Configuration Grammar

```
memory:
  <module_name>:
    base: <controller_builder_name>
    params:
      <param_name> : <value>
```

### Simple DRAM Cont Configuration

```
memory:
  dram_cont:
    base: simple_dram_cont
```
Machine Configuration - Machine

- Define number of instances of different modules
- Provides connection between modules using ‘Interconnects’
- Machine-specific options for each module

```yaml
machine:
sample_machine:
  description: 2 wide OoO Machine
  min_context: 1
  max_context: 1
  cores:
    - type: ooo2w
      name_prefix: ooo_2w_
      insts: 1
      options:
        threads: 1
  caches:
    - type: l1_64k_wb
      name_prefix: L1_I_
      insts: 1
    - type: l1_64k_wb
      name_prefix: L1_D_
      insts: 1
    - type: l2_256k_wb
      name_prefix: L2_
      insts: 1
  memory:
    - type: dram_cont
      name_prefix: MEM_
      insts: 1
  interconnects:
    - type: p2p
      connections:
        - core_S: I
          L1_I_S: UPPER
          L1_D_S: LOWER
          L2_0: LOWER
          L2_0: UPPER
          MEM_0: UPPER
```

Set min and max allowable CPU context
Module specific options
Connects cores and controllers using P2P
Connect two components
MARSS Statistics Collection

Avadh Patel
SUNY Binghamton
apatel@cs.binghamton.edu
Summary of Statistics Collection

- Collect and store detailed statistics of each module
- Store statistics in following methods:
  - YAML (default format)
  - Flat – non-hierarchical
  - Periodic Dump
  - MongoDB
- *Separate user and kernel level* statistics

Sample Statistics in YAML Format

```yaml
base_machine:
  ooo_0_0:
    thread0:
      result:
        ok: 17558763
        block: 4209145
        fail: 16596393
        cache-miss: 5532243
        skip: 75
        width: [1212365438, 7097168, 3607205, 120331, 721548]
```

marss86.org
Statistics Data Storage

- Storage is detached from statistics objects
- Allows multiple counter values for different phase of simulation
  - Currently supports two phases:
    - User level
    - Kernel level
- Future work: collect statistics for each task or specific functions of the task
Statistics Collection Interface - Counters

- Statistics collection support Classes:
  - `StatObj<W64>` `counter`;
    - Used for basic counters
  - `StatArray<W64, SIZE>` `counter_array`;
    - Array of counters
  - `StatEquation<W64, double, StatFormula>` `computed_value`;
    - Auto compute value at the end of simulation
  - Supported formulas:
    - `StatObjFormulaAdd`
    - `StatObjFormulaDiv`
Statistics Collection Interface - Statable

- All statistics objects are stored within ‘Statable’ objects
- All Statable objects are attached to global statistics tree

Sample Statable Class

```cpp
class HitMissCounter : public Statable
{
    StatObj<W64> hit;
    StatObj<W64> miss;
    StatEquation<W64, W64, StatObjFormulaAdd> total;

    HitMissCounter(Statable *parent) :
        Statable(parent, "hit_miss")
        , hit("hit", this)
        , miss("miss", this)
        , total("total", this)
    {
        total.add_elem(&hit);
        total.add_elem(&miss);
    }
}
```
Getting Started with MARSS

Brendan Fitzgerald
SUNY Binghamton
fitz@cs.binghamton.edu

marss86.org
Web Resources

- Wiki: http://marss86.org
- Repo: https://github.com/avadhpatel/marss
- Mailing list: https://www.cs.binghamton.edu/mailman/listinfo/marss86-devel
- Blog: http://marssandbeyond.blogspot.com/
Disk Images

- MARSS uses unmodified disk images
  - If Qemu can boot it, MARSS can simulate it
- Download image from MARSS website
  - Ubuntu 11.04 (The image used in this presentation)
  - Ubuntu 9.04 with Parsec 2.1
  - Ubuntu 9.04 with splash 2
- Create your own image
  - Ubuntu-vm-builder
  - Qemu-img
From 0 to Simulation in 5 Steps (Step1)

- Install the requirement (Ubuntu host machine)
  - `sudo apt-get install g++ git scons libSDL1.2-dev zlib1g-dev`

- Download MARSS
  - `git clone https://github.com/avadhpatel/marss.git`

- Download and uncompress disk image
  - `tar -xvf ubuntu-natty.tar.bz2`

- Compile MARSS (MARSS uses scons as software construction tool)
  - `Scons`
From 0 to Simulation in 5 Steps (Step 2)

- **Start MARSS**
  - qemu/qemu-system-x86_64 ubuntu-natty.qcow2
  - Wait until the image boot to the guest OS

![Image of QEMU console output]

boots from (hd0,0) ext4 0c581cbb-4c0b-4503-a7f4-0bf9dec57e15
Starting up ...

Ubuntu 11.04 ubuntu tty1

```
ubuntu login: root
Password: 
```

Last login: Fri May 18 21:56:37 UTC 2012 on tty1
Welcome to Ubuntu 11.04 (GNU/Linux 2.6.38-10-virtual x86_64)

* Documentation: https://help.ubuntu.com/

root@ubuntu:~# ls
bin create_checkpoint FFT kill_sim start_sim stop_sim time_check
root@ubuntu:~# _
From 0 to Simulation in 5 Steps (Step 3)

- Switch to monitor screen (ALT+CTL+2)
- (Qemu) simconfig --machine single_core
From 0 to Simulation in 5 Steps (Step 4)

- Switch to guest console screen (ALT+CTL+1)
- Simulate ls command

./start_sim ;ls ;./stop_sim;./kill_sim
From 0 to Simulation in 5 Steps (Step 5)

- The host terminal shows the simulation progress
- Ptlsim.log
  - Basic stats
Building and running MARSS in detail
Machine Configuration File

- **machine**: shared_l2
  - **description**: Shared L2 Configuration
  - **min_contexts**: 2
  - **cores**: The order in which core is defined is used to assign
    - **type**: ooo
      - **name_prefix**: ooo_
    - **type**: l1_cache
      - **name_prefix**: L1_I
        - **insts**: $NUMCORES
        - **option**: private: true
    - **type**: l1_cache
      - **name_prefix**: L1_D
        - **insts**: $NUMCORES
        - **option**: private: true
    - **type**: l2_cache
      - **name_prefix**: L2
        - **insts**: 1
    - **type**: dram_cont
      - **name_prefix**: MEM
        - **insts**: 1
        - **option**: latency: 50
    - **interconnects**: split_bus
      - **type**: p2p
        - **connections**: L1_I_: I, L1_D_: D, L2: LOWER, MEM: UPPER
      - **type**: split_bus
        - **connections**: L1_I_: LOWER, L1_D_: LOWER, L2: UPPER

**Diagram**

- Ooo_1
  - L1_I_1
  - L1_D_1
  - L2_1
  - MEM_1
- Ooo_2
  - L1_I_2
  - L1_D_2
  - split_bus
Compiling MARSS

- Using scons
- Compiling keys
  - $c = N$: number of cores
  - Debug=1
    - enable debugging options
  - Logging
    - gcc -O1
- Pretty
  - Pretty 0
  - Pretty is 1
MARSS Runtime Simulation Configuration

- Simulator’s parameters can be set in multiple ways
  - When launching MARSS using config file
  - At run time using Qemu monitor CLI
  - From the guest machine using PTLCalls
MARSS Runtime Simulation Configuration – Method 1

- Host machine
  - Using simconfig file
  - Specify simconfig file in commandline when launching MARSS
    - qemu/qemu-system-x86_64 -simconfig test.cfg diskimag.qcow2

```
# MARSS config file
-stopinsns 100m
-machine single_core
-logfile test.log
{-#loglevel 10
-yamlstats test.yml
```
MARSS Runtime Simulation Configuration – Method 2

- Qemu console
  - using “simconfig” command
  - Can be issued any time
  - Simconfig –help shows full list of MARSS configuration parameters on the host machine

![Qemu console output showing simconfig command and options]

marss86.org
MARSS Runtime Simulation Configuration – Method 3

- Guest machine
  - Use binaries to pass a ptlcall to MARSS
  - Has to be compiled at the Host machine with MARSS code
    - Section 4.1
# Table of Most Used Commands

<table>
<thead>
<tr>
<th>Simconfig</th>
<th>Qemu monitor</th>
<th>Guest terminal</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-run</td>
<td>-run</td>
<td>start_sim</td>
<td>Start simulating</td>
</tr>
<tr>
<td>INV</td>
<td>-stop</td>
<td>stop_sim</td>
<td>Stop simulating</td>
</tr>
<tr>
<td>INV</td>
<td>-kill</td>
<td>Kill_sim</td>
<td>Kill MARSS</td>
</tr>
<tr>
<td>-kill-after-run</td>
<td>-kill-after-run</td>
<td>Not provided</td>
<td>Kill MARSS when run simulation finish</td>
</tr>
<tr>
<td>-logfile <em>name</em></td>
<td>-logfile <em>name</em></td>
<td>Not provided</td>
<td>Set log file to <em>name</em></td>
</tr>
<tr>
<td>-stopinsns <em>N</em></td>
<td>-stopinsns <em>N</em></td>
<td>Not provided</td>
<td>Stop simulating after <em>N</em> instructions</td>
</tr>
<tr>
<td>-stats <em>name.yml</em></td>
<td>-stats <em>name.yml</em></td>
<td>Not provided</td>
<td>Set stats file to <em>name.yml</em></td>
</tr>
<tr>
<td>-corefreq <em>N</em></td>
<td>-corefreq <em>N</em></td>
<td>Not provided</td>
<td>Set Chip Core Frequency in Hz</td>
</tr>
</tbody>
</table>
Statistics File

- MARSS collects both user and kernel level statistics
- Statistics are in plain text
- Statistics are structured in yaml
- Statistics are hierarchical in format

L2_cpurequest_miss_read

123
Sample Statistics File

- “Tags” show to what running level and core the stats belong to
- “performance” is the simulator performance not the benchmark’s performance
- Little hard to read?
  - We have a script for that
Mstats Script

- Process stats yaml files
- Supports simpoint weight files
- Flattens all or part of the stats to enable easy grep support
  - `mstats.py -y --flatten -n base_machine::ooo_.\*:commit -t user result.stats`
- Supports tag search
  - `mstats.py -y --yaml-out -n base_machine::ooo_.\*:commit::ipc -t user result.stats`
- Supports histograms for node
  - `mstats.py -y --hist -n base_machine::ooo_0_0::dispatch::width -t user result.stats`
Checkpoints

- For consistent runs MARSS uses checkpoints
- Use qemu savevm/loadvm feature
- Works with qcow2 image but doesn’t work with raw image
- Create_checkpoint binary in the provided images creates the checkpoint
- To start the simulation directly after the vm loaded add “-run” to the simconfig file
- Example:

```
root@ubuntu:~# ./create_checkpoint test ;./FFT ;./stop_sim _
```
Fast Forwarding

- Fast forwarding can be used to skip N number of instructions and start simulations
- Can skip user instructions or user+kernel instructions
- Checkpoint_after binary uses ptlcall to skip N user instructions and create checkpoint
Run_bench Script

- Run multiple benchmarks in parallel
- Run same benchmark multiple times
- Can have multiple configuration per benchmark
- Send email when the benchmarks done
- 

```bash
./run_bench.py -c util.cfg -e me@gmail.com -i 3 -d logdir -n 6 spec2006
```

- `-c Util.cfg`: run_bench configuration file
- `-e`: send email after completion using send_gmail.py
- `-d logdir`: store all the logfiles, stat files and output files in logdir
- `-i 3`: run every benchmarks for 3 times
- Benchmarks set to run
MARSS environment setup

Benchmark suites

Run configuration

Per suite
Create_checkpoints Script

- Create checkpoints for all the benchmarks in a suite in a single image
- Almost zero configuration is required for already supported suites
- Support
  - Spec
  - Parsec
  - Splash
- More benchmark suites can be added by simply adding the command lines and names
Use Case Study: Evaluating Main Memory System Efficiency in Full System Environment

Hongzhong Zheng
Rambus Labs
hzheng@rambus.com
System Simulation With Detail Memory Model

- MARSSx86 + **Detail Main Memory Model**
  - ARECIBO (Rambus Inc.)
  - DRAMSim2 (UMD)
    - More detail in ISCA-2012 MARSS Tutorial
- Detail main memory system modeling
  - Main memory system organization
    - Channels, DIMMs, Ranks, Banks
  - DRAM component timing
    - tCL, tRCD, tRP, tRC, tFAW, tRRD, etc.
  - Address mapping and page policy
    - Open/closed page policy, cache-line/page interleaving
  - Scheduling policies: FCFS, Hit-first etc.
  - Power model

![Diagram of System Simulation With Detail Memory Model]

**MARSSx86** Memory Interface

**Shared LLC**

**MemoryModel.so**

Cycle X: AcceptReq()

Cycle X+n: ResponseReq()
Memory Challenges of Many-Core CPUs

- Many-Core CPUs in large data centers
  - SeaMicro, Tilera etc.

- Up to 4 memory channels for each many-core CPU
  - Main memory system shared by all cores
  - Only one access severed by a channel bus at any given time
  - Potential power/performance system bottleneck

Detail many-core system evaluation and analysis required to conquer memory challenges
Background: Memory System Power

- **Background**
  - related to power state transition and power management policies

- **Operation**
  - Activation + Precharge
  - Read/write
  - I/O power
  - Driving output + termination

Rambus innovation to improve power efficiency

**Power Breakdown a Memory Intensive Workload (2CH-1D-2R-x4, 4Gb device)**

- I/O
- read/write
- operation
- background

Four ‘Ibm’ active instances on four-core system

21GB/s, 85% channel utilization
Module Threading to Reduce Activation Power

- Conventional module
  - All 8 devices in rank for single access

- Module threading
  - Reduce active power by operating a subset of devices in rank for single access
    - 2/4/8-way
  - Increase memory level parallelism
    - More independent banks
  - Extending data burst time
Evaluate Memory System Power Efficiency Using MARSS – Module Threading

Simulated computing system

<table>
<thead>
<tr>
<th>OS</th>
<th>Linux 2.6.31-4 64bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>32-core, ATOM processor</td>
</tr>
<tr>
<td></td>
<td>1.6GHz</td>
</tr>
<tr>
<td>ROB: LSQ:</td>
<td>64: 48; FetchQ: 24; IQ: 32</td>
</tr>
<tr>
<td>Fetch/Issue/Commit width:</td>
<td>2</td>
</tr>
<tr>
<td>INT/FP registers:</td>
<td>64/64</td>
</tr>
<tr>
<td>ITLB/DTLB: 16/16</td>
<td></td>
</tr>
</tbody>
</table>

Cache

| L1:32KB, 2-way, 2cc; DL1:32KB, 4-way, 2cc; L2:256KB, 8-way, 5cc; Queue: 128; cache coherence: MESI |

Main memory (ARECIBO)

| 2CH-1D-2R-8BA, DDR3-1600-x4, 4Gb/device |
| 11-11-11, read-first, power_down_fast |
| close page, auto precharge, BL8, queue:128 |
| memory controller overhead: 10ns |

Module Threading configuration

<table>
<thead>
<tr>
<th>Module Threading</th>
<th>baseline</th>
<th>operate 16 devices per access, 4 cyc data burst per access</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2-way MT</td>
<td>operate 8 devices per access, 8 cyc data burst per access</td>
</tr>
<tr>
<td></td>
<td>4-way MT</td>
<td>operate 4 devices per access, 16 cyc data burst per access</td>
</tr>
</tbody>
</table>

Many-Core ATOM based CPU

Two-level coherent cache

Dual-channel DRAM memory system
Module Threading Reduces Memory System Power

- Significant operation power saving
  - Only activate half devices for each access → operation power saving
  - More independent “ranks and threads” for low power mode → background power saving

*base: conventional DDR3-1600 without module threading
Module Threading Performance/Energy Impact

- Small performance loss due to extend data burst time
- Significant improvement on memory energy per bit by module threading

*baseline: conventional DDR3-1600 without module threading
LPDDRx vs. DDRx for Cloud Computing

- LPDDRx
  - Designed for mobile system with good energy efficiency
  - Limited bandwidth, long access latency and limited capacity

- LPDDRx for Cloud Computing
  - Bring better energy efficiency at cost of performance and cost
  - What’s the system performance impact?

Technical Pros and Cons of LPDDRx vs. DDRx

- Low voltage and low leakage
  - Good energy efficiency but slow row access cycle time

- NO DLL
  - Low background power
  - Long read synchronization latency
  - Higher jitter

- Unmatched Termination
  - Signal integrity challenges
  - LPDDR3 includes OTD in the recent published spec

- Wide DQs per component
  - Big challenge for building high capacity memory system
Evaluate Memory System Performance Using MARSS For DDR3 vs. LPDDR3

Simulated computing system

<table>
<thead>
<tr>
<th>OS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux 2.6.31-4 64bit</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Processor</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>32-core, ATOM processor</td>
<td></td>
</tr>
<tr>
<td>1.6GHz</td>
<td></td>
</tr>
<tr>
<td>ROB: 64; LSQ: 48; FetchQ:24; IQ:32</td>
<td></td>
</tr>
<tr>
<td>Fetch/Issue/Commit width: 2</td>
<td></td>
</tr>
<tr>
<td>INT/FP registers: 64/64</td>
<td></td>
</tr>
<tr>
<td>ITLB/DTLB: 16/16</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cache</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>IL1:32KB, 2-way, 2cc; DL1:32KB, 4-way, 2cc</td>
<td></td>
</tr>
<tr>
<td>L2:256KB, 8-way, 5cc;</td>
<td></td>
</tr>
<tr>
<td>Queue: 128; cache coherence: MESI</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Main memory (ARECIBO)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2CH-1D-2R-8BA, DDR3/LPDDR3-1600</td>
<td></td>
</tr>
<tr>
<td>read-first, power_down_fast</td>
<td></td>
</tr>
<tr>
<td>close page, auto precharge, BLB, queue:128</td>
<td></td>
</tr>
<tr>
<td>memory controller overhead: 10ns</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key timing</th>
<th>DDR3-1600 (ns)</th>
<th>LPDDR3-1600 (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>tRL</td>
<td>13.75</td>
<td>15</td>
</tr>
<tr>
<td>tWL</td>
<td>12.5</td>
<td>7.5</td>
</tr>
<tr>
<td>tRCD</td>
<td>13.75</td>
<td>18</td>
</tr>
<tr>
<td>tRP</td>
<td>13.75</td>
<td>18</td>
</tr>
<tr>
<td>tRAS</td>
<td>35</td>
<td>42</td>
</tr>
<tr>
<td>tRC</td>
<td>48.75</td>
<td>60</td>
</tr>
<tr>
<td>tRRD</td>
<td>6.25</td>
<td>10</td>
</tr>
<tr>
<td>tFAW</td>
<td>30</td>
<td>50</td>
</tr>
<tr>
<td>tXP</td>
<td>6.25</td>
<td>7.5</td>
</tr>
<tr>
<td>tWR</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>tRTP</td>
<td>7.5</td>
<td>7.5</td>
</tr>
<tr>
<td>tWTR</td>
<td>7.5</td>
<td>7.5</td>
</tr>
<tr>
<td>tRFC</td>
<td>110</td>
<td>90</td>
</tr>
<tr>
<td>tREFI</td>
<td>7800</td>
<td>3900</td>
</tr>
</tbody>
</table>
DDR3 vs. LPDDR3 System Performance Impact

- Performance impact depends on applications
  - High performance loss for memory sensitive workloads
  - Energy estimation not available due to power related number not known yet
**DDR3 vs. LPDDR3 Average Read Latency**

- **Impact**
  - LPDDR3 increases average read latency due to extended timing compared with DDR3
  - Translate to performance loss depending on applications
  - Full system evaluation brings accurate result and deep insight
    - Detail CPU model + memory subsystem model

- **Graphs**
  - System Performance Impact of LPDDR3-1600 as Server Memory for Many Core System (32-ATOM-Core, 2-issue/core, 256KB-LLC/core; 2CH-1D-2R)
  - Memory Read Latency Impact of LPDDR3-1600 as Server Memory for Many Core System (32-ATOM-Core, 2-issue/core, 256KB-LLC/core; 2CH-1D-2R)

  - The graphs show relative system performance and average read latency for various workloads such as `ibm`, `omnetpp`, `hmmer`, and `astar`. LPDDR3 is indicated by a red bar, while DDR3 is shown in blue. The graphs highlight the impact on memory latency sensitive workloads.
Summary

- MARSS is a great simulation infrastructure
  - Evaluates real applications in full system on varied computer system architectures
  - Provides detailed micro-architecture statistics
  - Reasonable simulation speed

- MARSS can enable detailed component simulation in full system environment
  - Has friendly interface to integrate third party component simulation module, e.g. main memory or I/O module
Intel TSX Support in MARSS

Kanad Ghose
SUNY Binghamton
ghose@cs.binghamton.edu
Transactional Memory Model – The Need

- Example: updating a data structure with different fields

- Activity P has to update B and C based on the value of B
- Activity Q has to update A and D based on the value of D
- P and Q execute concurrently
Solution 1: Coarse-Grained Locking

- Uses a single lock for the entire data structure:

  - Activity P written to grab this lock, perform the updates and release the lock
  - Activity Q follows a similar sequence
  - Serialization delay due to passing of lock, other activities updating different parts of the struct are delayed
Solution 2: Medium-Grained Locking

- Uses a lock for each list (blue list and orange list):

  - Activity P written to grab Lock 1 and Lock 2, perform the updates and release the locks
  - Activity Q written to grab Lock 1 and Lock 2, perform updates and release locks
  - Serialization delay due to passing of still present, but updates to other parts of list by other activities are not delayed
Solution 3: Fine-Grained Locking

- Uses a lock for each item in the lists

- Activity P written to grab Lock B and Lock C, perform the updates and release the locks
- Activity Q written to grab Lock A and Lock D, perform updates and release locks
- There is no contention over the locks, so lock passing delays are absent, updates to other parts of list or other list items by other activities are not delayed
Tradeoffs in the Solutions

- Solutions differ in many ways:
  - Number of locks used
  - Lock contention
  - Degree to which other activities are affected
  - Arguments for establishing correctness

- Coarse-grained solution: easy to argue correctness, only one lock but poor performance impeding parallelism

- Fine-grained: more locks to manage, error-prone solutions, difficult to argue about correctness in general BUT no performance impediments

- What is ideal: simplicity of coarse-grained locking with the performance of fine-grained locking – the transactional memory model has this ideal characteristic
The TM Model

- Parts of an activity that updates shared data is treated as an atomic transaction ("transactional region"), **explicit locking is not used**

- Transactions work in parallel and succeed (and **commit**) if they have no memory conflict and **abort** if they have a memory conflict
  - **Underlying system detects memory conflict**

- If transactions don’t conflict, no performance loss occurs – updates can be performed in parallel (as in the case of fine-grained locking)

- Aborts require handler to retry transactions or invoke a lock-based solution or other programmer-specified alternative
Intel TM Support

- Intel’s hardware support and ISA extensions to support TM (“Transactional Synchronization Support, TSX”) comes in two flavors:
  - **HLE - Hardware lock elision:**
    - HLE aborts handled by hardware using lock
  - **RTM – Restricted Transactional Memory**
    - Programmer demarcates transactional region in code with `XBEGIN` and `XEND` instructions
    - Abort handler address is specified as argument to `XBEGIN`
    - Abort handlers specified by user and adds flexibility
    - Two other instructions `XABORT` (explicit command to abort a transaction) and `XTEST` (check if within transactional region)

- **MARSS models the RTM support**
  - available in ‘features’ branch
Sample code with Intel TSX Instructions

- **Simple for loop with pthread locking and TSX instructions**

```c
for (int i = 0; i < 1000000; ++i) {
    pthread_mutex_lock(m);
    Value++;
    pthread_mutex_unlock(m);
}
```

```c
for (int i = 0; i < 1000000; ++i) {
    XBEGIN;
    value++;
    XEND;
}
```
RTM Implementation in MARSS

- Notes transactional state of L1 cache lines
- Maintains a buffer of potential updates made by a transaction, commencing with a XBEGIN
- Checks for memory conflicts with other transactions at XEND:
  - propagates updates on success
  - calls function to abort when conflicts are detected
  - delays for buffer setup, commitment and flushing modeled correctly
- Supports nested transactions
Implementation Details - 1

- New variables, structures:
  - **Context copy**: Keeps copy of CPU context at XBEGIN
  - **tsx_mode**: Per CPU context integer counter to indicate if CPU is in TSX mode or not and also indicate the nested level of transactions
  - **abort_addr**: Per CPU context will contain an 64 bit abort address, valid only when tsx_mode > 0
  - **tsx_memory_buffer**: Each CPU contains a set-associative buffer to store modified cache lines (address and data).

- Additions to core and cache models:
  - Core: new core_tsx_signal and callback-function added
  - Cache: new ops created to handle transaction states
    - **New op**: MEMORY_OP_TSX
    - Codes (RIP): Begin = 0x1, Commit = 0x2, Abort = 0x3
    - Core signal: Pointer to core’s callback function to abort TSX mode
    - Flags added to cacheline to indicate if they were read or written in the TSX mode
    - Cache coherence protocol modified to look at these flags
Implementation Details - 2

- XBEGIN: decoded into two uops:
  - Uop to call light assist function: `l_assist_xbegin` which:
    - updates tsx counter, saves CPU context to revert to on aborts, saves abort function address, initializes buffer entry, does bookkeeping checks
  - Memory fence, mf.all – prevents issues of loads and stores to memory

- XEND: decoded into a single uop:
  - This uop calls light assist function: `l_assist_xend` which:
    - checks for success or failure
    - on success updates tsx counter, clears saved context if outside transactional region, stalls ALL cores, propagates updates to RAM, unstalls cores
    - on failure calls `l_assist_xabort`, posts error code, performs bookkeeping functions etc.
Implementation Details - 3

- **XABORT**: decoded into a single uop
  - This uop calls `l_assist_xabort` which:
    - Notifies cache module through a `MEMORY_OP_TSX` with code = abort (0x3), stalls CPU, waits till cache callback function for abort is invoked
    - Restores saved CPU context, clears buffer
    - Performs some bookkeeping function and transfers control to specified abort function’s address etc.
# Implementation Details - 4

<table>
<thead>
<tr>
<th>Implementation</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>xbegin, xend, xabort, xtest</td>
<td>ptlsim/x86/decode-complex.cpp</td>
</tr>
<tr>
<td>I_assist_* functions</td>
<td>ptlsim/x86/decode-complex.cpp</td>
</tr>
<tr>
<td>Cache with TSX Support</td>
<td>ptlsim/cache/tsxMESILogic.{cpp, h}</td>
</tr>
<tr>
<td>Core with TSX Support</td>
<td>ptlsim/core/ooo/{ooo-exec, ooo-pipe}.cpp</td>
</tr>
<tr>
<td>CPU Context Changes</td>
<td>ptlsim/x86/ptlhwdef.h</td>
</tr>
</tbody>
</table>
MICRO-2012

Work-in-Progress

Avadh Patel
SUNY Binghamton
apatel@cs.binghamton.edu

marss86.org
Simulating Server Applications

- Emerging server applications:
  - Cloud services: Web, media, No-SQL, Big-data
  - Caching services like *memcached*
  - Complex to setup
- Modern Server applications are no longer single node and performance oriented
  - Proper evaluation requires “systems-of-systems” prospective
- MARSS is a perfect fit for these emerging workloads:
  - Supports executing unmodified binaries and full software stack
  - Already provides wide range of open-source emulation models for IO devices

marss86.org
Simulating Client-Server Setup

- Traditional Method

- System Under Test

* SUT and Clients can be in either same machine or in different machines.
Simulating Client-Server Setup (cont.)

- Issues with traditional method
  - Long simulation time – Unable to run full benchmarks
  - Introduce significant variations between simulation runs
    - No predefined order of requests from different clients
    - Unable to compare changes to system design
  - Difficult to find and manage ROI
Client-Server Simulation using Network Traces

- Goal: to remove inconsistencies with client requests
- Collect traces of client requests and replay them in controlled and consistent manner
- 3 step process:
  - Step-1: Collect network trace
  - Step-2: Process trace to extract Client Requests
  - Step-3: Replay trace in controlled environment
Step-1: Collect Network Packet Trace

- Capture all incoming and outgoing network packets
Step-2: Process Trace and Extract Requests

Channel Trace Generator

- **TCP Packet Extractor**
- **Channel Detector**
- **Insert Barrier before SEND**

- **Trace File**
  - Extract TCP level data and create* SEND/RECV packets

- **Channel Trace File**
  - Barrier placed before each SEND operation to order each request/response

- **Channels**
  - * Merge multiple packets of same type packets
Step-3: Replay Trace in Controlled Environment

- Channels allows concurrent requests to server
- Rate of Ethernet link can be simulated
Issues with Traces so far

- Run-by-run variations:
  - Servers tend to manage large number of threads and their scheduling is a mess
  - Dynamic runtime variations introduced by TCP/IP protocol stack in Linux kernel
- Some applications tend to open multiple ports at runtime which makes it hard to keep track using traces
Questions?

marss86.org