“deterministic deeds, done dirt cheap”

Joseph Devietti, Jacob Nelson, Tom Bergan
Luis Ceze, Dan Grossman
Determinism improves the software development cycle.

- **Debug**: Reverse debugging is possible.
- **Test**: No need to stress test. Testing results are reproducible. Tested inputs behave identically in production.
- **Deploy**: More robust production code. Production bugs can be reproduced in-house.
determinism improves the software development cycle
History of Deterministic Execution

Deterministic Execution for Restricted Programs

- Kendo [ASPLOS ‘09]
- Grace [OOPSLA ‘09]

Deterministic Execution for Arbitrary Programs

- DMP [ASPLOS ‘09]
- CoreDet [ASPLOS ‘10]
- dOS [OSDI ‘10]
- Determinator [OSDI ‘10]
- Calvin [HPCA ‘11]
- RC/DC [ASPLOS ‘11]
History of Deterministic Execution

seq. consistency  total store order  DRF0 [ISCA ‘90]

"Piled Higher and Deeper" by Jorge Cham
www.phdcomics.com
Jorge Cham © 2008
Contributions

**DMP-HB**
- A new deterministic consistency model based on DRF0 with improved performance

**RC/DC**
- A low-complexity hw/sw deterministic execution system
  - hw: store buffers and instruction counting
  - sw: everything else

**C/C++ compiler**
- Based on LLVM, runs on commodity multicore

**Hardware simulation using Pin**
starting simple: serialization

deterministic quantum size + deterministic scheduling
determinism

quantum round

threads

T_1

T_2

T_3

time →
recovering parallelism with DMP-TSO

**parallel mode**: buffer all stores (no communication)

**commit mode**: deterministically publish buffers

**serial mode**: for atomic ops

- T₁: wr A → rd A → parallel
- T₂: lock A
- T₃: rd A → lock B

Time →
Why is DMP-TSO slow?

Kendo [ASPLOS ‘09]

parallel

commit

serial

imbalance

T_1

T_2

T_3

time →
Why is DMP-TSO slow?

Kendo [ASPLOS ‘09]

 serialization

 imbalance

DMP-HB

parallel-mode synchronization
complements
relaxed consistency
synchronization in parallel mode with Kendo

[Olszewski et al., ASPLOS ’09]

thread with globally min insn count can do atomic op

T_1

T_2

T_3

lock A

instruct count →
Why is DMP-TSO slow?

Kendo [ASPLOS ‘09]

Serialization imbalance
Why is DMP-TSO slow?

parallel

commit

T_1
T_2
T_3

time →

Kendo [ASPLOS ‘09]
serialization
imbalance
DMP-HB
DRF0: happens-before consistency

[Adve and Hill, ISCA ‘90]

• happens-before edges defined by synchronization operations
• remote updates visible via cross-thread happens-before edges
• SC for DRF programs
• upholds C++/Java memory models
• programmer-visible model doesn’t change
sync in parallel mode (Kendo)
relaxed consistency (DRF0)
deterministic scheduling (DMP)

DMP-HB
DMP-HB: happens-before determinism

- **no serial mode**
- **less imbalance**

- **explicit fences**
  - rarely necessary

- **explicit fence iff**
  - inter-thread HB edge doesn’t cross commit
Outline

1. **DMP-HB**
   - A new deterministic consistency model with improved performance

2. **RC-DC**
   - A low-complexity hw/sw deterministic execution system
     - **hw**: store buffers and instruction counting
     - **sw**: everything else

3. Hardware simulation using Pin

4. C/C++ compiler based on LLVM, runs on commodity multicore
Store Buffers in Private $
application/OS$ can
choose nondeterminism
align context switches
with quantum boundaries

Precise Insn Counting
StartInsnCount
StopInsnCount
ReadInsnCount

Traps
SBFull
QuantumReached
Outline

1. **DMP-HB**
a new deterministic consistency model with improved performance

2. **RC\texttwiddle{DC}**
a low-complexity hw/sw deterministic execution system
   - hw: store buffers and instruction counting
   - sw: everything else

3. Hardware simulation using Pin

4. C/C++ compiler based on LLVM, runs on commodity multicore
Experimental Setup

Pin-based simulator
1 IPC, except for memory ops
PARSEC v2.1 with simsmall inputs

<table>
<thead>
<tr>
<th>structure</th>
<th>size</th>
<th>access latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>private L1</td>
<td>8-way, 32KB</td>
<td>1 cycle</td>
</tr>
<tr>
<td>private L2</td>
<td>8-way, 256KB</td>
<td>10 cycles</td>
</tr>
<tr>
<td>shared L3</td>
<td>16-way, 8MB</td>
<td>35 cycles</td>
</tr>
<tr>
<td>memory</td>
<td>-</td>
<td>120 cycles</td>
</tr>
</tbody>
</table>

extended CoreDet C/C++ compiler [ASPLOS ‘10]
8-core Intel Harpertown @ 2.8GHz, 10GB RAM
PARSEC v2.1 with simlarge inputs
Simulation: RC-DC Overheads

overhead < 60% in worst case

% overhead compared to nondet

quantum size (insns) 50k 50k 25k 1k 1k 50k 50k 50k 50k

blacksch dedup ferret fluid streamcl swaptions vips x264

overhead < 60% in worst case
Compiler: DMP-HB vs. DMP-TSO

% overhead compared to nondet

<table>
<thead>
<tr>
<th>threads</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>blackscholes</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>swaptions</td>
<td>150%</td>
<td>150%</td>
<td>150%</td>
<td>150%</td>
<td>150%</td>
<td>150%</td>
<td>150%</td>
<td>150%</td>
<td>150%</td>
</tr>
<tr>
<td>fluidanimate</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
<td>200%</td>
</tr>
<tr>
<td>fmm</td>
<td>250%</td>
<td>250%</td>
<td>250%</td>
<td>250%</td>
<td>250%</td>
<td>250%</td>
<td>250%</td>
<td>250%</td>
<td>250%</td>
</tr>
</tbody>
</table>

quantum size (insns)

200k | 200k | 50k | 50k
Conclusions

• DMP-HB: a new deterministic consistency model

• RC+DC: a new deterministic multiprocessor design
  – no speculation
  – lightweight hardware support

• Relaxed consistency is a natural optimization for determinism

source code and data available at http://sampa.cs.washington.edu
Thanks!

Questions?

source code and data available at
http://sampa.cs.washington.edu
1. Intra-processor dependencies are preserved.
2. All writes to the *same* location can be totally ordered based on their commit times, and this is the order in which they are observed by all processors.
3. All synchronization operations to the *same* location can be totally ordered based on their commit times, and this is also the order in which they are globally performed. Further, if $S_1$ and $S_2$ are synchronization operations and $S_1$ is committed and globally performed before $S_2$, then all components of $S_1$ are committed and globally performed before any in $S_2$.
4. A new access is not generated by a processor until all its previous synchronization operations (in program order) are committed.
5. Once a synchronization operation $S$ by processor $P_i$ is committed, no other synchronization operations on the *same* location by another processor can commit until after all reads of $P_i$ before $S$ (in program order) are committed and all writes of $P_i$ before $S$ are globally performed.