Deterministic Shared Memory Multiprocessing

Luis Ceze, University of Washington

joint work with Owen Anderson, Tom Bergan, Joe Devietti, Brandon Lucia, Karin Strauss, Dan Grossman, Mark Oskin.

Safe MultiProcessing Architectures at the University of Washington
A multithreaded voting machine

while (more_votes) {
    load t <- votes
    t++
    store t -> votes
}

thread 0

while (more_votes) {
    load t <- votes
    t++
    store t -> votes
}

thread 1

Non-determinism is evil.

votes == 2 votes == 2 votes == 1

votes == 2 votes == 2

Non-determinism is evil.
Shared Memory Parallel Programming is Hard

- sequential bugs
- concurrency bugs
- nondeterministic memory access interleavings

\{ 
  - hard to debug
  - hard to test
  - hard to replicate
  - hard to leverage crash information
\}
Record and Replay Systems

• What needs to be recorded?
  • volume of data?
  • practicalities?

• What problems does it solve?
Determinism Can Help

**Development**
- no more heisenbugs!
- time-travel debugging
- test inputs, not interleavings

**Deployment**
- reproduce bugs from field
- easy to synchronize replicas
- software behaves as tested

Effectively, make **parallel** programs behave like **sequential** programs...

*Can we remove non-determinism in arbitrary parallel programs *without* removing performance?*
DMP from 10,000’

• We only care about communicating instructions
• Deterministic serialization → same communication
  • …but what about performance?
• Recover parallelism from non-communicating insns
## Related Work on Determinism

<table>
<thead>
<tr>
<th>Helps with…</th>
<th>Record + Replay</th>
<th>DMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>…debugging?</td>
<td>☐</td>
<td>☺</td>
</tr>
<tr>
<td>…testing?</td>
<td>☐</td>
<td>☺</td>
</tr>
<tr>
<td>…replicas?</td>
<td>☐</td>
<td>☺</td>
</tr>
<tr>
<td>…deployment?</td>
<td>☐</td>
<td>☺</td>
</tr>
<tr>
<td>Needs hw?</td>
<td>usually</td>
<td>yes (for now)</td>
</tr>
</tbody>
</table>

**examples:** FDR, ReRun, Capo

- Deterministic parallel programming languages (Jade, DPJ, ...)
DMP Implementation Alternatives

PERFORMANCE

DMP-Serlia | DMP-SharingTable | DMP-TM | DMP-TMForward

COMPLEXITY
DMP-Serial Example
Can we do better?

• Only need to serialize communicating instructions

• Break each quantum into communication-free parallel prefix and communicative serial suffix
DMP-SharingTable

• Need to know *when communication happens*, to transition from parallel to serial mode
  • Leverage existing cache coherence protocol
  • When a line moves between processors, communication is (potentially) happening
  • The Sharing Table tracks information about ownership

• State of Sharing Table must **evolve deterministically**
  • Only allow updates during serial suffix
DMP-SharingTable Example

![Diagram of DMP-SharingTable Example]

- **Store A**
- **Store P**
- **Store B**
- **Load A**
- **Store Q**
- **Store B**

**Parallel Prefix**
- **Store A** to **Store P**
- **Store B** to **Load A**

**Serial Suffix**
- **Store B** to **Load A**
- **Store B** to **Store Q**

**Mode**
- Serial mode
- RAW
- WAW
Quantum Rounds

quantum

parallel prefix

serial suffix

race!
DMP-TM: Recovering Parallelism with Speculation

- DMP-SharingTable conservatively assumes that all changes in ownership are communication
  - ...but most changes in ownership are not communication (within round)

- Use TM support to speculate that a quantum is not involved in communication
  - If communication happens, rollback + re-execute

- Each quantum is an implicit transaction
  - Commit quanta in-order (need DT to commit)
DMP-TM Example
DMP-TM-Forward: Speculative Value Forwarding

• DMP-TM eliminates WAW and WAR dependencies
  • but cannot speculate past true (RAW) dependences

• Idea: speculatively forward values to “future” quanta
  • coherence protocol + ordered transactions make it easy to decide when and where to forward
  • rollback if a quantum’s speculatively read data is updated before the quantum commits
DMP-TM-Forward Example

we've recovered nearly all parallelism while maintaining deterministic execution
TM-Forward Rollback

Deterministic Order

store A
add
store A
add
load A
sub
store A
sub
add

Need value from youngest older store
TM-Forward Recursive Rollback

Deterministic Order

store A
add
store A

add
Load A
store B

add
sub
Load B
Better Quantum Building

- Any deterministic policy will work
- We want quanta that are free of communication
  - no communication $\rightarrow$ no serialization, no rollbacks
Experimental Methodology

• PIN-based simulator
  • Models serialization, quantum building, address conflicts and transaction rollbacks
  • Assumes constant IPC with free commits

• SPLASH2 and PARSEC benchmark suites
DMP Performance Results

- splash2 parsec benchmark suite
- Threads: 4, 8, 16
- DMP-Serial
- DMP-ShTab
- DMP-TM
- DMP-TMForward

Runtime normalized to nondeterministic parallel execution.
Supporting Debugging Instrumentation

- Need to allow instrumentation of programs and still keep the *same* original memory interleaving and behavior
  - so the programmer can attach a debugger and inspect state

- This requires conveying to the system what was added to the original code

\[\text{Diagram showing the process of instrumentation and debug support.}\]
Deploying Deterministic Execution

• Need to make it deterministic across machines too
• Same quantum formation
• Same order of deterministic token passing
If we have time...

• Software-only Sharing Table implementation
• Making execution deterministic across machines
• Dealing with nondeterminism from I/O and the OS

• Other interesting questions:
  • Security implications?
  • What does determinism mean in reactive (think DB server) applications?
  • How do we expose some of the DMP mechanisms to the programmer?
  • Determinism in race-free programs
  • Determinism in relaxed memory models
Current/Future Work

• Deterministic Multiprocessing
  • *software-only* implementation as a compiler+runtime (coming soon ;))
    • goal: preserve scalability, even if at the cost of higher absolute overhead
    • lots of static analysis and aggressive runtime system
  • evaluating impact in *testing*, dealing with “butterfly” effect
  • *system* issues (I/O, OS, etc.)
Luis Ceze, *University of Washington*

joint work with Owen Anderson, Tom Bergan, Joe Devietti, Brandon Lucia, Karin Strauss, Dan Grossman, Mark Oskin.

**samipa**  
Safe MultiProcessing Architectures  
at the University of Washington