BulkSC: Bulk Enforcement of Sequential Consistency

Luis Ceze, University of Washington

Safe MultiProcessing Architectures at the University of Washington

National Institute of Informatics, Tokyo
Memory Consistency Model

• Defines the values that a read can return

• Supported by the hardware

• Has major implications on programmability

Affects the whole software stack:
• applications
• OS, libraries, drivers
• compilers
• language semantics
Crash Course on Memory Models

- Discussion on Adve’s Tutorial
  - http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.53.4665

- Effect on programmability
  - who has to worry about it?

- Effect on performance
  - why are stricter models more expensive?

- A bit of history...
Sequential Consistency (SC)

Per-processor program order: memory operations from individual processors maintain program order

Single sequential order: the memory operations from all processors maintain a single sequential order

[λamport’79]
Problems with SC Enforcement

• Low Performance
  • restrictions on performance-enhancing reordering of memory operations

• Or Complex Implementation
  • buffer long history of speculative memory accesses
  • check this history against coherence events and cache displacements
  • coupled with key structures (LSQ, ROB, reg file, $)
  • typically fine-grain (instruction-level) undo

• Most current systems do not support SC

➡ We would like to change that!
➡ Support SC with simple hardware and high performance

[Gharachorloo’91]
[Ranganathan’96]
[Gniady’97, SC++]
**BulkSC: Bulk Enforcement of SC**

- Group instructions into Chunks, enforce SC only at Chunk granularity
  1. substantially reduce hardware complexity
  2. enable high performance
  3. retain programmability

- Execute a chunk **atomically** and in **isolation**, like a **single instruction**
**Chunk Execution: Atomicity and Isolation**

**Atomicity:** all updates in the chunk are made visible to other processors at once (all or nothing)

**Isolation:** a chunk should not see “outside” state changing during its execution
Requirements for Bulk Enforcement of SC

Per-processor program order: \textit{chunks} from individual processors maintain program order

Single sequential order: \textit{chunks} from all processors maintain a single sequential order
Bulk Operation [ISCA’06]

Addresses

[Bloom’70]

H

S

S₁ ∩ S₂

TM TLS SC

Bulk Framework

Signature Operations
Efficiently Operating with Chunks

- Key idea: Summarize in hardware addresses accessed by a chunk into a pair of signatures
  
<table>
<thead>
<tr>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>W</td>
</tr>
</tbody>
</table>

- Support signature operations with simple hardware
  - intersection, union, is-empty?, ...
  - much of the system operation is based on signatures

- At chunk commit
  - W signature is sent to other processors for disambiguation
  - R, W signatures are cleared
Chunk Atomicity and Isolation in Bulk

P1

Signatures | Cache
---|---
R₁ | A D
W₁ | B C

P2

Signatures | Cache
---|---
R₂ | B C
W₂ | F

commit

W₁ ∩ (W₂ ∪ R₂) ≠ ∅? (True)
squash!
**Program Chunk Order**

- Chunks are *arbitrarily* defined by the hardware
  - e.g. every 2000 dynamic instructions

- A processor commits chunks in *program order*

- High-performance:
  - Instructions inside a chunk arbitrarily reordered

- A processor can interleave the execution of instructions from 2 in-flight chunks
Single Sequential Order

• Need an Arbiter for chunk commits

• Naive:
  • allow a single commit at a time
Allowing Simultaneous Chunk Commits

- Conditions:
  - committing chunks’ memory operations should not intersect
    - both reads and writes
  - this can be efficiently enforced with chunk signatures
Arbitrating Simultaneous Chunk Commits

Write signatures stay in the arbiter until commit completes

\begin{align*}
W_1 \cap (R_2 \cup W_2) &= \emptyset? \quad \text{(True)} \\
W_1 \cap (R_3 \cup W_3) &= \emptyset? \quad \text{(True)} \\
W_2 \cap (R_3 \cup W_3) &= \emptyset? \quad \text{(False)}
\end{align*}
The whole process only uses signatures
Can support a distributed arbiter
Advantages of BulkSC

• **Simplifies the HW complexity of high-performance SC**
  - decouples consistency enforcement from core micro-architecture and caches
  - single-thread optimizations without worrying about multiprocessor issues
  - no associative structures in the processor
  - no extra bits in the cache for versioning [ISCA’06]

• **Memory ordering framework for MPs**
  - small extension to support TM, TLS, ...
Summary of Evaluation

• Cycle-accurate simulations [SESC]

- Result: SC with performance comparable to RC (1%) with little BW cost (5-13%)
- Much simpler hardware than SC++
Forward Progress

• Decrease chunk size in case of repeated squashes
• Perform pre-arbitration in the worst case
• Interaction with explicit synchronization:
  • it works fine, it is transparent
  • note that we chunk dynamic instructions, not static instructions
Also on the paper...

- Interaction with explicit synchronization, TM
- Forward progress
- I/O
- Distributed arbiter
- Directory design for signatures
- Two optimizations for private data
- Discussion on scalability
Scalability

Function of two factors:

• Ability to provide scalable arbitration
  • little communication per chunk commit
  • distributed arbiter

• No increase in false positives
  • longer chunks to tolerate longer latencies (might need larger sigs)
  • false positives in bulk disambiguations increase with the number of comparisons - not a big deal since directory acts as a filter
Why do We Really Need SC?

• Simple machine model
  • programmer can reason about the program more easily

• Consistency model affects the whole system stack

• Several programs synchronize with plain variables

• In any case, BulkSC offers same or better performance as RC
  • and it has other uses, TM/TLS, deterministic replay, etc..
Interaction with TM/TLS

• TM with BulkSC: just have chunks follow xaction markers
  • SC among transactions
  • and transactions wrt. non-transactional memory operations

• TLS: could potentially leverage the arbiter to impose task ordering
## Memory Models Across the Stack

<table>
<thead>
<tr>
<th>Language</th>
<th>Compiler</th>
<th>Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Strong</strong></td>
<td>reordering has to obey language model</td>
<td>reordering has to obey ISA or not be visible to SW</td>
</tr>
<tr>
<td>-not much room for reordering</td>
<td>-needs to insert fences to map language model to weaker ISA model</td>
<td>-room for reordering</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C++ program</th>
<th>Compiler</th>
<th>Assembly</th>
<th>Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Strong</strong></td>
<td>reordering has to obey language model</td>
<td>room for reordering</td>
<td>reordering may be visible to SW</td>
</tr>
<tr>
<td>-no worries about the HW reordering</td>
<td>-room for reordering</td>
<td>-room for reordering</td>
<td>-room for reordering</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Weak</strong></th>
<th>reordering may be visible to SW</th>
<th>-room for reordering</th>
</tr>
</thead>
<tbody>
<tr>
<td>-reordering has to obey ISA or not be visible to SW</td>
<td>-room for reordering</td>
<td>-room for reordering</td>
</tr>
</tbody>
</table>
Conclusions

• Presented BulkSC
  • coarse-grain, signature-based enforcement of SC

• Performance comparable to RC

• Simplicity: decouple consistency enforcement from processor design

• Independent of network ordering properties

• Generic ordering framework for speculative multiprocessors