BulkCompiler: High-Performance Sequential Consistency through Cooperative Compiler and Hardware Support

Wonsun Ahn, Shanxiang Qi, Marios Nicolaides, Josep Torrellas (UIUC)
Jae-Woo Lee, Xing Fang, Samuel Midkiff (Purdue University)
David Wong (Intel Corporation)
## Sequential Consistency (SC)

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T1 &amp; T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld x</td>
<td>st x</td>
<td>ld x</td>
</tr>
<tr>
<td>st y</td>
<td>st y</td>
<td>st x</td>
</tr>
<tr>
<td>ld z</td>
<td>st z</td>
<td>st y</td>
</tr>
<tr>
<td></td>
<td></td>
<td>st z</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ld z</td>
</tr>
</tbody>
</table>

- Accesses must appear to:
  - Interleave in a single total order
  - Follow per-thread program order
Why SC for All Codes (Even Racy Ones)?

• Much easier to debug:
  – Outcomes of loads easier to reason about
  – Debuggers can reproduce buggy interleaving
• Software correctness tools assume SC:
  – More relaxed memory models result in too many interleavings
• Under SC, semantics for data races are clear:
  – Easy specifications for safe languages (such as Java)
• Some system programmers code with data races for performance
  – Less chance of making mistakes due to memory ordering
**BulkSC**: High-Performance SC HW [ISCA ‘07]

- Chunk: dynamically formed group of instructions (e.g. 2000 instrs)
- Each chunk executed **atomically** and in isolation
- (Distributed) arbiter ensures a **total order** of chunk commits

---

High performance SC: Within a chunk:

- Instructions are fully reordered by HW
- Fences are no-ops in pipeline
SC Platform: Need SC HW + SC Compiler

- BulkSC → High performance SC HW
- For an SC language memory model: also need SC compiler
- Past work on compilation for SC [PPoPP ’05]:
  - Assumed no special HW support
  - Identified un-reorderable accesses & put fences
  - Always caused slowdowns due to
    - Lost compiler optimization opportunities
    - Hardware overhead of the fences
Example Compilation for SC (Conventional)

- sum, x, y are escaping variables

```
for(...) {
    sum += x + y;
}
```

```
int temp = x + y;
for(...) {
    sum += temp;
}
```

```
for(...) {
    mem_fence();
    sum += x + y;
}
```

- Loop invariant code motion introduces illegal ordering under SC
- May have to insert memory fences to prevent HW reordering
- Running on BulkSC:
  - Memory fences are not needed
  - But optimization is still illegal leading to lost opportunity
Contribution of this Work

• Given HW that supports SC through chunk-based execution
• Develop compiler that supports high performance SC by
  - Driving the chunk formation
  - Adapting code transformations to chunks
• Result:
  - Whole system, high-performance SC platform
  - Outperforms the relaxed Java Memory Model by avg 37%
  - Speedups come from code optimization within chunks
Rest of the Talk

• Mechanisms
• Infrastructure
• Results
Instructions Added to Drive Chunking

• *beginAtomic PC*
  - Starts new chunk and disables automatic chunking by HW
  - Takes as argument the PC of the Safe-Version of code
• *endAtomic&Cut*
  - Finishes the current chunk
  - Re-enables automatic chunking by HW
• *endAtomic*
  - Re-enables automatic chunking by HW

→ Atomicity allows compiler to ignore SC restrictions
Example Compilation for SC (BulkCompiler)

- sum, x, y are escaping variables

```java
for(...) {
    sum += x + y;
}
```

```java
beginAtomic;
for(...) {
    sum += x + y;
}
endAtomic;
```

```java
int temp = x + y;
for(...) {
    sum += temp;
}
endAtomic;
```

- HW guarantees atomic execution
- Compiler allowed to perform arbitrary optimizations inside
- If another thread accesses sum, x, y
  - HW detects failed speculation, squashes, and retries chunk
Transformations on Synchronization

- **Low-contention critical sections:**
  - Group many of them in same Atomic Region (AR)

beginAtomic

\[
\begin{aligned}
\text{acquire M1} & \quad i_1 \\
\text{release M1} & \quad i_2 \\
\text{acquire M2} & \quad i_3 \\
\text{release M2} & \quad i_4 \\
\end{aligned}
\]

\[\text{while (M1 == taken) \{ } \]

\[\text{while (M2 == taken) \{ } \]

endAtomic

\[
\begin{aligned}
\text{beginAtomic} & \quad i_1 \\
\text{while (M1 == taken) \{ } & \quad i_2 \\
\text{while (M2 == taken) \{ } & \quad i_3 \\
\end{aligned}
\]

\[
\begin{aligned}
\text{endAtomic} & \quad i_4 \\
\text{endAtomic} & \quad i_5 \\
\end{aligned}
\]

- Remove acquire / release
- Insert plain spins on lock variables
  - Lock may be owned
  - Owner will squash on release
- **Optimize** and reorder the code
- Critical sections need not be nested inside atomic regions
Transformations on Synchronization

- **Low-contention critical sections:**
  - Group many of them in same Atomic Region (AR)

```
beginAtomic
  i1
  acquire M1
  i2
  release M1
  i3
  acquire M2
  i4
  release M2
  i5
endAtomic
```

- **Remove acquire / release**
- **Insert plain spins on lock variables**
  - Lock may be owned
  - Owner will squash on release
- **Optimize** and reorder the code
- **Critical sections need not be nested inside atomic regions**

→ **Speedups relative to relaxed memory models**
Transformations on Synchronization (Cont’d)

- **High-contention** critical sections:
  - Tight-fit an atomic region
  - Reduce chance of squashes

\[
\text{beginAtomic}
\begin{align*}
\text{acquire M1} \\
\text{release M1} \\
\text{endAtomic}&\text{Cut}
\end{align*}
\]

\[
\text{beginAtomic}
\begin{align*}
\text{while (M1 == taken) } \\
\text{endAtomic}&\text{Cut}
\end{align*}
\]

→ Prevent slowdowns due to chunk squashes
Events Requiring Squashes of Atomic Regions

- One-time events
  - Data collisions with remote chunks
  - Interrupts
    ➔ Just retry the AR
- Recurring events (even after multiple retries)
  - Pathologically repeating data collisions
  - Cache overflows
  - Exceptions, page faults, system calls
    ➔ Jump to the Safe Version of the AR
Adding Safe Versions to Atomic Regions

- Create for each AR and pass PC as argument to `beginAtomic`
- Semantically a duplicate of the AR. But..
  - Does not use ARs to ensure SC (instead use fences)
  - Restricts compiler optimizations to SC only

⇒ HW guarantees forward progress in Safe Versions
BulkCompiler Algorithm

- Used Java Hotspot server compiler [JVM '01]
  - State-of-the-art commercial JVM
- Steps:
  1. Perform escape analysis and mark escaping references.
  2. Enclose each escaping reference in an atomic region.
  3. Expand each atomic region, merging when two meet.
     - Done to maximize compiler optimizations.
  4. Generate Safe Versions for all the atomic regions.
  5. Replace synch (CAS) operations with plain accesses.
Evaluation

• Compare two environments:
  – **Baseline:**
    • Unmodified Hotspot JVM
    • Conventional multicore (4 cores @ 4GHz w/ Simics)
    • Provides Java Memory Model to programmer
      – Java Memory Model is very relaxed (similar to RC)
  – **BulkCompiler:**
    • Hotspot JVM + BulkCompiler
    • BulkSC multicore (4 cores @ 4GHz w/ Simics)
    • Provides SC to programmer
  • Applications:
    – SPECjbb 2005, SPECjvm 98, JLex, Montecarlo
Average speedup is 37% (23% from compiler opts)
While still supporting SC
Conclusions

• Using chunk-based SC hardware, the compiler can:
  – Provide high performance SC, improving programmability
  – Achieve avg speedup of 37% over Java Memory Model
  – Speedups come from code optimization within chunks

• Atomic region support in HW enables new compiler optimizations

• Future work:
  – Study novel compiler optimizations
  – Apply to other memory models
BulkCompiler: High-Performance Sequential Consistency through Cooperative Compiler and Hardware Support

Wonsun Ahn, Shanxiang Qi, Marios Nicolaides, Josep Torrellas (UIUC)
Jae-Woo Lee, Xing Fang, Samuel Midkiff (Purdue University)
David Wong (Intel Corporation)
Tolerating Deadlocks

- HW detects deadlock using timeout mechanism
  - When instruction count of chunk becomes very large
- Squash offending chunk and jump to Safe Version
# Characterization

<table>
<thead>
<tr>
<th>Application</th>
<th>% of Dyn Instructions in ARs</th>
<th># of Dynamic ARs</th>
<th>Dyn AR Size</th>
<th># Sync Blocks per AR</th>
<th>Write Footprint (Lines)</th>
<th>Read Footprint (Lines)</th>
<th>% Instructions in AR Squashed</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECJBB05</td>
<td>44.5</td>
<td>323086</td>
<td>19117.2</td>
<td>212</td>
<td>489.4</td>
<td>865.6</td>
<td>0.79</td>
</tr>
<tr>
<td>JVM-db</td>
<td>75.8</td>
<td>22451</td>
<td>119176.0</td>
<td>2000</td>
<td>84.4</td>
<td>3123.0</td>
<td>0.40</td>
</tr>
<tr>
<td>JVM-jack</td>
<td>29.5</td>
<td>2382</td>
<td>30105.2</td>
<td>792</td>
<td>119.7</td>
<td>229.4</td>
<td>1.31</td>
</tr>
<tr>
<td>JVM-jess</td>
<td>62.6</td>
<td>33995</td>
<td>43475.6</td>
<td>102</td>
<td>141.1</td>
<td>449.7</td>
<td>0.27</td>
</tr>
<tr>
<td>JVM-raytrace</td>
<td>85.8</td>
<td>61419</td>
<td>19771.1</td>
<td>0</td>
<td>51.7</td>
<td>613.9</td>
<td>0.10</td>
</tr>
<tr>
<td>JVM-mtrt</td>
<td>77.5</td>
<td>61627</td>
<td>19589.0</td>
<td>0</td>
<td>305.5</td>
<td>1297.0</td>
<td>0.14</td>
</tr>
<tr>
<td>JVM-compress</td>
<td>92.7</td>
<td>1632082</td>
<td>5418.6</td>
<td>0</td>
<td>28.1</td>
<td>144.5</td>
<td>0.04</td>
</tr>
<tr>
<td>JLex</td>
<td>97.4</td>
<td>45846</td>
<td>131474.0</td>
<td>317</td>
<td>426.9</td>
<td>705.7</td>
<td>0.91</td>
</tr>
<tr>
<td>MonteCarlo</td>
<td>99.9</td>
<td>16778</td>
<td>82535.1</td>
<td>2000</td>
<td>11.0</td>
<td>13.0</td>
<td>0.34</td>
</tr>
<tr>
<td>Average</td>
<td>74.0</td>
<td>244407</td>
<td>52295.8</td>
<td>602</td>
<td>184.2</td>
<td>826.9</td>
<td>0.48</td>
</tr>
</tbody>
</table>
Execution in Chunks (Atomically & In Isolation)

Wonsun Ahn
The BulkCompiler