Speculative Synchronization: Applying Thread-Level Speculation to Explicitly Parallel Applications
José F. Martínez and Josep Torrellas
Dept. of Computer Science, University of Illinois, 2002

Presented by: David Itten
Agenda

- Thread-Level Speculation
- Introduction to Speculative Synchronisation
- Hardware needed
- Using Speculative Synchronisation
- Evaluation
Thread-Level Speculation TLS

- One safe thread
- Extracts speculative threads from serial code
- Threads go into potentially unsafe program sections
- Epoch numbers, lowest epoch number is safe thread
- Unsafe memory state in buffer
while (continue_condition) {
    ...
    x = hash[index1];
    ...
    hash[index2] = y;
    ...
}
Speculative Synchronization

- Execute code past active barriers, busy locks and unset flags
- Extra concurrency in presence of conservatively placed synchronisation
- Apply TLS concept to explicitly parallel applications
  - No ordering of speculative threads
Speculative Synchronization Unit (SSU)
Speculative Synchronization - Process

- Lock request
- Set Aquire / Release bits in SSU
- Lock free
- SSU spins on Lock var
- Lock acquired?
- Commit Cached var values
- Squash Spec / Receiving thread
- Cont. Speculative execution
- Conflict?
- Restart thread
- Set SSU in idle state
- Cont. Execution after CS
- Release lock
- Execute critical section
- Execute CS speculatively
- Safe thread
- Speculative thread(s)
Speculative execution

- Checkpoint the execution
  - Backup register states
- Processor hints for all memory accesses
- Set speculative bit in cache
  - Write back cache content to memory if dirty in all caches
Access conflicts

- Speculative thread receives a message for cache lines marked speculative
  - Squash receiver of message
  - Safe thread never squashed!

- Squash procedure:
  - Invalidate all dirty cache lines with speculative bit
  - Clear all speculative bits
  - Restore check pointed register state
  - Restart thread
SSU states

- Aquire / Release flags set
- SSU Active

⇒ Thread is executing speculatively CS
SSU states

- Acquire flag set
- SSU Active

➤ Thread already left the CS, executes code after the CS and wants to commit its values
**SSU states**

- Release flags set
- SSU Inactive

⇒ This is the safe thread, executing the CS
Speculative Flags

- Release bit kept clear (Release while speculative)
- Speculatively execute code after the flag
- Barriers can be built using flags and locks
Potential problems

- No free cache lines available: Stall execution
- Second lock:
  - Handle second lock like a normal variable
  - Wait for lock until thread becomes safe or lock is released
- Exceptions: Speculative threads are always squashed
- Irreversible actions (e.g. I/O access)
Evaluation

- NUMA processor architecture
- 16 or 64 nodes
- L1 and L2 write back caches
- 5 concurrent applications used for test
  - Hand-parallelized
  - Parallelizing compiler
  - Annotated applications which are transformed to parallel code
Evaluation

<table>
<thead>
<tr>
<th>Application</th>
<th>Description</th>
<th>Parallelization</th>
<th>Data Size</th>
<th>Processors</th>
<th>Barriers/Locks</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPLU</td>
<td>LU factorization</td>
<td>Compiler</td>
<td>Reference</td>
<td>16</td>
<td>Yes/No</td>
</tr>
<tr>
<td>Bisort</td>
<td>Bitonic sort</td>
<td>Annotations</td>
<td>16K nodes</td>
<td>16</td>
<td>Yes/No</td>
</tr>
<tr>
<td>MST</td>
<td>Minimum spanning tree</td>
<td>Annotations</td>
<td>512 nodes</td>
<td>16</td>
<td>Yes/Yes</td>
</tr>
<tr>
<td>Ocean</td>
<td>Ocean simulation</td>
<td>Hand</td>
<td>258x258</td>
<td>64</td>
<td>Yes/Yes</td>
</tr>
<tr>
<td>Barnes–Fine</td>
<td>N–body problem</td>
<td>Hand</td>
<td>16K particles</td>
<td>64</td>
<td>Yes/Yes(2048)</td>
</tr>
<tr>
<td>Barnes–Coarse</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Yes/Yes(512)</td>
</tr>
</tbody>
</table>

![Graph showing execution time for various benchmarks](image-url)

- Overhead
- Squashed
- Sync
- Useful (Speculative)
- Useful (Safe)
Evaluation

<table>
<thead>
<tr>
<th>Application</th>
<th>Description</th>
<th>Parallelization</th>
<th>Data Size</th>
<th>Processors</th>
<th>Barriers/Locks</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPLU</td>
<td>LU factorization</td>
<td>Compiler</td>
<td>Reference</td>
<td>16</td>
<td>Yes/No</td>
</tr>
<tr>
<td>Bisort</td>
<td>Bitonic sort</td>
<td>Annotations</td>
<td>16K nodes</td>
<td>16</td>
<td>Yes/No</td>
</tr>
<tr>
<td>MST</td>
<td>Minimum spanning tree</td>
<td>Annotations</td>
<td>512 nodes</td>
<td>16</td>
<td>Yes/Yes</td>
</tr>
<tr>
<td>Ocean</td>
<td>Ocean simulation</td>
<td>Hand</td>
<td>258x258</td>
<td>64</td>
<td>Yes/Yes</td>
</tr>
<tr>
<td>Barnes-Fine</td>
<td>N-body problem</td>
<td>Hand</td>
<td>16K particles</td>
<td>64</td>
<td>Yes/Yes(2048)</td>
</tr>
<tr>
<td>Barnes-Coarse</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Yes/Yes(512)</td>
</tr>
</tbody>
</table>

**Graph: Time Lost to Sync (%)**

- APPLU: 6.4%
- Bisort: 12.2%
- MST: 46.2%
- Ocean: 14.8%
- Barnes Fine: 15.4%
- Barnes Coarse: 21.5%
- Average: 19.4%

Legend:
- Overhead
- Squash (2nd Lock)
- Squash (False Data)
- Squash (True Data)
- Squash (Lock Sync)
- Barrier Sync
Conclusion

- Faster parallel execution for free
- Requires a hardware modification
- Barriers are still a problem
Questions