Lock Oscillation: Boosting the Performance of Concurrent Data Structures

Panagiota Fatourou  
FORTH ICS & University of Crete

Nikolaos D. Kallimanis  
FORTH ICS
The Multicore Era

- The dominance of Multicore Machines necessitates the development of efficient parallel software.
- Parallelism may be inefficient due to synchronization costs of parts that cannot be parallelized.
- Need for efficient synchronization mechanisms with low cost.
The cost of Synchronization

Synchronization requests (e.g. accesses to the same shared data) must be executed in mutual exclusion.

- Best time to execute \( m \) such requests \( \geq \) time required by a single thread to execute them, sequentially, sidestepping the synchronization protocol.

Ideally:

- One thread undertakes the task to execute all \( m \) synchronization requests.
- The rest of the threads execute only their local workload.

In practice:

- This is never the case: contention effects may have a drastic impact in performance.
The Basics of the Combining Technique

- Combining technique significantly enhances the performance.
- Each thread announces its operation by appending a node in the list.
- A thread attempts to become a combiner and serve, in addition to its own request, active requests by other threads.
- A thread that wants to perform a synchronization operation:
  1. It announces its requests,
  2. either try to become the combiner (not always “successfully”)
  3. or perform local spinning until the combiner performs their requests.
- The combiner applies, in addition to its operation, other announced operations before releasing the lock.

P. Fatourou & N. D. Kallimanis
Related Work

Combining Synchronization Protocols

Blocking:
- Oyama Algorithm: Oyama, Taura, and Yonezawa, PDSIA’99.

Wait-Free:
- P-Sim: Fatourou and Kallimanis, SPAA ’11.

Other synchronization protocols have lower or similar performance as CC-Synch.
Why performance is so low compared to ideal?
Why performance is so low compared to ideal?

- For announcing requests:
  1. At least one cache line is invalidated.

- For serving requests:
  2. A cache miss is caused to the combiner for reading a request and its arguments.
  3. Combiner causes at least one cache line invalidation for waking up each requesting thread.
  4. Requests are usually not placed on consecutive addresses → the prefetcher does not help.

```
List of synchronization requests
```

```
req₁  req₂  req₃  req₄
```

```
P. Fatourou & N. D. Kallimanis
```
Is it possible to further improve the performance?
Our Contribution

- **Osci** enables **batching on a single node**, the synchronization requests initiated by **multiple threads** running on the same core.

- A fat node contains more than one requests and is appended to the list by performing just a single expensive atomic operation.
  1. More requests are announced with less remote cache line invalidations.
  2. With a single cache miss, combiner efficiently applies more than one requests.
  3. More than one requesting threads wake up with one cache line invalidation.
  4. Processor's prefetcher handles the reading of announced requests more efficiently.

- When OSCI is combined with cheap context switching (i.e. user-level threads) performs extremely well.
- It outperforms by far all previous state-of-the-art synchronization algorithms.

P. Fatourou & N. D. Kallimanis
Our Contribution II

We discuss PSimX, a simple variant of PSim with highly upgraded performance.
- It ensures wait-freedom.
- Its performance is much closer to the ideal than that of PSim.
- Based on PSimX, it is straightforward to implement useful complex primitives (e.g. CAS on multiple words, etc.) in a wait-free manner, at a very low cost.

We built concurrent queues based on OSCI and PSimX which outperform all state-of-the-art concurrent queue implementations.

We built concurrent stacks based on OSCI and PSimX which outperform all state-of-the-art concurrent stack implementations.
The OSCI Synchronization Technique – General Idea

- Osci maintains:
  - a linked list of nodes that store synchronization requests
  - the shared variables implementing the simulated state
  - Each node of the list contains the requests announced by multiple active threads running on the same single core.
  - This “fat” node is appended in the list by performing a single expensive synchronization primitive (i.e. SWAP).
  - One of the threads that have announced requests in the head node of the list plays the role of the combiner.
Each thread initially allocates two nodes.

The first thread (or director) among those running on the same core, that wants to apply a request, successfully installs (i.e. successful CAS) a node to *Announce*.

After director has recorded its request:
- door: LOCKED → OPEN
- calls *Yield* to allow other threads running on the same core

All other threads on the same core:
1. run their computation,
2. eventually announce their requests, and
3. call *Yield*.

Whenever the director is rescheduled:
- door: OPEN → CLOSED
- Announces the node to the list of requests.
- Director is the only thread that can later the role of combiner.
The OSCI Synchronization Technique – Combiner’s side

<table>
<thead>
<tr>
<th>Tail</th>
</tr>
</thead>
<tbody>
<tr>
<td>next</td>
</tr>
<tr>
<td>door</td>
</tr>
<tr>
<td>&lt;request, ret, completed, locked&gt;</td>
</tr>
<tr>
<td>&lt;request, ret, completed, locked&gt;</td>
</tr>
<tr>
<td>&lt;request, ret, completed, locked&gt;</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

P. Fatourou & N. D. Kallimanis

- A combiner serves the requests of the list.
- After applying a request of some thread, it unlocks the thread by setting \(\langle \text{completed} = \text{true}, \text{locked} = \text{false} \rangle\).
- Whenever, the combiner thread gives up its role identifies the director from the next node (if any) to be the new combiner.
- If the list is non-empty, then there is exactly one combiner. If the list is empty, then no combiner exists.
Performance Evaluation I

- Osci outperforms CC-Synch by a factor of up to 11.
- The performance advantages of Osci over all other algorithms are even higher.
- PSimX outperforms all algorithms other than Osci.
Performance Evaluation II

Concurrent Queues based on Osci and PSimX outperform:
- LCRQ (Morrison & Afek ‘13)
- CC-Queue (Fatourou & Kallimanis ‘12)
- SimQueue (Fatourou & Kallimanis ‘11)
- MS-Queue (Michael & Scott ‘96)
- Two-locks queue (Michael & Scott ‘96)

Concurrent Stacks based on Osci and PSimX outperform:
- CC-Stack (Fatourou & Kallimanis ‘12)
- SimStack (Fatourou & Kallimanis ‘11)
- CLH-Stack
- Lock-Free stack (Treiber ‘86)
Performance Analysis

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>cache misses (all levels)</th>
<th>cycles spent in backend stalls</th>
<th>combining degree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Osci-x64</td>
<td>0.20</td>
<td>247</td>
<td>1404</td>
</tr>
<tr>
<td>Psim-x64</td>
<td>0.24</td>
<td>2306</td>
<td>1307</td>
</tr>
<tr>
<td>H-Synch-x32</td>
<td>0.47</td>
<td>666</td>
<td>32</td>
</tr>
<tr>
<td>CC-Synch</td>
<td>0.47</td>
<td>4210</td>
<td>1079</td>
</tr>
<tr>
<td>PSim</td>
<td>0.4</td>
<td>14300</td>
<td>22</td>
</tr>
</tbody>
</table>

➢ Osci spends the lowest amount of cache misses per operation.
➢ The cpu cycles spent in backend stalls per operation are the lowest.
➢ Osci achieves the highest combining degree.
➢ PSimX also spends a low amount of cache misses per operation and achieves high combining degree.
Thank You