The Adaptive Priority Queue with Elimination and Combining

Irina Calciu, Hammurabi Mendes, Maurice Herlihy
Brown University
Scalability in the Multicore Age

- Our machines are getting bigger, with more cores
- Scalability is far from ideal
- Synchronization is expensive
- We need better data structures for these new architectures
Priority Queue

• Abstract data structure
• Stores <key, value> pairs, where keys are priorities
• Interface: synchronous add(x) and removeMin()
• Implementation: heap or skiplist
• Usage: e.g. resource management
Lazy and Lockfree Skiplists (prior work) – Add()
Lazy and Lockfree Skiplists (prior work) – RemoveMin()

Bottleneck: contention on remove operations

[Lotan2000, Sundell2003]
Flat Combining (prior work)

[Thread]
[OMB REQ]

[Thread]
[OMB REQ]

[Thread]
[OMB REQ]

[Thread]
[OMB REQ]

Combiner

[Hendler2010]
Flat Combining (prior work) – RemoveMin()
Flat Combining (prior work) – RemoveMin()
Flat Combining (prior work) – RemoveMin()
Flat Combining (prior work) – Add()

Add operations are sequential
Flat Combining (prior work) – Add()
Goal (1): Combining + Parallel Adds

- **Head (Dummy)**
- **Combined Operations**
- **Parallel Adds**
- **Tail (Dummy)**
Goal (2): Parallelize Combined Adds Too

Head (Dummy) -> Combined Removes Parallel Adds ? -> Parallel Adds -> Tail (Dummy)
Stack and Queue Elimination (prior work)

[Hendler2004, Moir2005]

Elimination Array

Data Structure (Stack or Queue)
Parallelize Combined Adds Too: Use Elimination

Head (Dummy)  

Tail (Dummy)

Combined Removes Parallel Adds?
The Priority Queue at a Glance

• Elimination

• RemoveMin and small-value Add combining

• Large-value Add parallelism
Implementation: Elimination

Elimination Array

Add(x)

Remove()

Priority Queue

Head (Dummy)

Tail (Dummy)

Min
Implementation: Elimination

Elimination Array

Add(x)

Remove()

Priority Queue

Head (Dummy)

Tail (Dummy)

Min
Implementation: Combining

Elimination Array

rem, 0
y, stamp
x, stamp

Priority Queue

Add(x)
Remove()

server

Min

Head
(Dummy)

Tail
(Dummy)
Transitions of a Slot in the Elimination Array

(EMPTY, 0) → (REMREQ, stamp) → (INPROG, 0) → (val, 0) → (val, stamp) → (INPROG, 0) → addSeq() → (TAKEN, 0) → (TAKEN, 0)
Transitions of a Slot in the Elimination Array

(EMPTY, 0) -> (val, stamp) via Add
(remSeq()) -> (INPROG, 0) -> (val, 0) via Server
(val, stamp) -> (INPROG, 0) via Remove
(addSeq()) -> (TAKEN, 0) via Server
(EMPTY, 0) -> (TAKEN, 0) via Remove
Transitions of a Slot in the Elimination Array
Transitions of a Slot in the Elimination Array
Implementation: Parallel Adds

Elimination Array

Add(x)
Remove()

server

rem, 0 y, stamp x, stamp

Combined Operations
Parallel Adds

Head (Dummy) Tail (Dummy) Head (Dummy) Tail (Dummy)
Adaptive PQ Split: moveHead()
Adaptive PQ Split: moveHead()
Adaptive PQ Split: chopHead()
Adaptive PQ Split: chopHead()
Synchronization

- MoveHead() and chopHead() change the parallel skiplist
- We need to synchronize server and parallel adds
- Use RW Lock
- Server: acquire writeLock for moveHead() and chopHead()
- Parallel adds: acquire readLock
Synchronization

• Single writer lock

• Writer preference

• Implementation: based on timestamps
  • Server increments timestamp for moveHead() and chopHead()
  • Don’t hold the lock for the whole time of the parallel add
  • Do a clean find first (as verified by the timestamp)
  • Acquire read lock and finish the insertion
Linearizability - Elimination
Linearizability - Combining
50% Add Operations 50% RemoveMin Operations

Throughput vs Threads graph showing different operations like Add, RemoveMin, and their efficiency across varying thread counts. The graph indicates performance trends for different data structures such as pqe, fcpairheap, fcskiplist, and lazyskiplist.

Percentage breakdown for Parallel, Server, and Elimination operations in bar charts.
80% Add Operations 20% RemoveMin Operations

Throughput (Ops/ms)

<table>
<thead>
<tr>
<th>Threads</th>
<th>pqe</th>
<th>fcpairheap</th>
<th>fcskiplist</th>
<th>lazyskiplist</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>48</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>54</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Parallel Server Elimination

Add Operations
- 80%
- 20%

RemoveMin Operations
- 100%
Impact of Maintaining Two Skiplists

<table>
<thead>
<tr>
<th>Add() percentages</th>
<th>% moveHead()</th>
<th>% chopHead()</th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
<td>0.24%</td>
<td>0.03%</td>
</tr>
<tr>
<td>50</td>
<td>0.32%</td>
<td>0.01%</td>
</tr>
<tr>
<td>20</td>
<td>0.00%</td>
<td>0.00%</td>
</tr>
</tbody>
</table>
Hardware Transactions - Motivation

• RW Lock can be too expensive

• Use hardware transactions

• Intel TSX

• Speculative execution
Hardware Transactions (1)

• Naïve version
• Start a transaction
• Find + Insert
• End transaction
• Too many aborts
Hardware Transactions (2)

• Timestamp approach

• Server increments timestamp for moveHead() and chopHead()

• Find executes non-transactionally but has to be restarted if timestamp changes

• Insert executed in a transaction

• Read the timestamp in the transaction
Using Hardware Transactions

![Graph showing throughput for 80% Add operations and 20% RemoveMin operations with different thread counts.](image)
Using Hardware Transactions
Transactions Stats for 50% Add() and 50% RemoveMin()

<table>
<thead>
<tr>
<th>Working Threads</th>
<th>Transactions per successful operation</th>
<th>Fallbacks per total operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.01</td>
<td>0.00%</td>
</tr>
<tr>
<td>2</td>
<td>2.34</td>
<td>0.51%</td>
</tr>
<tr>
<td>3</td>
<td>3.21</td>
<td>1.73%</td>
</tr>
<tr>
<td>4</td>
<td>3.31</td>
<td>2.12%</td>
</tr>
<tr>
<td>5</td>
<td>3.46</td>
<td>2.74%</td>
</tr>
<tr>
<td>6</td>
<td>3.46</td>
<td>2.67%</td>
</tr>
<tr>
<td>7</td>
<td>3.61</td>
<td>3.25%</td>
</tr>
</tbody>
</table>
Summary

• First elimination algorithm for a priority queue

• Use two skiplist to separate small adds from large value adds

• Combining + Parallel Adds + Elimination

• HTM simplified the algorithm and improved performance
cs.brown.edu/~irina

cs.brown.edu/~hmendes
Transactions Stats for 3 Working Threads, 1 Server Thread

<table>
<thead>
<tr>
<th>Add percentage</th>
<th>Transactions per successful operation</th>
<th>Fallbacks per total operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1.32</td>
<td>0.00%</td>
</tr>
<tr>
<td>80</td>
<td>1.77</td>
<td>0.01%</td>
</tr>
<tr>
<td>60</td>
<td>2.37</td>
<td>0.29%</td>
</tr>
<tr>
<td>50</td>
<td>3.22</td>
<td>2.01%</td>
</tr>
<tr>
<td>40</td>
<td>3.64</td>
<td>5.24%</td>
</tr>
<tr>
<td>20</td>
<td>3.92</td>
<td>10.34%</td>
</tr>
<tr>
<td>0</td>
<td>1.09</td>
<td>0.00%</td>
</tr>
</tbody>
</table>