INTERRUPTING SNAPSHOTS
AND THE JAVA SIZE() METHOD

Moran Tzafrir
Tel-Aviv University

Joint work with Yehuda Afek and Nir Shavit
Concurrent Data-Structures

- Multicore machines are the mainstream computing platform.

- Inter thread communication is the bottleneck to application performance.

- Keeping concurrent data-structures scalable is key to applications’ scalability.
Concurrent Data-Structure Libraries

- Operations inherited from the sequential API

- Typical global-operations:

  ADT: LINKED-LIST
  - Insert()
  - Remove()
  - Find()
  - Size()
  - isEmpty()
  - Clear()

  ADT: QUEUE
  - Enq()
  - Deq()
  - Peek()
  - Size()
  - isEmpty()
  - Clear()

  ADT: RB-TREE
  - Insert()
  - Remove()
  - Find()
  - Size()
  - isEmpty()
  - Clear()

- Global-operations, show little or no scalability.
The Size() Operation

- \textit{Size()} linearizable operation added to any existing data-structure.

- Interface:
  - \textit{Update()} - inc/dec with add/delete of elements.
  - \textit{Size()} - get the number of elements in the data-structure.

- \textit{Update()} more frequent than \textit{Size()} by order of magnitude.
### Known Size() Solutions

<table>
<thead>
<tr>
<th></th>
<th>Scalable</th>
<th>Update Performance</th>
<th>Size Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Global-Counter + CAS</strong></td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Local-Counters + LOCKs</strong></td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Weaken the spec (JDK)</strong></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Interrupting-Snapshots

- New `Size()` algorithm based on atomic-snapshots.

- Unlike existing solutions:
  - Linearizable (and not an approximation)
  - Scalable
  - Wait-free

- Potentially first example of using atomic-snapshots in industrial code.
Local Counters + Snapshots

- **Local Counters**: one per thread.
- \( \text{Size}() = \sum_{i=1}^{n} \) (snapshot of local-counters)
- We need scalable atomic-snapshots

```
current  10  20  5
```

Scan()  Update()  35
Atomic Snapshot History

- Fundamental Problem: PODC, DISC, SPAA, STOC, FOCS
- Best in Theory: Attiya, Rachman, 1988
  - Update() - $O(n \log n)$ read/write
  - Scan() - $O(n \log n)$ read/write
- Suggested practical: Riany, Shavit, Touitou, 1995
  - Update() - $O(1)$ read/write
  - Scan() - $O(n)$ costly CAS operations

We need Practical Size().

- Update() - $O(1)$ read/write
- Scan() - $O(1)$ CAS operations
Single Scanner Algorithm

**Update()** - If `(curr.seq == thread seq)`, update current value. Else copy current to previous with new seq before updating current.
**Single Scanner Algorithm**

**Update(22)**
(Read scan-seq #6)

<table>
<thead>
<tr>
<th>scan-seq</th>
<th>#6</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>current</th>
<th>10</th>
<th>22</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>curr.seq</td>
<td>#6</td>
<td>#6</td>
<td>#6</td>
</tr>
<tr>
<td>previous</td>
<td>8</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

**Update()** - If (curr.seq == my seq), update current value. Else copy current to previous with new seq before updating current.
Single Scanner Algorithm

Due to $\text{Scan()}$, $\text{scan-seq} = #7$

<table>
<thead>
<tr>
<th>scan-seq</th>
<th>#8</th>
</tr>
</thead>
</table>

```
current:  | 10 | 22 | 5  |
curr.seq: | #6 | #Ø | #6 |
previous: | 8  | 22 | 0  |
```

$\text{Update()}$ - If $(\text{curr.seq} == \text{my seq})$, update current value. Else copy current to previous with new seq before updating current.
Scan() - Increment scan-seq (a write). Go over the local-counters, and collect values with curr.seq < my seq
Multi Scanner - Lock Free

Scan() - Increment scan-seq using CAS. Collect values with curr.seq < my seq.

If (curr.seq > my seq) abort and restart (without incrementing the scan-seq).
Counter Example - Multi Scanner

Scan(), IncAndFetch(), Collect 10, 24

Update(3), (curr.seq < my seq(#10)) → copy

Scan(), IncAndFetch(), Collect 10, 24

current 10 24 4
curr.seq #60 #10 #8
previous $0 23 $
Multi Scanner - Wait Free

Try to CAS on last-view.

CAS on last-view. From [9, 37] to [12, 39]

Scan(), IncAndFetch(12)
Collect: 11, 24, 4

Last-view: 92
scan -seq: 12

Borrow last-view result, since happened in our interval. → wait-free
Complexity

- **Update()** - Wait-free $\mathcal{O}(1)$ read/write operations

- **Scan()** - Wait-free Amortized $\mathcal{O}(1)$ CAS operations
Performance Evaluation

Compare
- New Lock-free Size implementation (NLF)
- New wait-free (NWF)
- Java Concurrency Package (JDK)
- Using [RianyShavitTouitou] Snapshot (RST)

On
- Sun UltraSPARC T2 (64x)
- Azul Vega2 7200 series (~200x)
Performance Evaluation

1% Size()

SPARC T2 - Throughput as function of CPUs
1% SIZE; 5% add; 5% remove; 89% contain; 5 samples; table size 1M

ops/ms

JDK
NWF
NLF
RST

New Wait-free
New Lock-free
Performance Evaluation

10% Size()

New Wait-free

New Lock-free

SPARC T2 - SCAN Throughput as function of CPUs
10% SCAN; 5% add; 5% remove; 5 samples; table size 1M

ops/ms

CPUs
Performance Evaluation

Same behavior on Azul
measure **Update()** throughput

**Size()** is responsible for difference in throughput
Conclusion

- Global operations `Size()`, `IsEmpty()` can be both accurate and scalable.

- Fundamental atomic-snapshot theory problem finds practical application
Future Work:

- Uses for atomic-snapshots in TM, concurrent-applications, ...
- Extending the Interruption Snapshots algorithm to a view that is more than a single word.
Thanks for your time ....

Questions ....