Anirban Mitra

anmitra@cs.stonybrook.edu
Department of Computer Science
Stony Brook University

March 30, 2013
Memory Errors

- One or more Memory bits is read differently from last written values
- Causes can be hardware bit corruption, cosmic rays, corruption in path between memory and CPU
Figure: Even a single memory corruption can be catastrophic. A simple linked list with memory error, entire tail lost due to a single pointer corruption.
- Hardware ECC (Error Correcting Codes) chips can help
- But is expensive in terms of processing time and money
- Even ECC cannot correct every corruption
### Resilient Priority Queue

**Table:** Mean Time Between Failures

<table>
<thead>
<tr>
<th>Mem. size</th>
<th>Mean Time Between Failures</th>
</tr>
</thead>
<tbody>
<tr>
<td>512 MB</td>
<td>2.92 hours</td>
</tr>
<tr>
<td>1 GB</td>
<td>1.46 hours</td>
</tr>
<tr>
<td>16 GB</td>
<td>5.48 minutes</td>
</tr>
<tr>
<td>64 GB</td>
<td>1.37 minutes</td>
</tr>
<tr>
<td>1 TB</td>
<td>5.13 seconds</td>
</tr>
</tbody>
</table>

**Figure:** A field study by Google researchers
Resilient Algorithms

- Make the algorithm and data structure capable of dealing with memory errors
- Design for damage control, minimize the disasters
- Memory model assumes that at maximum $\delta$ corruptions throughout the runtime
- $\delta$ is an input to the model, known in advance
- Also, assumes that it has some constant amount of reliable memory $P = O(1)$
- Reliable memory cannot get corrupt
Faithful Ordering

- In resilient sorting, recovery from corruption is expensive.
- Hence goal is modified to all un-corrupted items are guaranteed to be correctly sorted.
- Order of corrupted items is ignored.

Corrupted memory hence ignored
Structure of Resilient Priority Queue

Levels

L[0]  U[0]  D[0]


s[2]

Buffer array to store elements

Up Buffers

Down Buffers
Structure of Resilient Priority Queue

- Intuitively, elements in Up buffer are moving to upper layers
- Elements in Down buffer are moving to lower layers
Structure of Resilient Priority Queue

s[i+1] = 2s[i]
double linked list pointers stored reliably

Hence up to level k = O(log N)
Invariant A

All buffers are faithfully ordered
Invariant B

\[ D[i] \leq D[i+1] \]
\[ D[i] \leq U[i+1] \]
\[ 0 \leq i < k \]

Means concat of \( D[i] \) and \( D[i+1] \) is faithfully ordered.
Invariant C - Down at least Half Filled

\[ \frac{s[i]}{2} \leq D[i] < s[i], \quad 0 \leq i < k \]
Invariant D - Up at most Half Filled

\[ \left| U[i] \right| \leq \frac{s[i]}{2} \]
\[ 0 \leq i < k \]
Insert

Inserts go into up buffers which can violate invariant D
Insert

Faithfully merge $U[i]$, $D[i]$ and $U[i+1]$

$t$ is size of $D[i]$
Put $(t - \delta)$ in $D[i]$
and rest in $U[i+1]$
$U[i]$ empty
Insert

- **Why \( t - \delta \)? Suppose \( D[i] \) is fully filled then due to error here**

- **Corruption**

- **No corruption**

- **Due to error invariant B is violated, there can be \( \delta \) such error, hence the size \( (t - \delta) \)**
Push and Pull

- *Push* and *Pull* primitives used for *Insert* and *DeleteMin*
- *Push* pushes elements to higher layers when Up at most half invariant is broken
- Similarly, *Pull* pulls elements from higher layer when Down at least half invariant is broken
Algorithm for Push

1: function \texttt{Push}(U_i)
2: \hspace{1em} d_i = |D_i|
3: \hspace{1em} M = U_i \cup D_i \cup U_{i+1}
4: \hspace{1em} D_i = M[0..(d_i - \delta)]
5: \hspace{1em} U_i = []
6: \hspace{1em} \textbf{if} \ i = k \textbf{ then}
7: \hspace{1em} \hspace{1em} U_{k+1} = []
8: \hspace{1em} \hspace{1em} D_{k+1} = M[(d_i - \delta)..]
9: \hspace{1em} \hspace{1em} k = k+1
10: \hspace{1em} \textbf{else}
11: \hspace{1em} \hspace{1em} U_{i+1} = M[(d_i - \delta)..]
12: \hspace{1em} \hspace{1em} \textbf{if} \ U_{i+1} > s_{i+1}/2 \textbf{ then}
13: \hspace{1em} \hspace{1em} \hspace{1em} \texttt{Push}(U_{i+1})
14: \hspace{1em} \hspace{1em} \textbf{if} \ D_i < s_i/2 \textbf{ then}
15: \hspace{1em} \hspace{1em} \hspace{1em} \texttt{Pull}(D_i)
Algorithm for *Pull*

1: function **Pull**\( (D_i) \)
2: \[ d_i = |D_i| \]
3: \[ d_{i+1} = |D_{i+1}| \]
4: \[ M = U_i \cup D_i \cup U_{i+1} \]
5: \[ D_i = M[0..s_i] \]
6: \[ D_{i+1} = [s_i..(d_{i+1} + d_i - \delta)] \]
7: \[ U_{i+1} = [d_{i+1} + d_i - \delta..] \]
8: if \( U_{i+1} > s_{i+1}/2 \) then
9: \hspace{1em} Push\( (U_{i+1}) \)
10: if \( D_{i+1} < s_{i+1}/2 \) then
11: \hspace{1em} Pull\( (D_{i+1}) \)
**Insert**

- For insertion we maintain a buffer \( I \) of size \((\delta + \log n + 1)\) and simply append the new element to buffer.
- When \( I \) is full we faithfully sort it and faithfully merge with \( U_0 \).
- Call \textit{Push} on \( U_0 \) if at most invariant is broken.
- For \textit{DeleteMin} return the minimum of the first \( \delta + 1 \) elements of \( D_0 \) and all of \( I \).
- If \( D_0 \) underflows, call \textit{Pull}.
Insert and DeleteMin takes $O(\log n + \delta)$ amortized time

Push and Pull on a buffer is called at most once each request

Intuitively, $\Omega(s_i)$ operations happen between any two call at any level $L_i$ which gives the amortized bounds

Lower bound is proved to be the same

Uses $O(n + \delta)$ space
Relation with Cache Oblivious

- This structure was inspired by cache oblivious priority queue by Bender et al in 2002
- There are several data structures which are adapted from their cache oblivious versions
- Even though resilient model does not have memory hierarchy
- One reason could be that cache oblivious structure use chunks of array to gain by locality and employs less pointers
- So they become amenable to be adapted to resilient version because then the small number of pointers can be stored in reliable memory
- Note that there is a tree based cache oblivious priority queue too
