Transactional Garbage
and how to collect it
for fun and profit

Fadi Meawad    Ryan Macnak    Jan Vitek

S3Lab
Computer Science Dept
Purdue University
C# STMBench7 on Bartok

8-core, 1.60GHz Intel Xeon E5310. 8GB RAM. Physical Address Extension enabled, running Windows Server 2003 SP2.
Up to 98% of time spent in GC

8-core, 1.60GHz Intel Xeon E5310. 8GB RAM. Physical Address Extension enabled, running Windows Server 2003 SP2.
What does this have to do with Transactional Memory?
Let’s Benchmark

**GCBench**
- New micro-benchmark, creates a linked list of lists
- A transaction traverses lists, at every node, either:
  - Update node
  - Allocate unreachable or live object
  - Allocate object unreachable after commit
  - Make object unreachable

**Wormbench**
- C#, designed for the Bartok STM
- A “worm” with a triangular head and a tail lives in a matrix with other worms
- 15 ops, such as move forward or turn right
Let’s Benchmark

**STMBench7**
- Trees, graphs and indices as in CAD/CAM workloads
- 500MB of data
- Configure either r-, r/w- or w-dominated workload
- Long or short traversals

**LeeTM**
- Automatic circuit routing using Lee’s algorithm
- Pairs of points on a grid connected with non-intersecting paths.
8-core, 1.60GHz Intel Xeon E5310. 8GB RAM. Physical Address Extension enabled. Windows Server 2003 SP2.
Where did that time go?

<table>
<thead>
<tr>
<th># of threads</th>
<th>list size</th>
<th>100</th>
<th>200</th>
<th>300</th>
<th>400</th>
<th>500</th>
<th>600</th>
<th>700</th>
<th>800</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>0%</td>
<td>9%</td>
<td>23%</td>
<td>36%</td>
<td>48%</td>
<td>54%</td>
<td>61%</td>
<td>67%</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>12%</td>
<td>35%</td>
<td>56%</td>
<td>69%</td>
<td>74%</td>
<td>82%</td>
<td>85%</td>
<td>88%</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>25%</td>
<td>53%</td>
<td>72%</td>
<td>79%</td>
<td>87%</td>
<td>91%</td>
<td>93%</td>
<td>95%</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>39%</td>
<td>66%</td>
<td>81%</td>
<td>88%</td>
<td>92%</td>
<td>95%</td>
<td>96%</td>
<td>97%</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>40%</td>
<td>75%</td>
<td>87%</td>
<td>92%</td>
<td>95%</td>
<td>96%</td>
<td>97%</td>
<td>98%</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>49%</td>
<td>81%</td>
<td>90%</td>
<td>95%</td>
<td>97%</td>
<td>98%</td>
<td>98%</td>
<td>99%</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>60%</td>
<td>82%</td>
<td>93%</td>
<td>96%</td>
<td>97%</td>
<td>98%</td>
<td>99%</td>
<td>99%</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>64%</td>
<td>85%</td>
<td>94%</td>
<td>97%</td>
<td>98%</td>
<td>99%</td>
<td>99%</td>
<td>99%</td>
</tr>
</tbody>
</table>

C# GCbench, % of time spent in GC
Maybe it is just in Bartok!
What is Bartok STM?

Optimizing ahead-of-time research compiler & runtime

**STM**
- Object-based, in-place, optimistic updates.
- Read-enlistment, update-enlistment & undo-value logs
  - Allocated from normal heap
- Transaction ID is stored in object's header

**GC**
- 2-generational semispace copying collector
- Stop the world

[Harris, Plesko, Shinnar, Tarditi, Optimizing memory transactions, PLDI06]
### GC % using Multiverse

<table>
<thead>
<tr>
<th>GC%</th>
<th>Unsynchronized</th>
<th>Multiverse</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>48%</td>
<td>19%</td>
</tr>
<tr>
<td>2</td>
<td>57%</td>
<td>32%</td>
</tr>
<tr>
<td>3</td>
<td>67%</td>
<td>38%</td>
</tr>
<tr>
<td>4</td>
<td>61%</td>
<td>42%</td>
</tr>
<tr>
<td>5</td>
<td>75%</td>
<td>48%</td>
</tr>
<tr>
<td>6</td>
<td>70%</td>
<td>57%</td>
</tr>
<tr>
<td>7</td>
<td>84%</td>
<td>66%</td>
</tr>
<tr>
<td>8</td>
<td>84%</td>
<td>76%</td>
</tr>
</tbody>
</table>

Java GCbench size 800, % of time spent in GC
### GC % using Multiverse

<table>
<thead>
<tr>
<th></th>
<th>GC%</th>
<th>Total time (seconds)</th>
<th></th>
<th></th>
<th>Slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Unsync'd</td>
<td>Multiverse</td>
<td>Unsync'd</td>
<td>Multiverse</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>48%</td>
<td>19%</td>
<td>10</td>
<td>99</td>
<td>1.9x</td>
</tr>
<tr>
<td>2</td>
<td>57%</td>
<td>32%</td>
<td>18</td>
<td>146</td>
<td>2.7x</td>
</tr>
<tr>
<td>3</td>
<td>67%</td>
<td>38%</td>
<td>20</td>
<td>184</td>
<td>3.4x</td>
</tr>
<tr>
<td>4</td>
<td>61%</td>
<td>42%</td>
<td>31</td>
<td>208</td>
<td>2.9x</td>
</tr>
<tr>
<td>5</td>
<td>75%</td>
<td>48%</td>
<td>32</td>
<td>245</td>
<td>3.6x</td>
</tr>
<tr>
<td>6</td>
<td>70%</td>
<td>57%</td>
<td>41</td>
<td>318</td>
<td>4.4x</td>
</tr>
<tr>
<td>7</td>
<td>84%</td>
<td>66%</td>
<td>44</td>
<td>450</td>
<td>6.8x</td>
</tr>
<tr>
<td>8</td>
<td>84%</td>
<td>76%</td>
<td>47</td>
<td>769</td>
<td>12.6x</td>
</tr>
</tbody>
</table>

Java GCBench size 800, % of time spent in GC
Does it depend on the GC?

![Graph showing execution time (seconds) vs. number of threads for different GC configurations. The graph compares Multiverse-parallel, Multiverse-concMS, Multiverse-serial, Unsync'd-concMS, Unsync'd-serial, and Unsync'd-parallel.]
Does the problem Scale?

Azul Vega 3 3310B, two 54-core processors. 48GB of RAM, Azul VM. Concurrent Pauseless GC
Does the problem Scale?

Heap Size

- SMA-Multiverse
- NoSMA-Multiverse
- NoSMA-UnSynced
- SMA-UnSynced

GCBench size 800, Azul
What can we do about it?
Logs in Bartok

Object-based in place updates with undo logs

Reads
The read-object log contains read STM Word (version #)

Updates
An object opened for update has an updated-object log entry (previous STM word)
Upon update, old value is maintained in an undo log

Chunks
Logs are allocated in chunks maintained by the STM
Discarded at end of transaction

[Read-object log entries:
Node1
| v100 | 0 | 00 |
---|---|---|
| List VTable |
| Head |
| Tail |
| Sum = 42 |
| Transaction manager |
| Offset in log chunk |

[Node1: v100 0 00
| Node VTable |
| Value = 10 |
| Next = null |

[undo log entries:
List |
| SCALAR_FIELD |
| Offset = 3 |
| Prev = 42 |

[Harris, Plesko, Shinnar; Tarditi, Optimizing memory transactions, PLDI 06]
What is log reuse, and is it enough?

Log Reuse

- At transaction end, preserve log chunks into a pool rather than leave for GC
- When a transaction needs a new log chunk, try the pool first, otherwise allocate

Issues

- Hard to decide when to deallocate log chunks
- Large initializing Txs followed by small ones will result a large unused pool
- The pool will be traced by the GC
- Weak references are expensive
Dedicated Nurseries

Generational-GC nurseries
- Most objects either die young or live forever

Transaction nurseries
- Objects allocated in transaction not visible to other threads until commit
- Reclaim nursery in one step after abort
- Can support nested transactions
- Finalization?
Dedicated Nurseries
Dedicated Heap

- Much of transactional allocation is logs
- Lifetime of logs bounded by the transaction they serve
- Known lifetime allows manual memory management
- Cheap to allocate/free in chunks from a mutexed freelist
Dedicated Heap

Logs Heap
- Read Chunk
  - Read Entry
    - ...
- Update Chunk
  - Update Entry
    - ...
- Undo Chunk
  - Undo Entry
    - ...
- Read Chunk
  - Read Entry
    - ...
- Read Chunk
  - Read Entry
    - ...

GC'd Heap
- Tx Manager
- Application Obj
- Application Obj
- Application Obj
- Tx Manager

Connections:
- Not traced
- Weak
- Weak
- Strong
Does it work?

8-core, 1.60GHz Intel Xeon E5310. 8GB RAM. Physical Address Extension enabled, Windows Server 2003 SP2.
Does it work?

C# STMBench7

# of threads

ops per second

LTD-Modified
LTD-Original

LTE-Modified
LTE-Original
Does it work?

C# STM Bench7

# of threads
% increase in ops per second

Long traversals enabled
Long traversals disabled
LeeTM, issues

Issues
- Allocates a temp grid within the transaction
- Bartok STM does not optimize multidimensional array access (excessive logging)

Workarounds
- Allocate before the transaction (Opt)
- Use RowMajor array access (RM)
Results (cont’d)

C# LeeTM

RM-Original

RM-Modified

Opt-Original

Opt-Modified

OptRM-Original

OptRM-Modified

# of threads

1 2 3 4 5 6 7 8

total time (seconds)

0 10 20 30 40 50 60 70 80 90 100 110 120

25
Running on an 8-core, 1.60GHz Intel Xeon E5310 with 8GB of RAM and Physical Address Extension enabled, running Windows Server 2003 SP2.
Conclusion

Memory Usage

- Same overall allocated memory
- Less demands on GCed heap

Speed Up

Applying to other systems

- Not for library based STM systems, but with runtime support will work with most STM flavors