Can Concurrent Data Structures Rely on Automatic Memory Managers?

Erez Petrank
Technion

MSPC/ISMM Keynote, Beijing, June 2012
A Disconnect

“Parallel” Community
• PODC
• SPAA
• DISC

Memory Management Community
• MSPC
• ISMM
• PLDI
• OOPSLA
• POPL
Agenda

• Concurrent data structures and progress guarantees.
• The advantages of garbage collection.
• Lock-free GC: what do we need?
• State of the art: some cool techniques we have.
• The missing pieces.
Concurrent Data Structures

- E.g. linked lists

Support: insert, delete, contains.
Sequential Operations

- Delete 6:

- Insert 7:
But don’t try this concurrently!

- Delete 6 || Insert 7:

- A similar problem with concurrent deletes or inserts.
Solutions

- **Coarse-grained locking.**
  - Sequential with overhead...

- **Fine-grained locking**
  - Hand-over-hand, optimistic, lazy synchronization.

- **Lock-free (or wait-free) implementation.**
Compare and Swap (CAS)

• CAS (addr, expected, new)

  Atomically:
  If ( MEM[addr] == expected ) {
    MEM[addr] = new
    return (TRUE)
  }
  else return (FALSE)
A Lock-Free Linked List

• To insert or delete a node execute a CAS on the desired Pointer.

• The insert-delete problem still exists.
Harris’s Lock-Free Linked List

- Use the least bit of a pointer to mark it.
Harris’s Lock-Free Linked List

- Use the least bit of a pointer to mark it.
- “Logically” delete 6, by marking its outgoing pointer.

Then, physically delete it.
Note that the delete-insert conflict disappears

- Either a CAS to insert 7 after 6 succeeds,
- Or a CAS to logically delete 6 succeeds,
  - The redirection of 4 → 6 happens only thereafter.
- But not both.
Some Thread Must Make Make Progress

- Between several concurrent CASes -- one must succeed.
- So one of the threads must make progress.
Progress Guarantees

Great guarantee! Until recently considered difficult to achieve and inefficient.

Lock-Freedom
If you schedule enough steps across all threads, one of them will make progress.

Wait-Freedom
If you schedule enough steps of any thread, it will make progress.

Obstruction-Freedom
If you let any thread run alone for enough steps, it will make progress.

Somewhat weak.
Which Paper Starts with the Following?

A shared data structure is lock-free if its operations do not require mutual exclusion. If one process is interrupted in the middle of an operation, other processes will not be prevented from operating on that object. In highly concurrent systems, lock-free data structures avoid common problems associated with conventional locking techniques, including priority inversion, convoying, and difficulty of avoiding deadlock.

A shared data structure is lock-free if its operations do not require mutual exclusion. If one process is interrupted in the middle of an operation, other processes will not be prevented from operating on that object. In highly concurrent systems, lock-free data structures avoid common problems associated with conventional locking techniques, including *priority inversion, convoying, and difficulty of avoiding deadlock (and live-lock).*
Is Limited Locking Good Enough?

• The space shuttle to Mars: *Mars Pathfinder 1997*.  
  – A low-priority thread was acquiring a lock for a very short period.  
  – A high-priority thread was OK with waiting for such period.  
  – But a medium-priority thread kicked out the low-priority thread while it was holding the lock!  
  – The high priority thread missed its real-time deadline and the system would restart.  
  – That happened repeatedly.  

• Problem: priority inversion  
  – Lock freedom ensures this does not happen...
What is it good for?

• Real-time, OS, interactive systems, service level agreements, etc.
• But it’s always good to have.
  – Avoid deadlock, live-lock, convoying, priority inversion, etc.
• Scalability.
Back to concurrent data structures

- What about reclaiming a node?
- Many papers rely on garbage collection.
- Manual reclamation is problematic.
Reclaiming a node

- T2 is in the middle of \textit{contains}(7).
- T7 is deleting and reclaiming 6.
Reclaiming a node

- T2 is in the middle of \textit{contains}(7).
- T7 is deleting and reclaiming 6.
Reclaiming a node

- T2 is in the middle of `contains(7)`.
- Next, T7 is inserting 8.
Reclaiming a node

• T2 is in the middle of contains(7).
• Next, T7 is inserting 8.
Reclaiming a node

- T2 is in the middle of contains(7).
- Next, T7 is inserting 8.
Reclaiming a node

- Now T2 wakes up and continues to erroneously determine that 7 is not in the list!

Contains may return a wrong answer!
Most Designs Assume Automatic GC

• Book: “our implementations rely on garbage collection to recycle nodes”

• Lock-free search tree [Ellen et al. PODC 2010]: “For simplicity we assume nodes and Info records are always allocated new memory locations.”

• Lock-free Skip list [Formitchev and Rupert]: “We have not explicitly incorporated a memory management technique…”

• etc.

But GC does not support lock-free algorithms!
Why do they use GC?

• Because it’s so easy.
• Because it’s orthogonal.
  – “Whenever lock-free GC is presented, things will fall into place.”
Manual Memory Management with 

Hazard Pointers [Michael PODC’02]

• Each thread announces objects that should not be reclaimed.
• A thread may reclaim an object only if it is not announced.
• Reminiscent of garbage collection, but very partial.
Easy? Not so fast...

- When moving a hazard pointer from one node to the next, need to verify that the next node is still relevant.
- Concern: between “read next” and “modify hazard pointer” there was a reclamation...

Hazard Pointers:

```
4 7 9 15 21 64
T1 T2 T3
```
Verifying each Modification

- Proper verification is tricky! Hazard pointers are difficult to get right.
- Proper verification is costly! Linked-list overhead > 25%, more for skip-list etc.

Hazard Pointers:

```
4 -> 7 -> 9 -> 15 -> 21 -> 64
```

Diagram:

```
T1 -> T2 -> T3
```

Legend:

- T1
- T2
- T3
Verifying each Modification

• Proper verification is tricky! Hazard pointers are difficult to get right.

• Proper verification is costly! Linked-list overhead > 25%, more for skip-list etc.

• And hazard pointers are the best we have for manual reclamation.

• Lock-free GC would be so much easier!
A Lock-Free Garbage Collector
What do we want?

- Application pauses shortly for memory management,
- Bounded memory consumption (must control fragmentation)
- GC must terminate before heap exhausted!

Formal definitions exist [P et al. PLDI 2009]
What do we have?

• Several papers claim lock-free GC...
  – The serious ones do not provide GC progress.
  – Some of them do not really achieve GC
  – Some have a different notion of lock-freedom.

• Different scenarios: Sapphire, Metronome, Azul, etc.
Three Goals

• Task 1: Thou Shalt Not Block Program!
• Task 2: Thou Shalt Not Waste Memory!
• Task 3: Thou Shalt Terminate!
Task 1: Thou Shalt Not Block Program!

- On stock hardware use:
  Lock-free mark-sweep
  - Basically: lock-free data structures and care for mark-stack overflow
    [Pizlo et al. ISMM’07]

- If you have DCAS, or guaranteed transactions you can also use:
  Lock-free reference counting [Detlefs et al. PODC’02]
  Lock-free copying [Herlihy and Moss SPAA’91]

- Note: no GC progress!
Task 2: Thou Shalt Not Waste Memory!

To avoid fragmentation:

• Use copying with DCAS [Herlihy and Moss SPAA’91]

• For stock hardware use compaction [Pizlo et al. ISMM’07, PLDI’08].
Moving Objects Concurrently

• The problem: two copies of the same object.
• Which version should be used?

The Heap

Program Thread

Original
New
The diagram illustrates the concept of copying fields in a program thread. The original object and its copy are connected by arrows indicating that the field has been copied. There is also a notation that some part has already been copied.
Problem: status may change before we actually write.
Need to atomically access two locations: field and status.
Three Avenues [Pizlo et al. PLDI’08]

• **Chicken**: aborts copying when program interferes.
  – Independently in [Auerbach et al. EMSOFT’08]
• **Clover**: locks with extremely small probability.
• **Stopless**: Use an intermediate wide object.
Chicken

• Idea: if program writes to an object during the move then the move is aborted.

• Collector: after copying an entire object, if it was not modified, atomically mark it “moved”.

![Diagram of object move]

1. Object
2. Move
3. Marked as "moved"
Chicken

• Idea: if program writes to an object during the move then the move is aborted.

• Collector: after copying an entire object, if it was not modified, atomically mark it “moved”.

• Program: start by atomically writing non-movable.
  • Check if the object moved.
  • Finally, write.
Chicken

- Advantage: very low overhead, lock-free.
- Disadvantage: may fail to move objects.
• Idea: Employ a (well defined) probabilistic approach.

• Choose uniformly at random a value $\pi$ of double-word length, assume $\pi$ is never written to the heap.

• For each write this happens with probability $2^{-128}$ or $2^{-64}$. 
Collector Use of a Random Marker $\pi$

- Read value in original; write it to new object.
- Atomically Write $\pi$ to original assuming the value has not changed.
- If failed, try again.

Program uses original until it sees $\pi$. 

Diagram:

Original object:

```
Header $\pi$ $\cdots$ $\pi$ $w$
```

New object:

```
Header $u$ $\cdots$ $u$ $\cdots$
```
 Clover -- Discussion

• Which probability is small enough to be acceptable for lock-free (or real-time) systems?
• The probability that a person dies from a car crash in a single day is higher than 1/2,500,000 (in the U.S.).
• A hardware failure is much more likely.
• So is it enough that the program meets its deadlines with such probability?
  • Clover fails to remain lock-free with prob. \( \sim 10^{-20} \). Negligible for all practical purposes.
  • Failure prob. is bounded independently of the input or of any other program run characteristics.
Clover -- Summary

• Assume a random string $\pi$ never gets written to the memory.
• Advantage: simple and fast.
• Disadvantage:
  – Does not guarantee GC progress.
  – Locks with a negligible probability.
## Comparison

<table>
<thead>
<tr>
<th></th>
<th>Non-Blocking</th>
<th>Time-Overhead</th>
<th>Space-Overhead</th>
<th>GC progress</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clover</td>
<td>With very high prob.</td>
<td>😊</td>
<td>😊</td>
<td>😞</td>
</tr>
<tr>
<td>Chicken</td>
<td>😊</td>
<td>😊 😊</td>
<td>😊</td>
<td>😞</td>
</tr>
</tbody>
</table>
Task 3: Thou Shalt Terminate

- GC must terminate before heap is exhausted.
- So GC must be sure to make progress.
- We do not know how to do it (efficiently).
- Let’s look at state of the art, and what is still missing.

- Three main things are missing:
  - Lock-free stack scanning with guaranteed progress.
  - Lock-free handshaking to start a (concurrent) collection.
  - Lock-free fragmentation control with guaranteed progress.
GC Progress 1: Lock-Free Stack Scanning

• Must scan each thread’s stack to get the roots.
• Traditionally, each thread scans its own stack and reports.
  – But a thread may fail to cooperate.
• Concurrent GC algorithms assume an atomic stack scan for each thread.
• Can we get it while the thread may be running?
Lock-Free Stack Scanning
[Kliot et al. VEE’2009]

• Place a “return barrier” [Yuasa et al 2002] on the current frame and then all upper frames can be scanned.
• Handle current frame carefully...

Code that:
• Scans previous stack frame,
• Save next return address and replace it with address of this code
• Return to previously saved address
Lock-Free Stack Scanning
[Kliot et al. VEE’2009]

- Caveat: it is assumed that the program either executes, or is in an identifiable dormant state.
- Program executes some cautionary code when returning from a dormant state.

Many more details:
- Contention between collector and program on frame scan,
- Frames change while collector scans them,
- Copying collectors,
- Pointers between stack frames (for C#),
- etc.
GC Progress 2: Program & Collector Cooperation

- Must “let the program know” when a collection starts:
  - No thread is “interacting” with the previous collection.
  - No write is “in the middle”.

- Most collectors use handshakes.
  - The collector raises a flag
  - Each thread raises a flag (outside a write operation).
If a thread is not responding...

• We cannot tell that it is ready for a new collection.
  – Is not in the middle of a write.
• Work out a new concurrent algorithm that can work with a thread that may be in the middle of a write.
  – Maintain scalability and keep a reasonable overhead.
GC Progress 3: Controlling Fragmentation

• If we do not compact, is fragmentation really an issue?
• Bounded by $\frac{1}{2} \cdot \log(n)$, where $n$ is the size of the largest allocatable object [Robson 1974].
  – Arraylets (and object-lets) can nicely reduce $n$, but have some cost in complexity and running time [Bacon et al. 2003, Sartor et al. 2010, Pizlo et al. 2010]
• Trade-off between partial compaction efforts and space overhead [Bendersky & P 2011].

Personal opinion:
4x worst-case and 2x average-case is fine.
Controlling Space Overhead

• Option 1: design a lock-free compactor that makes progress.

• Option 2:
  – reduce allocatable object size to 64 (use arraylets and more),
  – use something similar to Robson’s allocator and make it fit your collector.

Algorithmic challenge

Engineering challenge

A nice engineering solution in the Schism collector [PLDI’10]; not trivial to port to another JVM. Can we solve the algorithmic challenge?
Conclusion

• An efficient lock-free GC would be of great use for simplifying the design of lock-free data structures.
• We don’t have it.
• We’ve made some progress, there are algorithmic and system challenges.

Can we/you do it?