Making Lockless Synchronization Fast: Performance Implications of Memory Reclamation

Tom Hart, University of Toronto
Paul E. McKenney, IBM Beaverton
Angela Demke Brown, University of Toronto
Making Lockless Synchronization Fast: Performance Implications of Memory Reclamation

Outline

• Motivation
• Memory Reclamation Schemes
• Results
• Conclusions
My Laptop Has Two Cores

- Multiprocessing becoming mainstream.
- Synchronization must be fast.
- Locks create problems:
  - Overhead
  - Serialization Bottleneck
  - Deadlock
  - Priority Inversion
Lockless Synchronization

• Using a shared object *without* locking.
  – Non-blocking synchronization.
  – Read-copy update.

✔ Can drastically improve performance. 😊

✗ Can lead to *read-reclaim races*. 😞
Read-Reclaim Races

Thread 1

A

Thread 2
Read-Reclaim Races

Thread 1: `remove(A);`

Thread 1

Thread 2

Making Lockless Synchronization Fast: Performance Implications of Memory Reclamation
Read-Reclaim Races

Thread 1:

Thread 2:

Thread 1: `remove(A);`

Thread 2: `read(A->next);`
Read-Reclaim Races

Thread 1: \texttt{remove(A);}  

Thread 2: \texttt{read(A->next);}  

When can Thread 1 call \texttt{free(A)} without interfering with Thread 2?
Contribution

• Much prior work solves read-reclaim races, but...
  – How do these solutions perform?
  – What *factors* determine performance?
  – Is the performance impact significant?
• Investigate with a microbenchmark:
  – Vary factors independently.
Outline

• Motivation
• **Memory Reclamation Schemes**
• Results
• Conclusions
Memory Reclamation Schemes

• Mediate read-reclaim races.

• Many have been proposed:
  – Quiescent-State-Based Reclamation [M&S]
    • Used with Read-Copy Update (RCU)
  – Epoch-Based Reclamation [Fraser]
  – Hazard Pointers [Michael]
  – Lock-Free Reference Counting [Valois, D. et. al.]
Memory Reclamation Schemes

- Mediate read-reclaim races.
- Many have been proposed:
  - **Quiescent-State-Based Reclamation** [M&S]
    - Used with Read-Copy Update (RCU)
  - Epoch-Based Reclamation [Fraser]
  - **Hazard Pointers** [Michael]
  - Lock-Free Reference Counting [Valois, D. et. al.]
Assumption!

• For the purposes of this presentation:
  – A thread accesses the elements of a shared data structure only through a well-defined set of operations.

• Operations:
  – find()
  – insert()
  – enqueue()
  – dequeue()
  – etc.
Quiescent-State-Based Reclamation

for (i=0;i<100;i++)
    if(list_find(L, i))
        break;

/* Do other work.... */

Thread has no references to any element in list L.
Quiescent-State-Based Reclamation

for (i=0; i<100; i++)
    if (list_find(L, i))
        break;

quiescent_state();

/* Do other work.... */

Introduce application-dependent quiescent states.

Thread has no references to any element in list L.
Quiescent-State-Based Reclamation

- *Grace period*: any interval in which each thread passes through a quiescent state.
Quiescent-State-Based Reclamation

- *Grace period*: any interval in which each thread passes through a quiescent state.

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>QS</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>QS</td>
</tr>
</tbody>
</table>

Execution Time
Quiescent-State-Based Reclamation

- **Grace period**: any interval in which each thread passes through a quiescent state.

---

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>QS</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>QS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>QS</td>
</tr>
</tbody>
</table>

Execution Time

Making Lockless Synchronization Fast: Performance Implications of Memory Reclamation
Quiescent-State-Based Reclamation

- **Grace period**: any interval in which each thread passes through a quiescent state.

```
Thread 1

Thread 2       QS  QS  QS

Thread 3       QS

Execution Time
```

Making Lockless Synchronization Fast: Performance Implications of Memory Reclamation
Quiescent-State-Based Reclamation

- *Grace period*: *any* interval in which each thread passes through a quiescent state.

![Diagram showing execution time for three threads](image)
Quiescent-State-Based Reclamation

Any element removed before this point.....

... can be safely reclaimed after this point.

Grace Period

Thread 1

Thread 2

Thread 3

Execution Time
Epoch-Based Reclamtion

- Similar to quiescent-state-based scheme.
- Instead of `quiescent_state()`, uses:
  - `lockless_begin()`
  - `lockless_end()`
- *Within* the body of an operation.
  - Application-independent.
Hazard Pointers

Thread 1: remove(A);

Thread 2: read(A->next);
Hazard Pointers

Global Hazard Pointer Array:

HP[0]
HP[1]
HP[2]
HP[3]

Thread 1:
remove(A);
free_later(A);
...
if (!find(HP, A))
free(A);

Thread 2:
HP[2] = A;
if (A has been removed)
goto RETRY;
read(A->next);

Making Lockless Synchronization Fast: Performance Implications of Memory Reclamation
Outline

- Motivation
- Memory Reclamation Schemes
- Results
- Conclusions
Performance Factors

• In our paper, we consider:
  – Number of CPUs
  – Number of threads
  – Choice of data structure
  – Workload (read-to-update ratio)
  – Length of chains of elements
  – Memory constraints

• Look at a few in this presentation.
Performance Factors

• In our paper, we consider:
  - Number of CPUs
  - **Number of threads**
  - Choice of data structure
  - Workload (read-to-update ratio)
  - **Length of chains of elements**
  - Memory constraints

• Look at a few in this presentation.
Sequential Consistency

- Parallel schedule of instructions is equivalent to a legal serial schedule; ie.
  - Machine instructions are not reordered.
  - Memory references are globally ordered.
Sequential Consistency

- Hardware is *not* sequentially-consistent.
  - CPUs can reorder instructions for performance.
- Must force sequential consistency.
  - Use a *memory fence*.
- Fences affect relative performance:
  - Fences are expensive (orders of magnitude).
  - Reclamation schemes need **different numbers of fences**!

Don't Bet On It!!!
Data Structures and List Length

- Can affect the number of fences needed.
  - Linked lists have long chains of elements.
  - Well-designed hash tables have short chains.
Example – Hazard Pointers

```c
for (cur = list->head; cur != NULL; cur = cur->next) {
    *hazard_ptr = list->cur;
    memory_fence();
    /* continue...*/
}
```
Example – Hazard Pointers

```c
for (cur = list->head; cur != NULL; cur = cur->next) {
    *hazard_ptr = list->cur;
    memory_fence();
    /* continue...*/
}
```

Making Lockless Synchronization Fast: Performance Implications of Memory Reclamation
Example – Hazard Pointers

```c
for (cur = list->head; cur != NULL; cur = cur->next) {
    *hazard_ptr = list->cur;
    memory_fence();
    /* continue...*/
}
```
Example – Hazard Pointers

```c
for (cur = list->head; cur != NULL; cur = cur->next) {
    *hazard_ptr = list->cur;
    memory_fence();
    /* continue...*/
}
```

\[ \text{O(n) fences needed!! 😞} \]
Example – Epochs

lockless_begin();  // calls memory_fence() */
for (cur = list->head; cur != NULL; cur = cur->next) {
    /* continue...*/
}
lockless_end();    // calls memory_fence() */

Making Lockless Synchronization Fast: Performance Implications of Memory Reclamation
Making Lockless Synchronization Fast: Performance Implications of Memory Reclamation

Example – Epochs

```c
lockless_begin();  /* calls memory_fence() */
for (cur = list->head; cur != NULL; cur = cur->next) {
    /* continue...*/
}
lockless_end();    /* calls memory_fence() */
```
Example – Epochs

```
lockless_begin();  /* calls memory_fence() */
for (cur = list->head; cur != NULL; cur = cur->next) {
    /* continue...*/
}
lockless_end();  /* calls memory_fence() */
```
Example – Epochs

O(1) fences needed! 😊

lockless_begin(); /* calls memory_fence() */
for (cur = list->head; cur != NULL; cur = cur->next) {
    /* continue...*/
}
lockless_end(); /* calls memory_fence() */
Example – Quiescent States

One fence per several operations. 😊😊😊

```
for (cur = list->head; cur != NULL; cur = cur->next) {
    /* continue...*/
}
```
Traversal Length

O(n) fences add up!!!

Making Lockless Synchronization Fast: Performance Implications of Memory Reclamation
Traversals Length - LFRC

O(n) atomic increments are much worse!!!
Traversal Length - LFRC

![Graph showing performance implications of memory reclamation with quiescent states, hazard, epochs, and reference counts.]

**So, we should always use quiescent states or epochs, right?**

Making Lockless Synchronization Fast: Performance Implications of Memory Reclamation
Threads and Memory

• Hazard pointers bound unfreed memory.
  – (Provided number of hazard pointers is finite.)
  – Everything with no hazard pointer can be freed.
  – Other schemes are more memory-hungry.

• What happens when there are many threads?
  – More threads than CPUs ⇒ Preemption.
Grace periods are infrequent, so performance suffers!
Preemption With Yielding

If we yield on memory allocation failure, grace periods are more frequent!
Outline

• Motivation
• Memory Reclamation Schemes
• Results
• Conclusions
Summary of Results

• Schemes have very different overheads.
  – Difference between “faster than locking” and “slower than locking.”

• No scheme is always best.

• Quiescent-state-based reclamation has the lowest best-case overhead.
  – No per-operation fences. 😊

• Hazard pointers are good when there is preemption and many updates.
Significance

• Understanding performance factors lets us:
  ✔ Choose the right scheme for a program.
  ✔ Design new, faster schemes.
Future Work
Future Work

- Macrobenchmark
- Use lockless synchronization with SPLASH-2
- Quiescent-State-Based Reclamation with realtime workloads
Questions?

- Tom Hart
  - http://www.cs.toronto.edu/~tomhart
- Angela Demke Brown
  - http://www.cs.toronto.edu/~demke
- Paul McKenney