Memory Management for Concurrent Data Structures on Hardware Transactional Memory

Peter Pirkelbauer
Amalee Wilson
Hadia Ahmed
Reed Milewicz

Computer and Information Sciences
University of Alabama at Birmingham
Non-blocking Stack (pop)

```cpp
pair<int, bool> pop(Stack::Top& top) {
    Stack* nexttop = nullptr;
    Stack* currtop = top.load();
    do {
        if (currtop == nullptr) return make_pair(0, false);
        nexttop = currtop->next;
    } while (!top.compare_exchange_strong(currtop, nexttop));
    int res = currtop->val;
    delete currtop; // OK?
    return make_pair(res, true);
}
```

![Diagram of stack and thread](image)
Non-blocking Stack (pop)

```cpp
code
pair<int, bool> pop(Stack::Top& top) {
    Stack* nexttop = nullptr;
    Stack* currtop = top.load();

    do {
        if (currtop == nullptr) return make_pair(0, false);

        nexttop = currtop->next;
    } while (!top.compare_exchange_strong(currtop, nexttop));

    int res = currtop->val;
    delete currtop; // OK?
    return make_pair(res, true);
}
```

Diagram:
```
Stack

Thread 1
```

`pirkelbauer@uab.edu
Transact'17
February 7, 2017 2 / 36`
Non-blocking Stack (pop)

```cpp
class Stack {
    // Stack implementation details
};
	pair<int, bool> pop(Stack::Top& top) {
        Stack* nexttop = nullptr;
        Stack* currtop = top.load();

        do {
            if (currtop == nullptr) return make_pair(0, false);
            nexttop = currtop->next;
        } while (!top.compare_exchange_strong(currtop, nexttop));

        int res = currtop->val;
        delete currtop; // OK?
        return make_pair(res, true);
    }
```
Solutions

- Garbage collection
- Epochs, Harris (2001)
- Repeated Offender, Herlihy et al. (2004)
- Reference Counting, Gidenstam et al. (2007)
- QSense, Balmau et al. (2016)
Epochs

Idea

- Record when a thread starts and finishes an operation.
- Record when an object gets removed.
- Delay deletion until threads are past the deletion time or inactive.
Epochs

Thread 1

Stack

Thread 2

ctr 5

ctr 3

Top

7

12
Epochs

Thread 1

Stack

Thread 2

Cannot be freed b/c Thread 1 started unfinished op 5.
Epochs

Thread 1

ctr

5

Stack

Top

12

Thread 2

ctr

4

7

removed at

5

3

=> delay freeing the element...
Epochs

Thread 1

Stack

Thread 2

After Thread 1 is done it can no longer get a reference to the removed element. The element can be freed.
Epochs

Benefits
- Low overhead (coarse grain)
- Can bulk free at most every $n$ times, $n > |threads|$

Problems
- What if a thread fails in the middle of an operation?
Hardware Transaction
In principle
HTM makes memory management easy.
In principle

HTM makes memory management easy.
In principle

HTM makes memory management easy.
In practice

- Hardware limits transaction size
- It may be undesirable to keep entire data structure in transaction
- Access data structures in partitions.
- Nodes at partition boundaries need to be safe.
- Cost of memory management is reduced greatly.
Benefits of longer transactions

- Memory management is cheap
- Relaxed memory operations (Threads sync at begin and end of transaction)
- Amortize cost of transactional overhead over more operations

Drawbacks of longer transactions

- Increases chance of transactional aborts
  - conflict aborts
  - capacity aborts
  - OS aborts
Solution

Compute TX size over rolling average of \( n \) successful operations. Dragojević et al., 2011.
Epochs / HTM
Epochs for HTM

Problem
A delayed or failed thread prevents memory from being reclaimed.

Solution
Abort operation executed by delayed thread. (Debra+, Brown (2015)).

HTM
- HTM offers hardware support for aborting operation.
- increment other thread’s epoch counter.
Transaction and Epochs

- Start epoch
- Validate epoch in transaction
- Abort and restart operation if counter is even

```c
start_epoch();
// transaction partition
if (tx_begin()) {
    ...
    if (!validate_epoch())
        tx_abort();
    ...
    tx_end();
} end_epoch();
```
Start epoch

Validate epoch in transaction
  Abort and restart operation if counter is even

Use of Epochs

```c
start_epoch();
// transaction partition
if (tx_begin()) {
...
  if (!validate_epoch())
    tx_abort();
...
  tx_end();
} end_epoch();
```
- Start epoch
- Validate epoch in transaction
  - Abort and restart operation if counter is even

```
start_epoch();
// transaction partition
if (tx_begin()) {
    ...  
    if (!validate_epoch())
        tx_abort();
    ...  
    tx_end();
}
end_epoch();
```
Epochs and Reclamation

Removed Object
- Address to be freed
- Epoch vector when object was removed
- Survival count

Thread Aborts
Abort other threads that prevent object reclamation for \( n \) times.
- A transaction is tried $n$ times before partition size is reduced
- Lock-based fall back path
- HTM and C++ atomics

```c
void cancel_transaction(size_t threadid, size_t currepoch) {
    epoch_t* epochptr = lookup_thread(threadid);
    epochptr->epoch.compare_exchange_strong(currepoch, currepoch+1);
}
```
Epochs Summary

- Small memory and time overhead
  - Operation on a data structure requires transaction
Evaluation
Hazard Pointers for HTM

- Publish needed nodes at end of partition
- Scan other threads for nodes that are referenced
  - Storing hazard pointers reduces effective transaction size

Dragojević et al., 2011.
- Publish beginning and end of pointer array on stack
  + No overhead for storing hazard pointers
- Collecting other threads pointers requires TX
  Alistarh et al. 2014.
Each object has a reference counter
- Increment counter / decrement counter if object is no longer needed
  + No need to scan other thread’s pointers
  - frequent conflicts on counter accesses
  - counters reduce effective transaction size significantly

Dragojević et al., 2011.
<table>
<thead>
<tr>
<th>Systems</th>
<th>Intel Haswell</th>
<th>IBM Power 8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 socket</td>
<td>2 socket</td>
</tr>
<tr>
<td></td>
<td>4 cores</td>
<td>20 cores</td>
</tr>
<tr>
<td></td>
<td>8 threads</td>
<td>160 threads</td>
</tr>
<tr>
<td></td>
<td>Cache line: 64 bytes</td>
<td>Cache line: 128 bytes</td>
</tr>
<tr>
<td></td>
<td>Level 1: 32K, 8-way</td>
<td>Level 1: 64K, 8-way</td>
</tr>
<tr>
<td></td>
<td>Level 2: 256K, 8-way</td>
<td>Level 2: 512K, 8-way</td>
</tr>
<tr>
<td></td>
<td>Max TX (load): 4M</td>
<td>Max TX (load): 8K</td>
</tr>
<tr>
<td></td>
<td>Max TX (store): 32K</td>
<td>Max TX (store): 8K</td>
</tr>
<tr>
<td></td>
<td>no progress guarantee</td>
<td>no progress guarantee</td>
</tr>
</tbody>
</table>
Evaluation

Linked List

- 100K operations (total)
- 10% untimed initial inserts
- alternating insert and erase
- threads access disjoint regions
Evaluation - IBM Power 8

The image shows a bar chart with the x-axis representing the number of threads ranging from 1 to 160, and the y-axis representing time in milliseconds ranging from 200 to 20000. The chart compares different synchronization methods (HTM / No Manager, HTM / Epochs, HTM / HP, HTM / Ref Counting, HTM / StackTrack, Locking / Epochs, Lockfree / HP) across various thread counts.

The legend indicates the different synchronization methods, and the bars are color-coded accordingly. The chart notes that lower times are better, suggesting that the goal is to minimize the execution time across different thread counts for each synchronization method.
Skip list

- 4M operations (total)
- 10% untimed initial inserts
- alternating insert and erase
- threads access disjoint regions
- Intel: 32 levels, elements in next layer $\frac{1}{2}$
- IBM: 16 levels, elements in next layer $\frac{1}{8}$
How does the memory management technique impact TX size?
Evaluation - Intel Haswell / Linked list

![Graph showing performance comparison of different HTM methods across threads](image)

- HTM / No Manager
- HTM / Epochs
- HTM / Publish and Scan
- HTM / Ref Counting
- HTM / StackTrack

Steps/Transaction vs Threads
Summary

- Introduced Epochs / HTM
- HTM is a powerful mechanism to speed up common case
- Most memory management techniques perform similarly

Future Work: Evaluation with other data structures and access patterns
Thank you!