Lock-Free Programming

Andrei Alexandrescu
andrei@metalanguage.com

"Life is just one darn thing after another"

Elbert Hubbard

“Multithreading is just one darn thing after, before, or simultaneously with another”

Agenda

- Architectural trends and how they affect programming styles
- Lock-based vs. lock-free
- CAS-based code
- Retiring old data in lock-free programming
- Conclusions

Architectural Trends

- Yesterday: scarce processing power, wiring relatively unimportant
- Today: lots of processing power, albeit hard to access
- Tomorrow: tons of *inaccessible* processing power
- Transistor count on the rise
- Connectivity becomes nightmarish
- Light won't travel any more faster
  - At 10 GHz: 3 cm/cycle!

Needs

- Need more parallelism
  - ILP: ~2.5 instructions/cycle
- Need to put more on the chip
  - 2005: all ?P vendors will release logical MPs
- Need better on-chip data locality/retention
  - Memory latency and bandwidth issues
- Power issues
  - Speculative loading and execution = wasted power (and bandwidth munching)

Multithreading

- MT is one of precious few software techniques to increase processor utilization
- Serial code is hard to parallelize
  - Parallel code is easy to parallelize
- Not all threads stall at the same time
- Do more work with less power

Lock-based vs. Lock-free

- Lock-based:
  - Access to shared data protected by mutex locking/unlocking
  - Inside a locked region, arbitrary operations can be perfrmed
- Lock-free:
  - No need for locking (duh)
  - Precious few ops allowed on shared data

Impossibility/universality

- 1991: Herlihy paper “Wait-free synchronization”
- Some primitives cannot synchronize any shared data structure for >2 threads
  - e.g., atomic queues!
- Some other primitives are enough to implement any shared data structure
  - e.g., CAS

CAS

- Do this atomically:

```
template <class T>
bool CAS(T* addr, T expected, T fresh) {
    if (*addr != expected) return false;
    *addr = fresh;
    return true;
}
```

- Usually `T = {int32, int64, ...}`
- Implemented by all major processors
  - This year: transactional memory

Defining terms

- **Wait-free procedure:** completes in a bounded number of steps regardless of the relative speeds of other threads
- **Lock-free procedure:** at any time, at least one thread is guaranteed to make progress
  - Probabilistically, all threads will finish timely
- Mutex-based procedures:
  - Not wait-free
  - Not lock-free

Advantages of lock-free

- Fast (~ 4 times faster than best locks)
- Deadlock immunity
- Livelock immunity
- Thread-killing immunity
  - Killing a thread won't affect others
- Asynchronous signal immunity
  - Reentrancy is automatic
- Priority inversion immunity
  - Easier design

Disadvantages

- Priorities uncontrollable
  - Can increase contention gratuitously
- Hard to program
  - Herlihy's proofs assumed infinite memory
  - GC is a big helper
  - Hard even with GC
- Use locks for 98% of the code
- Use CAS for 2% of the code to increase performance by 98%

Basic CAS-based idioms

- In your class, keep pointers to the shared data (don’t embed it)
- When updating shared data:
  - Do all the work on the side in another pointer
  - CAS-in the new pointer
  - Do that in a loop to make sure you update the right data

- If garbage collection, then done!

Example

```
class Widget {
  Data * p_;
  ...
  void Use() { ... use p_ ... }
  void Update() {
    Data * pOld, * pNew = new Data;
    do {
      pOld = p_;
      ...
    } while (!CAS(&p_, pOld, pNew));
  }
};
```

Retiring Old Data

- Problem: when to `delete` the old `p_`?
  - Reference counting?
    - Can’t do, need DCAS
  - Wait some, then delete?
    - Fragile approach: how long is enough?
    - (How large is a large enough buffer?)
  - Keep the reference count next to `p_`
    - Requires CAS2
    - Writes are locked by reads

Hazard Pointers

- Idea: maintain a global singly-linked list of "pointers in use" – the hazard pointers (**`hlist`**)
  - The list is easy to manipulate with CAS only
- Whenever a thread replaces a pointer, it puts the old one in a thread-local, private list (**`rlist`**)
- When **`rlist`** has grown up to a fixed size:
  - Do the set difference rlist – hlist
  - **`delete`** all pointers in the result set!

Example

```
void Widget::Use() {
  hlist->add(p_);
  ... use p_ ...
  hlist->remove(p_);
}
void Widget::Update() {
  ... replace p_ with pNew ...
  rlist->add(pOld);
  if (rlist->size() > R) {
    set<Data*> d = difference(rlist, hlist);
    ... delete all in d ...
  }
};
```

Optimizations

- A set difference can be computed in O(R) if one of the lists is sorted (at cost O(R log R))
- Hashing would be an alternative
  - O(R) expected complexity
- In any case, the algorithm is wait-free
  - Scans the wait-free `hlist` and the thread-local private `rlist`

Choosing parameters

- So, complexity of the scanning algo is **`O(R)`**
- Maximum number of retired pointers that haven't been deleted is **`N * R`**
- **`N`** is the number of writers
- A good choice:
  - **`R = (1 + k) * H`**
  - **`H`** is the max number of readers
  - **`k > 1`** small positive number
  - Each scan deletes **`R – H = O(R)`** pointers
- So the amortized time to **`delete`** any unused pointer is constant

Conclusions

- Efficient programs are hip again
- Threads are hipper than ever
- Lock-free offer high-efficiency for simple structures
- Lock-based programming is easier for complex structures
- CAS-based code is cool

Bibliography

- Maged Michael: *Scalable lock-free dynamic memory allocation*, PLDI 2004
- Maged Michael: Hazard Pointers: *Safe Memory Reclamation for Lock-Free Objects*, IEEE TPDS 1994
- Maurice Herlihy: *Wait-Free Synchronization*, ACM Transactions on Programming Languages and Systems, 13(1):124--149, January 1991
- Andrei Alexandrescu: *Generic<Programming>* in CUJ, Oct and Dec 2004 (2nd with M. Michael)