Techniques for Implementing Concurrent Data Structures on Modern Multicore Machines

Stephen Tu

Why do we care?

- Single-processor speed has more or less flat-lined.
  - Multi-core machines ubiquitous.
- Unfortunately, throwing more hardware does not always translate into speedup.
  - A lot of code is written sequentially, or with parallelism as an afterthought.
- Need to understand how to effectively utilize the hardware.

What we are not covering

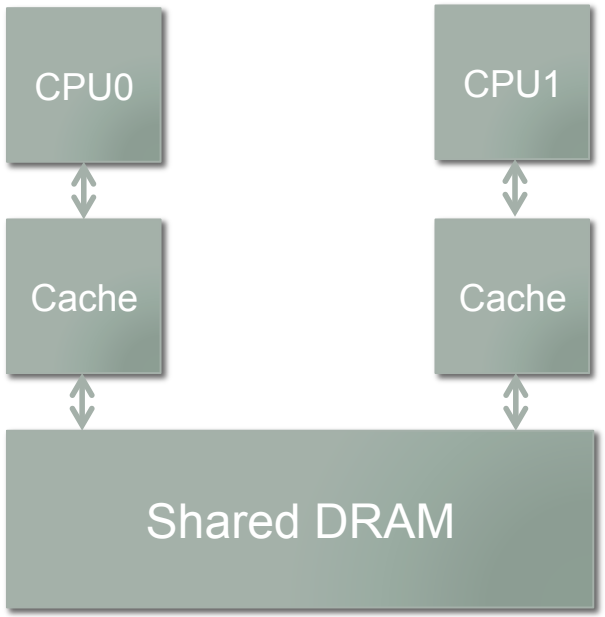
- We'll focus on techniques for shared memory multi-processor machines
  - N processors, each can read/write to the same shared memory.
- This is a different set of techniques than those for programming multiple machines (which *usually* do not have shared memory).
  - This is also a very interesting area in computer science with a different set of hard problems, i.e. independent failures, lost messages, network partitions, adversarial nodes.

Alright, let’s write some code…

Not so fast!

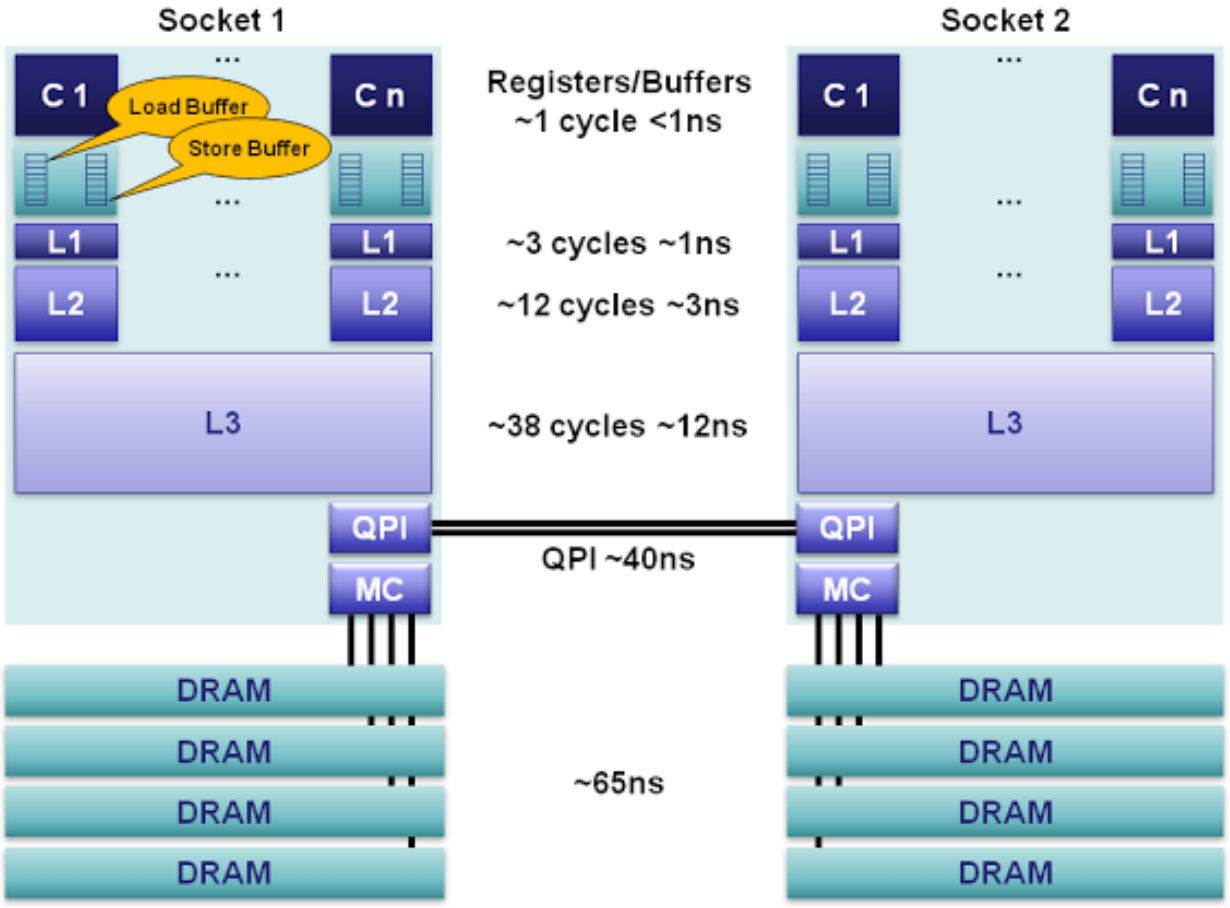
Cache-Coherence Primer

- An oversimplification of modern shared memory machines



- For performance, almost all modern CPU caches are *write-back*: the value in the cache is the most up-to-date version.

Real example: Intel Sandy Bridge



http://mechanical-sympathy.blogspot.co.uk/2013/02/cpu-cache-flushing-fallacy.html

Cache-Coherence Protocol

- If caches are write-back, then how do we make a write to memory by CPU0 visible to CPU1?
  - One answer is to punt- this isn't necessarily bad since many variables are not intended to be shared between multiple CPUs, like each CPU's stack.
- Straw-man solution: have CPU0 send CPU1 the contents of every memory write, so CPU1 can keep its local cache synchronized.
  - Not desirable because CPU1 probably does not care about most of CPU0's writes.
  - Would cause a lot of unnecessary bus traffic on the interconnect.

Cache-Coherence Protocol

- Commonly deployed solution: multiple readers, single writer per cache-line (MSI protocol)
  - Recall that a cache-line is the unit of transfer between memory and cache, sort of like a disk block between disk and memory.
  - Each cache-line can be in one of three states:
    - **Modified**: the contents no longer equals that of main memory
    - **Shared**: the contents equals that of main memory
    - **Invalid**: does not contain valid contents
  - Only a single cached copy of a cache-line can be in the **Modified** state (single writer). Multiple cached copies of a cache-line can be in the **Shared** state (multiple readers).

Cache-Coherence Protocol

- Read path (I-State):
  - Request the M-state cache-line, if any, downgrade to S-state, and write back dirty cache-line to memory.
  - Obtain copy of cache-line and transition to S-state.
- Read path (S-State):
  - No-op.
- Read path (M-State):
  - No-op.

Cache-Coherence Protocol

- Write path (I-State):
  - Either: (A) request the M-State cache-line downgrade to I-State and write-back, or (B) request all S-State cache-lines to downgrade to I-State.
  - Obtain copy of cache-line and transition to M-State.
- Write path (S-State):
  - Request all other S-State cache-lines downgrade to I-State.
  - Transition to M-State.
- Write path (M-State):
  - No-op.

Why care about cache-coherence?

- Cache-coherence seems like some hardware detail that programmers do not need to care about…
  - Mostly true, unless we want to write scalable programs on multi-core
- **Take-away**: Multiple CPUs updating a shared cache-line is inherently a scalability bottleneck!
  - Now you know why: the CC protocol must continuously issue invalidation requests to move the cache-line between multiple CPUs, causing a “ping-pong” effect and high bus traffic.
- **The secret sauce of implementing scalable data structures is avoiding this kind of contention.**
  - Easier said than done, of course.

An aside: out-of-order execution

- Another aspect which makes concurrent programming tricky is out-of-order instruction execution.
  - For performance reasons, no modern processor executes instructions in program order!
    - The only guarantee given is that a single thread cannot observe out-of-order execution. **But other threads can.**
  - This results in surprising behavior. For example, this is observable on x86 [Sewell 10]:

Proc 0	Proc 1
MOV [x]←1 MOV EAX←[y]	MOV [y]←1 MOV EBX←[x]
Allowed Final State: Proc 0:EAX=0 ∧ Proc 1:EBX=0	

- Could spend an entire workshop alone talking about OoO execution and relaxed memory models!

[Sewell 10] Sewell et al. *x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors*. CACM 2010.

Correctness Conditions

- We need a way to argue about correctness for concurrent data structures.
- We are used to reasoning about correctness in sequential data structures. But what happens when we have concurrent operations?
- Suppose the following code was executed concurrently:

T1:
q.push_back(1)
q.push_back(2)

T2:
q.push_back(3)
q.push_back(4)

- Most would probably agree that the only valid outcomes are enumerated as:
  - q = [1, 2, 3, 4], q = [1, 3, 4, 2], q = [1, 3, 2, 4],
    q = [3, 1, 2, 4], q = [3, 1, 4, 2], q = [3, 4, 1, 2]

Correctness Conditions

- Why would we think that q = [2, 1, 3, 4] is not correct?
  - T1 executes q.push_back(1) before q.push_back(2), so we know this violates a “happens-before” relation.
- Why do we allow both q = [1, 2, 3, 4] and q = [1, 3, 2, 4]?
  - Arguably, T1 executes q.push_back(2) at “roughly” the same time as T2 executes q.push_back(3), so both are “OK”.
- ***Linearizability*** [Herlihy 90] is a way to formalize this intuition.

[Herlihy 90] – Herlihy and Wing. *Linearizability: a correctness condition for concurrent objects.* ACM TOPLAS 1990.

Linearizability: Definition

- Let q be an object. We break up a method call on q into a request event (*when the method starts executing*) and a response event (*when the method returns with a result*).
- Let an *execution history* H be defined as a sequence of method requests and responses on q by various threads.
- An execution history H is called *linearizable* if the following conditions hold:
  - There exists some permutation of H, call it H', such that each method request is immediately followed by its corresponding response (H' is sequential).
  - If a method response precedes a method request in H, then it also does in H' (all happens-before relations are preserved).
  - H' is a legal history. That is, if we were to execute H' sequentially, then we would get the same history back.

Linearizability in practice

- Informally, it is sufficient to think of linearizability as such:
  - Every method call appears to take place “instantaneously” at some point between its request and response. This point is called the *linearization point*.
  - In other words, no other thread can *partially observe* the effects of a method call. Furthermore, we have some real time guarantee.
  - For example, the linearization point of a critical section protected by a lock is when the lock is released.
- Once again, why do we care?
  - This might seem like an academic exercise, but identifying linearization points is a very useful way to reason about correctness.
  - Suppose you have n methods in a concurrent data structure. Without identifying linearization points, then you have to reason about all possible O(n^2) concurrent interactions.
    - With linearization points, you only need to do O(n) work to identify those points.

Enough of this background, let’s see some code

Concurrent Singly Linked List

- Why are we studying linked lists? Shouldn't we have mastered this already?
  - While you might be able to whip out a sequential implementation in your sleep, turns out a concurrent implementation requires some thought.
  - Great way to demonstrate the techniques (since we are all intimately familiar with the data structure).
- Code snippets online: https://github.com/stephentu/scalex
  - Snippets written in the new C++11 standard, for x86_64. I recommend using g++ >= 4.7 for compilation.
  - Sorry, no Mac OSX support.

Solution One: Single global lock

- The most obvious (perhaps too obvious) place to start is by acquiring a global lock before each method call.
- Has some really desirable properties:
  - Trivially linearizable – correctness is easy.
  - Easy to read/maintain
- For infrequently used data-structures, this is a completely reasonable approach.
- Okay, but let's do better…

Solution Two: Per-node locks

- Instead of having a single lock over the entire list, let's place a lock in each list node.
  - An example of *fine-grained* locking.

```
typedef spinlock /* spinlock.hpp */ lock_type;
typedef std::unique_lock<lock_type> unique_lock;
typedef std::shared_ptr<node>       node_ptr;

struct node {
    node() : value_(), next_() {}
    node(const T &value, const node_ptr &next)
      : value_(value), next_(next) {}

    // Note: mutex_ must be held in order to access next_
    mutable lock_type mutex_;
    T value_;
    node_ptr next_;
};
```

Reference counting

- Since we are working with C++, we cannot ignore memory management. We use reference counting (std::shared_ptr) to make the code cleaner.
  - Could be explicit about memory management.
- std::shared_ptr is not thread-safe. Multiple threads cannot modify the *same* std::shared_ptr instance concurrently without explicit synchronization.
  - We avoid this problem because by holding a mutex any time we access a shared_ptr instance.
- More on reference counting later…

Per-node locks: Traversal

- **Invariant**: Holding on a node's lock prevents it from being mutated.
- So list traversal must be pretty simple, right? Simply lock a node, read its next pointer, release the current node's lock, and move on.
- Consider the following race condition:

T1:

```
node0->mutex_->lock();
node1 = node0.next_;
node0->mutex_->unlock();

node1->mutex_->lock(); // oops
```

T2:

```
// remove node1 from list
```

Solution: hand-over-hand locking

- We need a way to ensure that the action of reading a next pointer and locking the next node is atomic. Otherwise, we could end up reading removed nodes.
- Hand over hand locking to the rescue: Lock the next node *before* releasing the lock on the current node.
  - remove() also needs to follow this protocol. Coming soon.

Code snippet

- Here is how size() is implemented, with HOH locking:

```
size_t
size() const
{
    size_t ret = 0;
    node_ptr prev = head_;
    head_->mutex_.lock();
    node_ptr cur = head_->next_;
    while (cur) {
        cur->mutex_.lock();    // acquire next lock first
        prev->mutex_.unlock(); // then release cur lock
        ret++;
        prev = cur;
        cur = cur->next_;
    }
    prev->mutex_.unlock();
    return ret;
}
```

Per-node locks: Removals

- In order for the HOH locking to work, we need remove() to obey the protocol.
- **Invariant**: If either lock on a node or a node's predecessor is held, then a node cannot be unlinked.
- Given that we specified that a node's next value can only be accessed with a node's mutex held, it should be clear that we need to hold both locks on removal:
  - `prev->next_ = cur->next_; // unlink cur`

Code snippet

- Here's how remove() is implemented:

```
void
remove(const T &val)
{
    node_ptr prev = head_;
    prev->mutex_.lock();
    node_ptr cur = prev->next_;
    while (cur) {
        cur->mutex_.lock();
        if (cur->value_ == val) {
            prev->next_ = cur->next_; // unlink cur
            cur->mutex_.unlock();
            cur = prev->next_;
        } else {
            prev->mutex_.unlock();
            prev = cur; cur = cur->next_;
        }
    }
    prev->mutex_.unlock();
}
```

What did we give up?

- Suppose we are traversing size(), and are currently in the middle of the list. Now suppose pop_front() is called, followed by push_back().
  - **This execution cannot possibly be linearizable.** Proof:
  - We did not observe the removal of the front of the list, so our linearization point must come *before* the request of pop_front().
  - We will observe the insertion of the element by push_back(), so our linearization point must come *after* the response of push_back().
  - But pop_front() *happens-before* push_back(), so we cannot construct a legal serial history.
  - remove() has similar issues.

Linearizability vs. performance

- Often in practice, only certain operations can be made to be both linearizable and scalable.
  - This is usually limited to those that mutate a single element, instead of perform scans like size() and remove().
- By using finer-grained locking, we allow for more concurrency at the cost of linearizability.
- Possible irrelevant. It is very important that we have linearizable push_back() and pop_front() (which we do), but consistent size() is probably not as important.
- Can use *optimistic* techniques, if we expect modifications to be infrequent.
  - See [Kung 81] for an overview on the idea of OCC.

[Kung 81] – Kung and Robinson. *On optimistic methods for concurrency control*. ACM TODS 1981.

Locks are so 1980s. Show me some of these lock-free data structures that all the cool kids are talking about nowadays. Get with the times!

Hardware primitives

- Modern processors usually support a variety of atomic primitives.
- Atomic primitive #1: load/store
  - You might not realize, but even `mov` is atomic on x86*.
  - Without this, you pretty much cannot do anything.
- Atomic primitive #2: compare-and-swap (CAS)
  - CMPXCHG on x86 (with a LOCK prefix).

```
template <typename T>
bool compare_and_swap(T *dst, T exp, T desired)
{   // do atomically
    if (*dst == exp) {
        *dst = desired;
        return true;
    }
    return false;
}
```

The power of CAS

- It turns out that compare-and-swap, while seemingly trivial, is actually a really powerful primitive.
  - Can be used to solve the *infinite consensus problem*- see Herily and Shavit – The Art of Multiprocessor Programming for a proof.
  - In other words, any concurrent data structure which is implementable on a Turing machine can be implemented in a wait-free manner using CAS.
    - Once again, see Herily and Shavit for a proof. This follows as a consequence of being able to solve the infinite consensus problem.

Lock-free linked list

- We'll remove all locks from our nodes.
- We'll have a deleted bit on all nodes, which is true if the node was *logically* removed (but not physically).
  - This allows us to be lazy about cleaning up nodes. Our LL can have a bunch of deleted nodes within it.
- In practice, we steal the lowest bit from the next pointer for the deleted bit.
  - Desirable, because we can set the deleted bit and next pointer in one atomic CAS.
  - Is safe, because malloc() must return an aligned address [C99 Section 7.20.3].
  - atomic_ref_ptr is our reference counting implementation (like std::shared_ptr), with the ability to mark the low bit.

Reference counting again

- Before, we noted std::shared_ptr was not thread-safe, but we were free from data races because we protected all accesses with a lock.
- Now, with no lock, we switch to atomic_ref_ptr (our own construction) which is thread-safe…
  - *But we had to use a mutex internally*. So our “lock-free” implementation becomes not lock free.
  - More on how to fix this later.
- Why don’t we just explicitly manage memory instead of doing reference counting?
  - Can’t anymore! Before, reference counting was simply a convenience. Now it’s actually a requirement for correctness...
  -

Why reference counting is necessary*

- Because we no longer have any locks, there is no way to ensure that a node is not removed while holding onto a reference to that node.
- This is also why we need the deleted bit- at any point in time, a node could be concurrently removed *while we still hold a reference to it*.
- We'll see later on how to do reference counting in a less invasive way.
  - No, the answer is not to use a general garbage collector.

Lock-free node

```
struct node;
typedef atomic_ref_ptr<node> node_ptr; // atomic_reference.hpp
struct node : public RefCountImpl {
    node() : value_(), next_() {}
    node(const T &value, const node_ptr &next)
      : value_(value), next_(next) {}

    ~node()
    {
        assert(next_.get_mark()); // sanity check
    }

    T value_;
    node_ptr next_;

    inline bool
    is_marked() const
    {
        return next_.get_mark();
    }
};
```

Lock-free traversal

- Traversal is actually pretty straight-forward. No need to do HOH locking.

```
size_t
size() const
{
    size_t ret = 0;
    node_ptr cur = head_->next_;
    while (cur) {
        if (!cur->is_marked()) {
            ret++;
        } else {
            // reap cur for garbage collection
        }
        cur = cur->next_;
    }
    return ret;
}
```

Lock-free removal

- Removal of a node proceeds in two phases.
- First, *logically* remove the node from the data structure by setting the deleted bit. At this point, any subsequent reads of the node will skip over the node.
  - For push_back()/pop_front(), marking the node to remove is the linearization point.
- Second, *physically* unlink the node from the list, by using CAS on the predecessor's next pointer.

Lock-free removal

```
void
remove(const T &val)
{
  node_ptr prev = head_;
  node_ptr p = head_->next_, *pp = &head_->next_;
  while (p) {
    if (p->value_ == val) {
      if (p->next_.mark()) { // logically remove p from list
        // try to physically unlink
        if (pp->compare_exchange_strong(p, p->next_))
          ; // successful unlink, reap p for GC
      }
      // advance the current ptr, keep prev the same
      p = p->next_;
    } else {
      prev = p;
      pp = &p->next_;
      p = p->next_;
    }
  }
}
```

Improving reference counting

- As mentioned before, thread-safe manipulation of reference counting pointers requires a lock
  - Fundamentally, there is no way to do a load (read the pointer) and store (increase reference count) *atomically* between two *different* cache lines, so we need a lock.
  - Otherwise, there's always a race between the load of the pointer and the increase of the reference count, during which the count could drop to zero.
- Furthermore, reference counting is a scalability bottleneck!
  - Increment/decrement a shared reference count between different threads is *exactly* what causes cache-line ping-pong-ing.

Epoch based garbage collection

- **Key insight 1**: we do not care about the *actual* value of the reference count, just when it drops to zero.
- **Key insight 2**: we do not have to garbage collect *immediately* when the reference count drops to zero, can delay collection.
- These two insights combine to create a technique known as epoch based garbage collection.
  - This idea is known as “read-copy-update” (RCU) in the linux community.
  - RCU is used widely within the linux kernel.
- Many different variants, we’ll talk about the simplest.

Epoch based garbage collection

- **Idea:** divide time into epochs (say 10ms).
- Run a background task which runs the following loop:
  - last_epoch = current_epoch++; // advance the current epoch by 1
  - Wait for all outstanding threads to finish last_epoch
  - Garbage collect all references freed in last_epoch
- **Why this works:** By freeing a reference (including unlinking it from any data structures) in epoch *e*, by the time epoch *e+1* comes around, we are guaranteed that no outstanding references exist anymore (because all threads finished epoch *e*).

Epoch based garbage collection

- Readers/writers must only touch RCU-protected references while within an RCU region.
- For scalex, this looks something like:

```
#include “rcu.hpp”
void do_work()
{
    scoped_rcu_region r;
    // do work with x

    // free x
    r.release(x);
}
```

Epoch based garbage collection

- The GC loop is actually really simple:

```
for (;;) {
   nanosleep(&t, NULL); // sleep an epoch
   const epoch_t cleaning_epoch = global_epoch.load();
   global_epoch.store(cleaning_epoch + 1);

   delete_queue elems;
   for (size_t i = 0; i < NSyncs; i++) { // loop over all threads
     sync &s = syncs[i].elem;
     {
       lock_guard<spinlock> l(s.local_critical_mutex);
     }
     // now the next time the thread enters a critical section, it
     // *must* get the new global_epoch, so we can now claim its
     // deleted pointers from cleaing_epoch = global_epoch - 1
     delete_queue &q = s.local_queues[cleaning_epoch % 2];
     elems.insert(elems.end(), q.begin(), q.end());
     q.clear();
   }

   // free elems
 }
```

What does this gain us?

- Recall with reference counting, every pointer access requires modifying shared cache-lines to manipulate the reference count.
- With RCU, pointer access is just a load. In the absence of mutations, all threads will hold a shared copy (S-State) in their caches, making access fast.
  - Of course, when the GC loop runs, other threads will share cache-lines with the GC thread, but we amortize the cost of this by sharing in bulk, and relatively infrequently (10ms is a lot of time in CPU cycles).

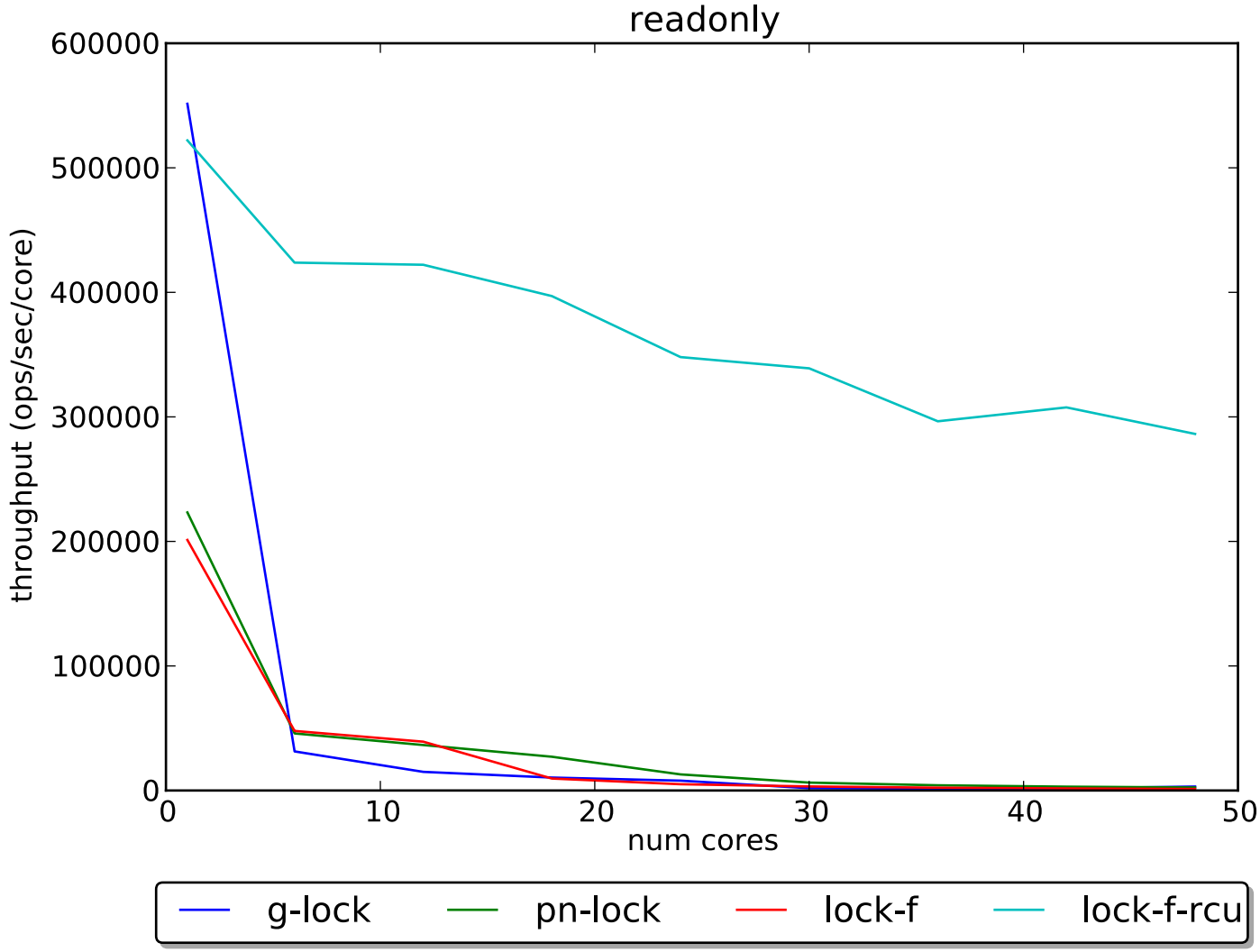
Memory allocation

- Another big scalability bottleneck: memory allocator.
- libc's built-in memory allocator essentially uses a global lock to protect its internal data structures.
  - So all this hard work we put in scaling our linked list is effectively nullified by calling malloc().
- Luckily, scalable memory allocators exist and are fairly robust. Examples include Jason Evan's jemalloc, and Google's tcmalloc.
  - Both are used extensively: jemalloc is used in FreeBSD's memory allocator, Firefox, and internally by Facebook. tcmalloc is used in WebKit and internally by Google.

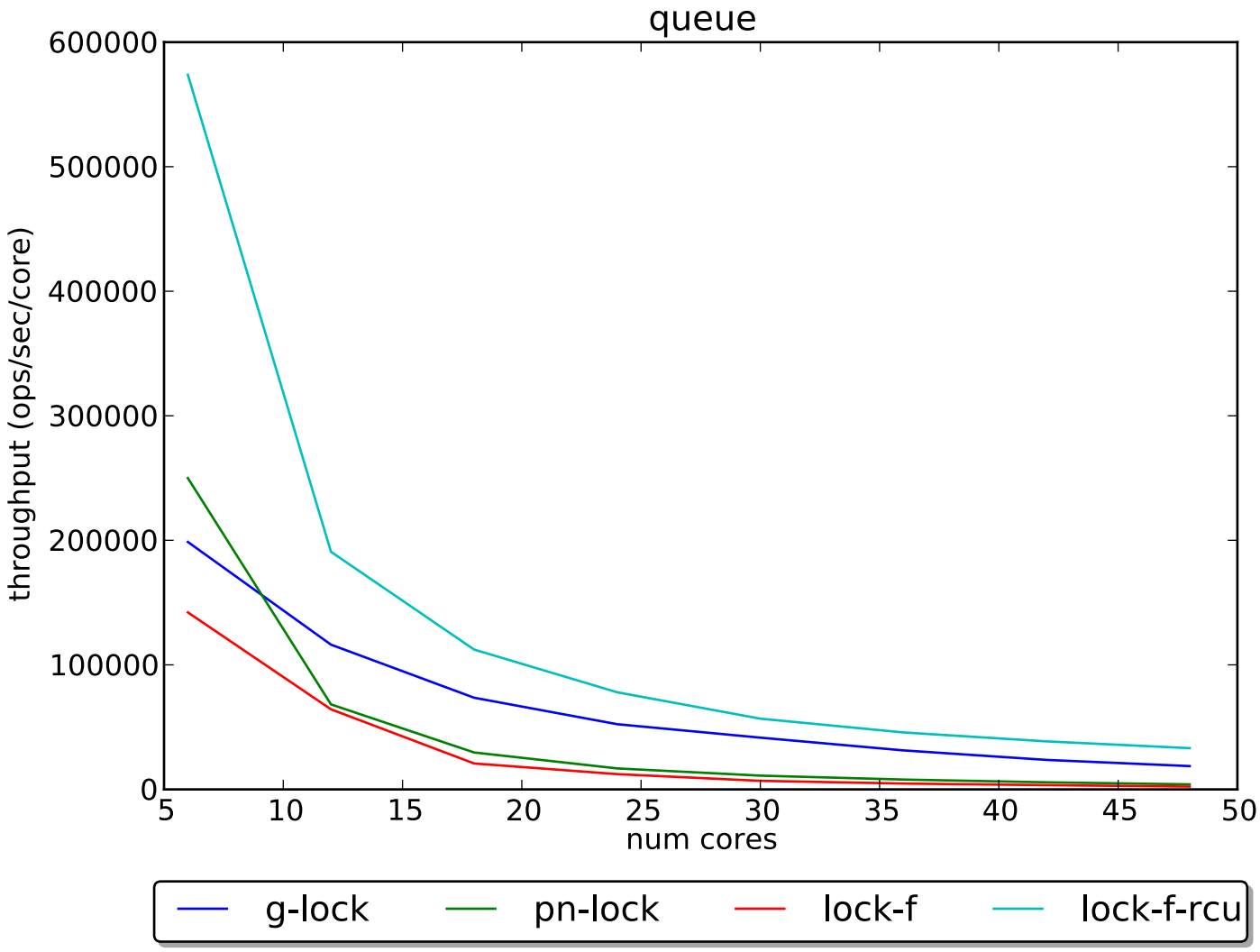
The big bakeoff

- So how do these implementations perform in practice?
- Two simple benchmarks:
  - Read only benchmark: N threads concurrently iterate over a small (100 elements) list.
  - Queue benchmark: N/2 threads concurrently append to the tail of a list, while N/2 threads concurrently remove from the head of the list.
- Machine specs:
  - 8x6 2.4GHz AMD Opteron
  - 64GB RAM
  - Linux 3.8

Read only benchmark



Queue benchmark



Conclusion

- Concurrent data structure programming is a very exciting and challenging area of computer science.
  - Things that we often take for granted suddenly become very important to think about.
- We *barely* scratched the surface today. Many more directions to go:
  - More formal reasoning about correctness and liveness.
  - More advanced data structures.
  - More primitive support from hardware, such as transactional memory.
  - Techniques and tools for debugging concurrent data structures.

Thanks! Questions?