But What About Updates?
Overview

- Aren't parallel updates a solved problem?
- Special cases for parallel updates
  - Split counters
  - Per-CPU/thread processing
  - Stream-based applications
  - Read-only traversal to location being updated
  - Allegiance-based updates
  - Hardware lock elision
- Possible additions to parallel-programming toolbox
Aren't Parallel Updates A Solved Problem?
Perfect partitioning leads to perfect performance and stunning scalability!
Parallel-Processing Workhorse: Hash Tables

Perfect partitioning leads to perfect performance and stunning scalability!

In theory, anyway...
Read-Mostly Workloads Scale Well: Hash Table
Update-Heavy Workloads, Not So Much...

And the horrible thing? Updates are all locking ops!
But Hash Tables Are Partitionable! What is Wrong?
But Hash Tables Are Partitionable! # of Buckets?

Some improvement, but...
But Hash Tables Are Partitionable! What is Wrong?

\begin{itemize}
\item NUMA effects:
  \begin{itemize}
  \item First eight CPUs on one socket, ninth on another
  \item No hash-bucket locality in workload: partitioned data, but not workload
  \item High cache-miss overhead: Buckets pass from one socket to the other
  \end{itemize}
\end{itemize}
Electrons move at 0.03C to 0.3C in transistors and, so need locality of reference.
Problem With Physics #1: Finite Speed of Light
Problem With Physics #2: Atomic Nature of Matter

Source

No complaints for eons, and now, suddenly, we’re too $\text{big}$?!.

I feel so fat!

Base

Drain

And our dielectric constant isn’t big enough for them! They can go find some other $#*$@ atom! Sheesh!
Read-Only Accesses Dodge The Laws of Physics!!!

Read-only data remains replicated in all caches
Updates, Not So Much...

Read-only data remains replicated in all caches, but each update destroys other replicas!
But Hash Tables Are Partitionable! What is Wrong?

- **NUMA effects:**
  - First eight CPUs on one socket, ninth on another
  - No hash.bucket locality in workload: partitioned data, but not workload
  - High cache-miss overhead: Buckets pass from one socket to the other

- **Can avoid NUMA effects:**
  - Partition hash buckets over NUMA nodes
    - Just like distributed systems do: See Dynamo paper
  - Use tree instead of hash table and do range partitioning
  - Do range partitioning across multiple hash tables, one per socket
  - If moderate number of updates and lots of memory, replicate hash table, one instance per socket
  - Minimize update footprint: Fine-grained locking
    - But if you tune your hash tables properly, this buys you little
  - Hardware transactional memory: Avoid locking overhead
    - More on this later in this presentation
  - Quantum entangled update???
Update-Heavy Workloads Painful for Parallelism!!!
But There Are Some Special Cases...
But There Are Some Special Cases

- Split counters (used for decades)
- Per-CPU/thread processing (perfect partitioning)
  - Huge number of examples, including the per-thread/CPU stack
  - But not everything can be perfectly partitioned
- Stream-based applications
- Read-only traversal to location being updated
- Hardware lock elision
- The Issaquah Challenge
Split Counters
Split Counters

- Have a per-CPU/thread counter: DEFINE_PER_CPU(u32, ctr);
- For updates, CPU/thread non-atomically updates its own counter
- For reads, sum all the counters
- Rely on commutative and associative laws of addition
- Plus rely on short-term inaccuracy permitted for statistical counters
- Constant work done for updates, linear scaling, great performance
Split Counters Diagram

Increment only your own counter
Split Counters Diagram

Counter 0
Counter 1
Counter 2
Counter 3
Counter 4
Counter 5

Sum all counters
Split Counters Diagram

Counter 0
Counter 1
Counter 2
Counter 3
Counter 4
Counter 5

Sum all counters
While they continue changing

It is possible to avoid the O(n) behavior on reads, see Counting chapter of “Is Parallel Programming Hard, And, If So, What Can You Do About It?”
Split Counters: What If You Need Them To Keep Still?

```c
DEFINE_PER_CPU(count);
br_read_lock();
this_cpu_inc(count);
br_read_unlock();

sum = 0;
br_write_lock();
for_each_possible_cpu(cpu)
    sum += per_cpu(count, cpu);
br_write_unlock();
```
DEFINE_PER_CPU(count);
br_read_lock();
this_cpu_inc(count);
br_read_unlock();

sum = 0;
br_write_lock();
for_each_possible_cpu(cpu)
  sum += per_cpu(count, cpu);
br_write_unlock();

Yes, the read lock guard updates and the write lock guards reads. This is why we now have lglocks (local-global locks)
Perfect Partitioning
Perfect Partitioning

- **Sharded lists**
  - Given element in partition, modified only by CPUs in that partition
    - Partition by key range
    - Partition by hashed value (favorite of Google, Amazon, ...)
    - Forward update to CPU in the corresponding partition, see next section
  - Set as special case of list
  - Very fast for heavy update workloads, still suffer read-write misses

- **Localized caches**
  - For example, per-socket cache
  - Blazing lookup speed!!!
  - But beware of memory footprint and cache miss rates!

- **Per-CPU atomics help userspace per-CPU partitioning**

- **Honorable mention: Queued locking**
Stream-Based Applications
Stream-Based Applications

- Adrian Sutton of LMAX presented this at linux.conf.au 2013:
  - http://www.youtube.com/watch?v=UvE389P6Er4
  - http://mechanical-sympathy.blogspot.com/

- Only two threads permitted to access a given location

- Use fixed-array circular FIFOs to pipe data between data-processing stages (represented by individual threads/CPUs)
  - Confining a processing stage to a single socket is not a bad plan. ;-)

- Get nearly uniprocessor performance, especially for heavy-weight processing
Example Stream-Based Application
Read-Only Traversal To Location Being Updated
Read-Only Traversal To Update Location

- Consider a radix tree

- Classic locking methodology would:
  1) Lock root
  2) Use fragment of key to select descendant
  3) Lock descendant
  4) Unlock previous node
  5) Repeat from step (2)

- The lock contention on the root is not going to be pretty!
Better Read-Only Traversal To Update Location

- Improved locking methodology might:
  - rcu_read_lock()
  - Traversal:
    - Start at root without locking
    - Use fragment of key to select descendant
    - Repeat until update location is reached
    - Acquire locks on update location
    - Do consistency checks, retry from root if inconsistent
  - Carry out update
  - rcu_read_unlock()

- Eliminates contention on root node!

- But need some sort of consistency-checks mechanism...
  - Sequence locking
  - “Deleted” flags on individual data elements
Sequence-Locked Read-Only Traversal

- for (;;)
  - rcu_read_lock()
  - seq = read_seqbegin(&myseq)
  - Start at root without locking
  - Use fragment of key to select descendant
  - Repeat until update location is reached
  - Acquire locks on update location
  - If (!read_seqretry(&myseq, seq))
    - break
  - Release locks on update location and rcu_read_unlock()

- Carry out update

- Release locks on update location and rcu_read_unlock()
Sequence-Locked Read-Only Traversal

- `for (;;)`
  - `rcu_read_lock()`
  - `seq = read_seqbegin(&myseq)`
  - Start at root without locking
  - Use fragment of key to select descendant
  - Repeat until update location is reached
  - Acquire locks on update location
  - If (!read_seqretry(&myseq, seq))
    - break
  - Release locks on update location and `rcu_read_unlock()`

- Carry out update

- Release locks on update location and `rcu_read_unlock()`

- But tree-shape updates must `write_seqcount_begin`

  dcache does something sort of like this, see `d_move()`.
Deletion-Flagged Read-Only Traversal

- for (;;)
  - rcu_read_lock()
  - Start at root without locking
  - Use fragment of key to select descendant
  - Repeat until update location is reached
  - Acquire locks on update location
  - If update location's deleted flag is not set:
    - break
  - Release locks on update location
  - rcu_read_unlock()

- Carry out update
- Release locks on update location and rcu_read_unlock()

Could dcache do something like this?
Read-Only Traversal To Location Being Updated

- Focus contention on portion of structure being updated
- Of course, full partitioning is better!
- But why not automate read-only traversal?
Hardware Lock Elision
Hardware Lock Elision

- If two lock-based critical sections have no conflicting accesses, why serialize them?
  - Conflicting access: concurrent accesses to the same location, at least one of which is a write

- Recent hardware from IBM and Intel supports this notion
  - Andi Kleen's ACM Queue article: http://queue.acm.org/detail.cfm?id=2579227
  - http://www.power.org/documentation/power-isa-version-2-07/

- Good results for some benchmarks on smallish systems:
Is Hardware Lock Elision The Silver Bullet?

- Some drawbacks:
  - Must have software fallback (aside from small mainframe transactions)
    - Not a cure-all for lock-based deadlocks
    - However, in some cases, might allow coarser locking
  - Still must avoid conflicting accesses
    - “Some restructuring may be required”
    - Even when the software does not care about the conflicts
  - Critical section's data references must fit into cache (Intel optimizes)
  - Critical section cannot contain irrevocable operations (like syscalls)
  - “Lemming effect”: self-perpetuating software fallback
  - Does not repeal the laws of physics
    - Speed of light and size of atoms remain the same :-)
  - Does not match the 2005 hype (but what would?)

- No silver bullet, but promising for a number of cases
Hardware Lock Elision: Toy Example

- Toy problem: Create a dequeue that can operate in parallel
  - Difficult to create lock-based dequeue that is parallel at both ends
  - Problem: Level of concurrency varies with dequeue state
Hardware Lock Elision: Toy Example

- Toy problem: Create a dequeue that can operate in parallel
  - Difficult to create lock-based dequeue that is parallel at both ends
  - Problem: Level of concurrency varies with dequeue state
  - But is this really a hard problem?
Hardware Lock Elision: Lock-Based Solution

- Use two lock-based dequeues
  - Can always insert concurrently: grab own dequeue's lock
  - Can always remove concurrently unless one or both are empty
    - If yours is empty, grab both locks in order!
    - Not hard, regardless of what a number of ca. 2005 papers might say :-)

![DIagram](image.png)
Hardware Lock Elision: Lock-Elision Solution

- But lock elision is even easier:
  - One dequeue protected by one lock!
  - The hardware automatically runs parallel when it is safe to do so
Hardware Lock Elision: Lock-Elision Solution

- But lock elision is even easier:
  - One dequeue protected by one lock!
  - The hardware automatically runs parallel when it is safe to do so

- However, there are some drawbacks (as always):
  - I/O, system calls, and other irrevocable operations defeat elision
  - Old hardware defeats elision
    - Though I am sure that both Intel and IBM would be more than happy to sell you some new hardware!
  - In many cases, restructuring required to avoid conflicting accesses
  - Hardware limitations (cache geometry, etc.) can defeat elision
  - Moderate levels of contention result in single-threaded execution even if the dequeue's state enables concurrent operation
Hardware Lock Elision: Lock-Elision Solution

- But lock elision is even easier:
  - One dequeue protected by one lock!
  - The hardware automatically runs parallel when it is safe to do so

- However, there are some drawbacks (as always):
  - I/O, system calls, and other irrevocable operations defeat elision
  - Old hardware defeats elision
    - Though I am sure that both Intel and IBM would be more than happy to sell you some new hardware!
  - In many cases, restructuring required to avoid conflicting accesses
  - Hardware limitations (cache geometry, etc.) can defeat elision
  - Moderate levels of contention result in single-threaded execution even if the dequeue's state enables concurrent operation

- But why are you putting everything through one dequeue???
Hardware Lock Elision: Potential Game Changers

What must happen for HTM to take over the world?
Hardware Lock Elision: Potential Game Changers

- Forward-progress guarantees
  - Mainframe is a start, but larger sizes would be helpful
- Transaction-size increases
- Improved debugging support
  - Gottschich et al: “But how do we really debug transactional memory?”
- Handle irrevocable operations (unbuffered I/O, syscalls, ...)
- Weak atomicity
Hardware Lock Elision: Potential Game Changers

- Forward-progress guarantees
  - Mainframe is a start, but larger sizes would be helpful
- Transaction-size increases
- Improved debugging support
  - Gottschich et al: “But how do we really debug transactional memory?”
- Handle irrevocable operations (unbuffered I/O, syscalls, ...)
- Weak atomicity – but of course the Linux-kernel RCU maintainer and weak-memory advocate would say that...
Hardware Lock Elision: Potential Game Changers

- **Forward-progress guarantees**
  - Mainframe is a start, but larger sizes would be helpful
- **Transaction-size increases**
- **Improved debugging support**
  - Gottschich et al: “But how do we really debug transactional memory?”
- **Handle irrevocable operations (unbuffered I/O, syscalls, …)**
- **Weak atomicity: It is not just me saying this!**
  - Herlihy et al: “Software Transactional Memory for Dynamic-Sized Data Structures”
  - Shavit: “Data structures in the multicore age”
  - Haas et al: “How FIFO is your FIFO queue?”
  - Gramoli et al: “Democratizing transactional memory”
- **With these additions, much greater scope possible**
Hardware Lock Elision: Potential Game Changers

- But transactional memory can do complex updates atomically – And just how are you going to do that with locking???
- So, beyond a certain point, isn't TM the only game in town?
Atomic Multi-Structure Update: Issaquah Challenge
Atomic Multi-Structure Update: Issaquah Challenge

Atomically move element 1 from left to right tree
Atomically move element 4 from right to left tree
Atomic Multi-Structure Update: Issaquah Challenge

Atomically move element 1 from left to right tree
Atomically move element 4 from right to left tree
Without contention between the two move operations!
But let's go one better: Do both moves as one atomic operation!
But let's go one better: Do both moves as one atomic operation! Better yet, any group of moves, adds, and deletes between any combination of linked data structures!!! After all, why mess around?
Locking Regions for Binary Search Tree
Possible Upsets While Acquiring Locks...

Before

After

What to do?
Possible Upsets While Acquiring Locks...

What to do? Drop locks and retry!!!
Allegiance Structures

- Each data element has an allegiance pointer
- NULL pointer says “member of current structure”
- Non-NULL pointer references an allegiance structure
  - Allegiance of multiple data elements can be switched atomically
Allegiance Structures

- Each data element has an allegiance pointer
- NULL pointer says “member of current structure”
- Non-NULL pointer references an allegiance structure
  - Allegiance of multiple data elements can be switched atomically
  - Easy to say, right? :-)

© 2014 IBM Corporation
Example Allegiance Structure Before Switch

- Data Structure A
  - Allegiance
    - Data Element 1
      - Allegiance
        - Offset=0

- Data Structure B
  - Allegiance
    - Data Element 1
      - Allegiance
        - Offset=1

- Allegiance Switch
  - 0
    - ENOENT
  - ENOENT
    - 0
Example Allegiance Structure After Switch

Data Structure A

Allegiance

Data Element 1

Data Structure B

Allegiance

Offset=0

Allegiance

Offset=1

Allegiance Switch

0

-ENOENT

-ENOENT

0
Example Allegiance Structure: Abbreviation

Data Structure A

Allegiance
Data Element 1

Allegiance
Offset=0

Allegiance Switch

Allegiance
Data Element 1

Allegiance
Offset=1

-ENOENT
0

-ENOENT
0

0

Data Structure B
Abbreviated Allegiance Switch Operation (1/6)

Initial state: First tree contains 1,2,3, second tree contains 2,3,4.
First tree contains 1,2,3, second tree contains 2,3,4.
After insertion, same: First tree contains 1,2,3, second tree contains 2,3,4.
After allegiance switch: First tree contains 2, 3, 4, second tree contains 1, 2, 3. Transition is single store, thus atomic! (But lookups need barriers in this case.)
Abbreviated Allegiance Switch Operation (5/6)

Unlink old nodes and allegiance structure
Abbreviated Allegiance Switch Operation (6/6)

After waiting a grace period, can free up allegiance structures and old nodes
Allegiance Structures

- Each data element has an allegiance pointer
- NULL pointer says “member of current structure”
- Non-NULL pointer references an allegiance structure
  - Allegiance of multiple data elements can be switched atomically
  - Easy to say, right? :-)
- And, given a reasonable API, not all *that* hard to do!!!
  - But what about performance and scalability???
Allegiance Structures: Performance and Scalability

90% lookups, 3% insertions, 3% deletions, 3% full tree scans, 1% moves
Allegiance Structures: Performance and Scalability

100% moves
Allegiance Structures: Performance and Scalability

- Bottlenecks include alignment and memory allocation

100% moves
Providing perfect performance and scalability is like committing the perfect crime. There are 50 things that might go wrong, and if you are a genius, you might be able to foresee and prevent 25 of them.”
– Paraphrased from Body Heat, with apologies to Kathleen Turner fans

Issues thus far:
– Getting possible-upset checks right
– Non-scalable random-number generator
– Non-scalable memory allocator
– Node alignment (false sharing)
– Premature deletion of moved elements (need to remove allegiance!)
– Unbalanced trees (false sharing)
– User-space RCU configuration (need per-thread call_rcu() handling)
– Getting memory barriers correct (probably more needed here)
– Threads working concurrently on adjacent elements (false sharing)
– Need to preload destination tree for move operations (contention!)
– Probably more left!!!
Allegiance Structures: Known Antecedents

- **Fraser**: “Practical Lock-Freedom”, Feb 2004
  - Insistence on lock freedom: Great complexity, poor performance
  - Similarity between Fraser's OSTM commit and allegiance switch

- **McKenney, Krieger, Sarma, & Soni**: “Atomically Moving List Elements Between Lists Using Read-Copy Update”, Apr 2006
  - Block concurrent operations while large update is carried out

- **Triplett**: “Scalable concurrent hash tables via relativistic programming”, Sept 2009

- **Triplett**: “Relativistic Causal Ordering: A Memory Model for Scalable Concurrent Data Structures”, Feb 2012
  - Similarity between Triplett's key switch and allegiance switch
  - Could share nodes between trees like Triplett does between hash chains, but would impose restrictions and API complexity
Special Cases For Parallel Updates: Summary

- There is currently no silver bullet:
  - Split counters
    - Extremely specialized
  - Per-CPU/thread processing
    - Not all algorithms can be efficiently partitioned
  - Stream-based applications
    - Specialized
  - Read-only traversal to location being updated
    - Great for small updates to large data structures, but limited otherwise
  - Hardware lock elision
    - Some good potential, and some potential limitations

- Linux kernel: Good progress by combining approaches
- Lots of opportunity for collaboration and innovation
Possible Additions To Parallel-Programming Toolbox
Possible Additions To Parallel-Programming Toolbox

- **OpLog for update-mostly operations**
  - Each CPU/thread has timestamped operation log, updates can cancel
  - Read operations force updates to be applied, as do some updates
  - Prototyped for Linux-kernel rmap with good results

- **The scalable commutativity rule**
  - Operations that cannot commute imply scalability bottleneck
    - fork()/exec() does not commute with other threads' address-space, file-descriptor, or signal-state operations – a combined fork()/exec(), e.g., posix_spawn(), would commute (but good luck getting apps to use it!)
    - “Lowest available FD” rule limits multithreaded open/close performance
  - Excellent guide for future API design
  - Similar to [http://paulmck.livejournal.com/16478.html](http://paulmck.livejournal.com/16478.html)
    - But way more complete and precise

- And maybe also atomic moves based on allegiance...
Summary
Summary

- We are farther along with read-mostly methods than with update-heavy methods.
- But there are some good approaches for update-heavy workloads for some special cases:
  - Split counters
  - Per-CPU/thread processing
  - Stream-based applications
  - Read-only traversal to location being updated
  - Hardware lock elision
  - Some recent research might prove practical
- We can expect specialization for update-heavy workloads:
  - Though generality would be nice if feasible!
To Probe Deeper (1/4)

- Hash tables:
  - http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html Chapter 10

- Spit counters:
  - http://events.linuxfoundation.org/sites/events/files/slides/BareMetal.2014.03.09a.pdf

- Perfect partitioning
  - Candide et al: “Dynamo: amazon's highly available key-value store”
    - http://doi.acm.org/10.1145/1323293.1294281
    - http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html Section 6.5
  - McKenney: “Retrofitted Parallelism Considered Grossly Suboptimal”
    - https://www.usenix.org/conference/hotpar12/retro%EF%AC%81tted-parallelism-considered-grossly-sub-optimal
  - McKenney et al: “Experience With an Efficient Parallel Kernel Memory Allocator”
  - Bonwick et al: “Magazines and Vmem: Extending the Slab Allocator to Many CPUs and Arbitrary Resources”
    - http://static.usenix.org/event/usenix01/full_papers/bonwick/bonwick_html/
  - Turner et al: “PerCPU Atomics”
To Probe Deeper (2/4)

- Stream-based applications:
  - Sutton: “Concurrent Programming With The Disruptor”
    - http://www.youtube.com/watch?v=UvE389P6Er4
  - Thompson: “Mechanical Sympathy”
    - http://mechanical-sympathy.blogspot.com/

- Read-only traversal to update location
  - Arcangeli et al: “Using Read-Copy-Update Techniques for System V IPC in the Linux 2.5 Kernel”
    - https://www.usenix.org/legacy/events/usenix03/tech/freenix03/full_papers/arcangeli/arcangeli_html/index.html
  - Corbet: “Dcache scalability and RCU-walk”
    - https://lwn.net/Articles/419811/
  - Xu: “bridge: Add core IGMP snooping support”
  - Howard: “A Relativistic Enhancement to Software Transactional Memory”
  - McKenney et al: “URCU-Protected Hash Tables”
    - http://lwn.net/Articles/573431/
To Probe Deeper (3/4)

- Hardware lock elision: Overviews
  - Kleen: “Scaling Existing Lock-based Applications with Lock Elision”
    • http://queue.acm.org/detail.cfm?id=2579227

- Hardware lock elision: Hardware description
  - POWER ISA Version 2.07
    • http://www.power.org/documentation/power-isa-version-2-07/
  - Intel® 64 and IA-32 Architectures Software Developer Manuals
  - Jacobi et al: “Transactional Memory Architecture and Implementation for IBM System z”
    • http://www.microsymposia.org/micro45/talks-posters/3-jacobi-presentation.pdf

- Hardware lock elision: Evaluations
  - http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html Section 16.3

- Hardware lock elision: Need for weak atomicity
  - Herlihy et al: “Software Transactional Memory for Dynamic-Sized Data Structures”
    • http://research.sun.com/scalable/pubs/PODC03.pdf
  - Shavit et al: “Data structures in the multicore age”
    • http://doi.acm.org/10.1145/1897852.1897873
  - Haas et al: “How FIFO is your FIFO queue?”
    • http://dl.acm.org/citation.cfm?id=2414731
  - Gramoli et al: “Democratizing transactional programming”
    • http://doi.acm.org/10.1145/2541883.2541900
To Probe Deeper (4/4)

- Possible future additions
  - Boyd-Wickizer: “Optimizing Communications Bottlenecks in Multiprocessor Operating Systems Kernels”
  - McKenney: “N4037: Non-Transactional Implementation of Atomic Tree Move”
Legal Statement

▪ This work represents the view of the author and does not necessarily represent the view of IBM.

▪ IBM and IBM (logo) are trademarks or registered trademarks of International Business Machines Corporation in the United States and/or other countries.

▪ Linux is a registered trademark of Linus Torvalds.

▪ Other company, product, and service names may be trademarks or service marks of others.
Questions?

Use the right tool for the job!!!

Image copyright © 2004 Melissa McKenney