Generalized Construction of Scalable Concurrent Data Structures via Relativistic Programming

Josh Triplett

Portland State University

June 2, 2011
Overview

- Introduction to Relativistic Programming
- Precise definitions of the RP ordering primitives
- Programming model for relativistic programs
- General construction technique for relativistic data structures
Necessity of concurrent programming

• Say you wrote a program to do some CPU-bound computation
  • Protein folding
Necessity of concurrent programming

• Say you wrote a program to do some CPU-bound computation
  • Protein folding
  • Video compression
Necessity of concurrent programming

- Say you wrote a program to do some CPU-bound computation
  - Protein folding
  - Video compression
  - Reading and writing a data structure

```
a → b → c → d → e
```
Baseline
Faster CPU

Fold 16x as many proteins, or compress video 16x faster.
No effort required.
Multi-core CPU

No free improvement anymore

The First Nehalem Processor

A Modular Design for Flexibility
Programming complexity

• You now need to change your code to take advantage of concurrency
You now need to change your code to take advantage of concurrency

Tradeoff: Concurrency versus programming complexity
Distributed systems

- Partition data and algorithms
- Communicate via message passing
- No shared memory
- Often highly scalable
- Modern systems look like this, but provide a shared memory abstraction
End-to-end principle

• Avoid implementing a costly abstraction that every program must use but not every program benefits from
• The shared memory abstraction has a high cost
• We don’t want to force shared-memory programs to use pure message-passing, either
• Modern systems have highly optimized caches, shared addressing, and semi-coherent shared memory
• Scalable concurrent programs should embrace that architecture
Overview of concurrent programming techniques

• Mutual exclusion
  • Reader-writer locking
  • Fine-grained locking

• Non-blocking synchronization

• Transactional memory

• Relativistic programming
Mutual exclusion

- Wrap all your data structures in one big lock serializing access
Mutual exclusion

- Wrap all your data structures in one big lock serializing access
- If you’re very lucky, your code doesn’t get any slower
- Synchronization adds high overhead—hundreds of times the cost of the critical section
- No actual concurrency to take advantage of multiple cores
Reader-writer locking

- Allow multiple readers to run simultaneously
Reader-writer locking

- Allow multiple readers to run simultaneously
- Non-parallelizable parts of reader synchronization take hundreds of cycles
- Actual read-side critical section takes a few cycles
Reader-writer locking

- Allow multiple readers to run simultaneously
- Non-parallelizable parts of reader synchronization take hundreds of cycles
- Actual read-side critical section takes a few cycles
- No actual reader concurrency
Reader-writer locking

- Allow multiple readers to run simultaneously
- Non-parallelizable parts of reader synchronization take hundreds of cycles
- Actual read-side critical section takes a few cycles
- No actual reader concurrency
- Fairness problems
Fine-grained locking

- Associate locks with parts of the data structure
Fine-grained locking

- Associate locks with parts of the data structure
- Disjoint-access parallelism
Fine-grained locking

- Associate locks with parts of the data structure
- Disjoint-access parallelism
- Additional locks add cost in the uncontended case
- Could amortize lock cost
Fine-grained locking

- Associate locks with parts of the data structure
- Disjoint-access parallelism
- Additional locks add cost in the uncontended case
- Could amortize lock cost... but that gives us coarse-grained locking again
Status of mutual exclusion

- Many problems; mostly explored, though not necessarily solved
- Well-established programming model
Status of mutual exclusion

- Many problems; mostly explored, though not necessarily solved
- Well-established programming model
- Wildly successful—the standard approach
Status of mutual exclusion

• Many problems; mostly explored, though not necessarily solved
• Well-established programming model
• Wildly successful—the standard approach
• Remaining problems inherent
• Does not scale linearly, if at all
Non-blocking synchronization

- Optimistically assume no concurrent conflicting operation
- Roll back on conflict
Non-blocking synchronization

- Optimistically assume no concurrent conflicting operation
- Roll back on conflict
- Maximum disjoint-access parallelism
Non-blocking synchronization

- Optimistically assume no concurrent conflicting operation
- Roll back on conflict
- Maximum disjoint-access parallelism
- Equivalent to maximally fine-grained locking
Status of non-blocking synchronization

- Rollbacks cause “useless parallelism”
Status of non-blocking synchronization

- Rollbacks cause “useless parallelism”
- Limited conflict-detection primitives available (CAS, LL/SC)
- Primitives expensive (hundreds of cycles), and contentious
- Must fit all synchronization through these primitives—possibly introducing extra indirection or copies
Status of non-blocking synchronization

- Rollbacks cause “useless parallelism”
- Limited conflict-detection primitives available (CAS, LL/SC)
- Primitives expensive (hundreds of cycles), and contentious
- Must fit all synchronization through these primitives—possibly introducing extra indirection or copies
- No standard methodology
- Memory management unspecified
- Requires the entire algorithm to deal with concurrent memory accesses and the associated portability issues
Status of non-blocking synchronization

- Rollbacks cause “useless parallelism”
- Limited conflict-detection primitives available (CAS, LL/SC)
- Primitives expensive (hundreds of cycles), and contentious
- Must fit all synchronization through these primitives—possibly introducing extra indirection or copies
- No standard methodology
- Memory management unspecified
- Requires the entire algorithm to deal with concurrent memory accesses and the associated portability issues
- Easy if an implementation of your data structure already exists
- Otherwise, sufficient complexity to allow publishing a paper

“Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms” for examples
Status of non-blocking synchronization

- Rollbacks cause “useless parallelism”
- Limited conflict-detection primitives available (CAS, LL/SC)
- Primitives expensive (hundreds of cycles), and contentious
- Must fit all synchronization through these primitives—possibly introducing extra indirection or copies
- No standard methodology
- Memory management unspecified
- Requires the entire algorithm to deal with concurrent memory accesses and the associated portability issues
- Easy if an implementation of your data structure already exists
- Otherwise, sufficient complexity to allow publishing a paper
- ...which will have mistakes in it discovered years later (see “Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms” for examples)
Software transactional memory

- Simplified optimistic concurrency
- Wrap each logically "atomic" operation in a transaction
- Roll back conflicting transactions
Software transactional memory

• Simplified optimistic concurrency
• Wrap each logically “atomic” operation in a transaction
• Roll back conflicting transactions
• Builds chains of NBS operations automatically
Software transactional memory

- Simplified optimistic concurrency
- Wrap each logically “atomic” operation in a transaction
- Roll back conflicting transactions
- Builds chains of NBS operations automatically
- Maximum disjoint-access parallelism
Software transactional memory

- Simplified optimistic concurrency
- Wrap each logically “atomic” operation in a transaction
- Roll back conflicting transactions
- Builds chains of NBS operations automatically
- Maximum disjoint-access parallelism
- Rollbacks still cause useless parallelism
- Can hide complexity but not cost (see Keir Fraser’s implementation, which needs $3n + 1$ CAS operations)
Software transactional memory

- Simplified optimistic concurrency
- Wrap each logically “atomic” operation in a transaction
- Roll back conflicting transactions
- Builds chains of NBS operations automatically
- Maximum disjoint-access parallelism
- Rollbacks still cause useless parallelism
- Can hide complexity but not cost (see Keir Fraser’s implementation, which needs $3n + 1$ CAS operations)
- Slower than locking, particularly in the uncontended case
- Problems not well-explored like locking (see “Why The Grass May Not Be Greener On The Other Side: A Comparison of Locking vs. Transactional Memory”)
Hardware transactional memory

- “Hardware will solve our performance problems”
Hardware transactional memory

- “Hardware will solve our performance problems”
- Not generally available (see the failed Sun Rock CPU)
Hardware transactional memory

- “Hardware will solve our performance problems”
- Not generally available (see the failed Sun Rock CPU)
- Spurious failures due to hardware quirks such as cache-line associativity or capacity
Hardware transactional memory

- “Hardware will solve our performance problems”
- Not generally available (see the failed Sun Rock CPU)
- Spurious failures due to hardware quirks such as cache-line associativity or capacity
- Software fallbacks required for generality and portability
Common theme: ignoring concurrency

- Making sequential code work “concurrently”
- Letting you write sequential code without thinking about concurrency
- Disjoint-access parallelism at best
Embracing concurrency

- Joint-access parallelism
- Minimal synchronization overhead
Embracing concurrency

- Joint-access parallelism
- Minimal synchronization overhead
- Reader-writer locks had the right idea
- Readers can easily run concurrently
- Many read-mostly workloads exist
Embracing concurrency

- Joint-access parallelism
- Minimal synchronization overhead
- Reader-writer locks had the right idea
- Readers can easily run concurrently
- Many read-mostly workloads exist
- Should not allow readers to block writers or vice versa
- Should make the writer bear the synchronization overhead
Relativistic Programming

- Allow concurrent readers and writer
Relativistic Programming

- Allow concurrent readers and writer
- Avoid unnecessary communication between CPUs
- Don’t enforce a single global view of memory
- Provides joint-access parallelism

RP primitives:
- Read and write shared addresses with ordering guarantees, avoiding portability issues with concurrent memory accesses.
- In particular, publish and dereference pointers.
- Delineate readers using fast CPU-local operations
- Writer can wait for existing readers to finish
Relativistic Programming

- Allow concurrent readers and writer
- Avoid unnecessary communication between CPUs
- Don’t enforce a single global view of memory
- Provides joint-access parallelism
- RP primitives:
  - Read and write shared addresses with ordering guarantees, avoiding portability issues with concurrent memory accesses.
  - In particular, publish and dereference pointers.
Relativistic Programming

- Allow concurrent readers and writer
- Avoid unnecessary communication between CPUs
- Don’t enforce a single global view of memory
- Provides joint-access parallelism
- RP primitives:
  - Read and write shared addresses with ordering guarantees, avoiding portability issues with concurrent memory accesses.
  - In particular, publish and dereference pointers.
  - Delineate readers using fast CPU-local operations
  - Writer can wait for existing readers to finish
Example: insertion into a relativistic linked list

- Initial state of the list; writer wants to insert b.

- Initialize b’s next pointer to point to c.
- The writer can then “publish” b to node a’s next pointer.
- Readers can immediately begin observing the new node.
Example: insertion into a relativistic linked list

- Initial state of the list; writer wants to insert b.
- Initialize b’s next pointer to point to c
Example: insertion into a relativistic linked list

- Initial state of the list; writer wants to insert b.
- Initialize b’s next pointer to point to c
- The writer can then “publish” b to node a’s next pointer
Example: insertion into a relativistic linked list

- Initial state of the list; writer wants to insert b.
- Initialize b’s next pointer to point to c.
- The writer can then "publish" b to node a’s next pointer.
- Readers can immediately begin observing the new node.
Example: removal from a relativistic linked list

- Initial state of the list; writer wants to remove node b
- Sets a's next pointer to c, removing b from the list for all future readers
- Wait for existing readers to finish
- Once no readers can hold references to b, the writer can safely reclaim it.
Example: removal from a relativistic linked list

- Initial state of the list; writer wants to remove node b
- Sets a’s next pointer to c, removing b from the list for all future readers
Example: removal from a relativistic linked list

- Initial state of the list; writer wants to remove node b
- Sets a’s next pointer to c, removing b from the list for all future readers
- Wait for existing readers to finish
Example: removal from a relativistic linked list

- Initial state of the list; writer wants to remove node b
- Sets a’s next pointer to c, removing b from the list for all future readers
- Wait for existing readers to finish
- Once no readers can hold references to b, the writer can safely reclaim it.
RP versus NBS/TM

- RP does not attempt non-blocking writers
- RP never has to deal with rollbacks
- RP addresses portability issues with concurrent memory accesses
- RP provides semi-automatic garbage collection via “wait for readers to finish”
RP versus NBS/TM

- RP does not attempt non-blocking writers
- RP never has to deal with rollbacks
- RP addresses portability issues with concurrent memory accesses
- RP provides semi-automatic garbage collection via “wait for readers to finish”
- Still need to construct writers carefully to support concurrent readers
Status of Relativistic Programming

- Excellent concurrency
- Readers have near-perfect linear scalability
- Many existing data structures
Status of Relativistic Programming

- Excellent concurrency
- Readers have near-perfect linear scalability
- Many existing data structures
- Easy if an implementation of your data structure already exists
- Otherwise, sufficient complexity to allow publishing a paper
Status of Relativistic Programming

- Excellent concurrency
- Readers have near-perfect linear scalability
- Many existing data structures
- Easy if an implementation of your data structure already exists
- Otherwise, sufficient complexity to allow publishing a paper
- ...which will hopefully not have mistakes in it discovered years later
Concurrency versus complexity

Concurrenta

Complexity
Concurrency versus complexity

- Global locking

Concurrency

Complexity
Concurrency versus complexity

Concurrency

Complexity

Global locking

R/W locking

Fine-grained locking
Concurrency versus complexity

Concurrency

 Complexity

Global locking

R/W locking

Fine-grained locking

NBS

Publishable
Concurrency versus complexity

Concurrency

Complexity

Transactional memory
Fine-grained locking
R/W locking
Global locking

NBS
Publishable
Concurrency versus complexity

Concurrency

Complexity

Transactional memory

Fine-grained locking

R/W locking

Global locking

Publishable

RP

NBS
What makes Relativistic Programming hard?

- No established programming model
  - Mechanisms exist, but not methodology
- No reasoning model
- Violates expectations established by locking model
  - Memory can change out from under a reader
  - Unknown ordering guarantees
Contribution of this paper

- Precise definitions of the RP ordering primitives
- A standard programming model for relativistic programs
- A general construction technique using the RP ordering primitives to enforce the programming model, providing relativistic algorithms for arbitrary data structures
Memory ordering terminology

- Readers consist of a series of reads: $R_1, R_2, \ldots, R_m$
- Writers consist of a series of writes: $W_1, W_2, \ldots, W_m$
- Readers and writers have an order:
  - $R_i \prec R_j$ if $R_i$ comes before $R_j$ in reader program order
  - $W_i \prec W_j$ if $W_i$ comes before $W_j$ in writer’s desired partial order
- Writers may relax ordering between two writes
Memory ordering terminology: addresses

- Reads and writes have a target address
- A read at an address return a value written by some previous write to that address.
- $R_i$ observes $W_j$ if $R_i$ returns the value written by $W_j$
- Otherwise, $R_i$ fails to observe $W_j$
Read barriers

- A read barrier enforces causality between ordered reads.
- Given a reader performing reads $R_i$ and $R_j$, and a read barrier $RB$ such that $R_i \prec RB \prec R_j$, if $R_i$ observes a write $W_k$ at the same address as $R_i$, then $R_j$ will not fail to observe any write ordered before $W_k$.
- In other words: if the earlier read observes a write, the later read will not fail to observe an earlier write.
Write barriers

- A write barrier enforces causality between ordered writes.
- Given a writer performing writes \( W_i \) and \( W_j \), and a write barrier \( WB \) such that \( W_i \prec WB \prec W_j \), if a reader performs read operations \( R_k \) and \( R_l \) at the same addresses as \( W_i \) and \( W_j \) respectively, and \( R_l \prec R_k \), then if \( R_l \) observes \( W_j \), \( R_k \) will observe \( W_i \).
- In other words: if the earlier read observes the later write, the later read will observe the earlier write.
Wait-for-readers operation

- Wait-for-readers enforces ordering between write operations and an entire reader.
- Given a writer performing writes $W_i$ and $W_j$, and a wait-for-readers operation $B$ such that $W_i \prec B \prec W_j$, if any read $R_k$ at the same address as $W_i$ fails to observe $W_i$, no other read in the same reader as $R_k$ may observe $W_j$.
- In other words: if any read fails to observe the earlier write, no read in the same reader may observe the later write.
Common elements of relativistic write algorithms

• Avoid disrupting current readers
• Maintain data structure consistency after each operation
• Observe details of reader traversal algorithm
• In particular, take advantage of traversal order
• Use the “wait for readers” operation to stop considering old readers; new readers will see operations done so far
Common concern: ordering of write operations

• As seen by readers
• If a writer does N operations in order, we don’t want to consider every possible subset of those operations
• Much easier to consider every prefix—each operation must keep structure consistent for readers
• Equivalent to saying a new reader may start and run to completion at any time
Common concern: ordering of write operations

- As seen by readers
- If a writer does N operations in order, we don’t want to consider every possible subset of those operations
- Much easier to consider every prefix—each operation must keep structure consistent for readers
- Equivalent to saying a new reader may start and run to completion at any time
- Less strict than mutual exclusion, which makes the block of operations atomic
- Still straightforward to reason about
Ordering property for relativistic programming

- Given a reader performing two reads $R_i$ and $R_j$, and a writer performing two writes $W_k$ and $W_l$ at the same addresses as $R_i$ and $R_j$ respectively, if $W_k \prec W_l$, then we do not allow the case where $R_i$ fails to observe $W_k$ but $R_j$ observes $W_l$.

- Note in particular that we require this property regardless of the ordering of $R_i$ and $R_j$ within the reader.

- In other words: readers appear to occur atomically between two write operations, and don’t observe concurrent writes.
Examples of the relativistic ordering property

- First write initializes a node, second write inserts it into a data structure. Ordering prevents access to uninitialized memory.

- First write detaches a node from a data structure, second write reclaims the memory of the node. Ordering prevents access to reclaimed memory.

- First write inserts a copy of a node earlier in a data structure, second write removes the original copy later in the data structure. Ordering ensures the node doesn't go missing.

(Trees, hash tables)
Examples of the relativistic ordering property

- First write initializes a node, second write inserts it into a data structure. Ordering prevents access to uninitialized memory.
- First write detaches a node from a data structure, second write reclaims the memory of the node. Ordering prevents access to reclaimed memory.
Examples of the relativistic ordering property

- First write initializes a node, second write inserts it into a data structure. Ordering prevents access to uninitialized memory.
- First write detaches a node from a data structure, second write reclaimed the memory of the node. Ordering prevents access to reclaimed memory.
- First write inserts a copy of a node earlier in a data structure, second write removes the original copy later in the data structure. Ordering ensures the node doesn’t go missing. (Trees, hash tables)
How do we enforce this ordering property?

- We want readers to observe writes in the order writers perform them
- We have two ordering operations:
  - Write barrier
  - Wait-for-readers
- Both operations delay the writer to enforce ordering
- When do we use each operation?
Three cases

• We have writes $W_i$ and $W_j$ such that $W_i \prec W_j$.
• What barrier do we put between the writes?
• We have three cases, depending on reader traversal order:
  • If no reader performs reads at both of those addresses, no reader may violate the ordering constraint, so the writer needs no barrier
  • If readers perform reads $R_k$ and $R_l$ at the same addresses as $W_i$ and $W_j$ respectively, the read order matters:
    • If the reads occur in reverse order of the writes ($R_l \prec R_k$), a write barrier suffices: if $R_l$ observes $W_j$, then $R_k$ will observe $W_i$.
    • If the reads occur in the same order as the writes ($R_k \prec R_l$), the writer needs to wait for readers: if $R_k$ fails to observe $W_i$, then $R_l$ cannot observe $W_j$. 
When to use a write barrier

- Writes in reverse traversal order
- Reader reads position of $W_2$ then position of $W_1$
- Write barrier delays $W_2$ until $W_1$ globally visible
- If a reader sees $W_2$, it will subsequently see $W_1$
When to use a write barrier

- Writes in reverse traversal order
- Reader reads position of $W_2$ then position of $W_1$
- Write barrier delays $W_2$ until $W_1$ globally visible
- If a reader sees $W_2$, it will subsequently see $W_1$
- Makes writes become visible in writer program order
- Matches expectations
When to wait for readers

- Writes in traversal order
- Reader reads position of $W_1$ then position of $W_2$
- Write barrier provides no useful ordering guarantee
- Writer must not overtake a reader in progress
- Wait for readers to finish to flush out readers already past the position of $W_1$
- New readers will observe $W_1$ before $W_2$
When to wait for readers

- Writes in traversal order
- Reader reads position of $W_1$ then position of $W_2$
- Write barrier provides no useful ordering guarantee
- Writer must not overtake a reader in progress
- Wait for readers to finish to flush out readers already past the position of $W_1$
- New readers will observe $W_1$ before $W_2$
General construction of relativistic data structures

- Each write operation must maintain consistency for readers
- Readers may observe a prefix of a writer’s operations
- Readers may not observe writes inconsistently with a writer’s partial order
General construction of relativistic data structures

- Each write operation must maintain consistency for readers
- Readers may observe a prefix of a writer’s operations
- Readers may not observe writes inconsistency with a writer’s partial order
- When performing a series of causally related operations, whose order matters to readers:
  - Between writes where no reader observes both, use no barrier
  - Between writes in reader traversal order, wait for readers
  - Between writes in reverse traversal order, use a write barrier
Questions?
Image credits

- DVD from Wikimedia Commons, by “Lorian” and “Tene”;

- Protein folding from Wikimedia Commons, by “DrKjaergaard”;
  http://en.wikipedia.org/wiki/File:Protein_folding.png

- Pentium with MMX Technology from Wikimedia Commons, by “Denniss”;

- Pentium 4 from Wikimedia Commons, by “Eric Gaba”;

- Nehalem die callout from Intel;
  http://www.intel.com/pressroom/archive/releases/2008/20081117comp_sm.htm