

Portland State University

PDXScholar

Computer Science Faculty Publications and
Presentations

Computer Science

6-2011

Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming

Josh Triplett

Portland State University

Paul E. McKenney

Jonathan Walpole

Portland State University

Follow this and additional works at: https://pdxscholar.library.pdx.edu/compsci_fac



Part of the [Computer and Systems Architecture Commons](#), and the [Software Engineering Commons](#)

Let us know how access to this document benefits you.

Citation Details

Josh Triplett, Paul E. McKenney, and Jonathan Walpole, "Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming" (2011). Presentation to USENIX Annual Technical Conference (USENIX ATC'11).

This Presentation is brought to you for free and open access. It has been accepted for inclusion in Computer Science Faculty Publications and Presentations by an authorized administrator of PDXScholar. Please contact us if we can make this document more accessible: pdxscholar@pdx.edu.

Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming

Josh Triplett¹ Paul E. McKenney² Jonathan Walpole¹

¹Portland State University

²IBM Linux Technology Center

June 16, 2011

Synchronization = Waiting

- Concurrent programs require synchronization
- Synchronization requires some threads to **wait** on others
- Concurrent programs spend a lot of time **waiting**

Locking

- One thread accesses shared data
- The rest **wait** for the lock

Locking

- One thread accesses shared data
- The rest **wait** for the lock
- Straightforward to get right
- Minimal concurrency

Fine-grained Locking

- Use different locks for different data
- Disjoint-access parallelism
- Reduce **waiting**, allow multiple threads to proceed

Fine-grained Locking

- Use different locks for different data
- Disjoint-access parallelism
- Reduce **waiting**, allow multiple threads to proceed
- Many expensive synchronization instructions

Fine-grained Locking

- Use different locks for different data
- Disjoint-access parallelism
- Reduce **waiting**, allow multiple threads to proceed
- Many expensive synchronization instructions
- **Wait** on memory
- **Wait** on the bus
- **Wait** on cache coherence

Reader-writer locking

- Don't make readers **wait** on other readers
- Readers still **wait** on writers and vice versa

Reader-writer locking

- Don't make readers **wait** on other readers
- Readers still **wait** on writers and vice versa
- Same expensive synchronization instructions
- Dwarfs the actual reader critical section

Reader-writer locking

- Don't make readers **wait** on other readers
- Readers still **wait** on writers and vice versa
- Same expensive synchronization instructions
- Dwarfs the actual reader critical section
- No actual reader parallelism; readers get serialized

Non-blocking synchronization

- Right there in the name: non-blocking
- So, no **waiting**, right?

Non-blocking synchronization

- Right there in the name: non-blocking
- So, no **waiting**, right?
- Expensive synchronization instructions

Non-blocking synchronization

- Right there in the name: non-blocking
- So, no **waiting**, right?
- Expensive synchronization instructions
- All but one thread must retry
- Useless parallelism: **waiting** while doing busywork
- At best equivalent to fine-grained locking

Transactional memory

- Non-blocking synchronization made easy
- (Often implemented using locks for performance)

Transactional memory

- Non-blocking synchronization made easy
- (Often implemented using locks for performance)
- Theoretically equivalent performance to NBS
- In practice, somewhat more expensive

Transactional memory

- Non-blocking synchronization made easy
- (Often implemented using locks for performance)
- Theoretically equivalent performance to NBS
- In practice, somewhat more expensive
- Fancy generic abstraction wrappers around **waiting**

How do we stop waiting?

- Reader-writer locking had the right idea
- But readers needed synchronization to **wait** on writers
- Some **waiting** required to check for potential writers
- Can readers avoid synchronization entirely?

How do we stop waiting?

- Reader-writer locking had the right idea
- But readers needed synchronization to **wait** on writers
- Some **waiting** required to check for potential writers
- Can readers avoid synchronization entirely?
- Readers should not wait at all

How do we stop waiting?

- Reader-writer locking had the right idea
- But readers needed synchronization to **wait** on writers
- Some **waiting** required to check for potential writers
- Can readers avoid synchronization entirely?
- Readers should not wait at all
- Joint-access parallelism: Can we allow concurrent readers and writers on the same data at the same time?

How do we stop waiting?

- Reader-writer locking had the right idea
- But readers needed synchronization to **wait** on writers
- Some **waiting** required to check for potential writers
- Can readers avoid synchronization entirely?
- Readers should not wait at all
- Joint-access parallelism: Can we allow concurrent readers and writers on the same data at the same time?
- What does “at the same time” mean, anyway?

Modern computers

- Shared address space
- Distributed memory
- Expensive illusion of coherent shared memory

Modern computers

- Shared address space
- Distributed memory
- Expensive illusion of coherent shared memory
- “At the same time” gets rather fuzzy

Modern computers

- Shared address space
- Distributed memory
- Expensive illusion of coherent shared memory
- “At the same time” gets rather fuzzy
- Shared address spaces make communication simple
- Incredibly optimized communication via cache coherence

Modern computers

- Shared address space
- Distributed memory
- Expensive illusion of coherent shared memory
- “At the same time” gets rather fuzzy
- Shared address spaces make communication simple
- Incredibly optimized communication via cache coherence
- When we have to communicate, let’s take advantage of that!
- (and not just to accelerate message passing)

Relativistic Programming

- By analogy with relativity: no absolute reference frame
- No global order for non-causally-related events
- Readers do no waiting at all, for readers or writers
- Minimize expensive communication and synchronization
- Writers do all the waiting, when necessary
- Reads linearly scalable

What if readers see partial writes?

- Writers must not disrupt concurrent readers
- Data structures must stay consistent after every write
- Writers order their writes by waiting
- No impact to concurrent readers

Outline

- Synchronization = Waiting
- Introduction to Relativistic Programming
- Relativistic synchronization primitives
- Relativistic data structures
- Hash-table algorithm
- Results
- Future work

Relativistic synchronization primitives

- Delimited readers
 - No waiting: Notification, not permission

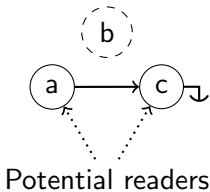
Relativistic synchronization primitives

- Delimited readers
 - No waiting: Notification, not permission
- Pointer publication
 - Ensures ordering between initialization and publication

Relativistic synchronization primitives

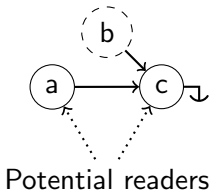
- Delimited readers
 - No waiting: Notification, not permission
- Pointer publication
 - Ensures ordering between initialization and publication
- Updaters can wait for readers
 - Existing readers only, not new readers

Example: Relativistic linked list insertion



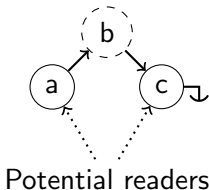
- Initial state of the list; writer wants to insert b.

Example: Relativistic linked list insertion



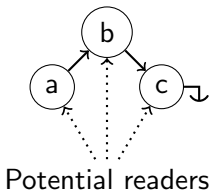
- Initial state of the list; writer wants to insert b.
- Initialize b's next pointer to point to c

Example: Relativistic linked list insertion



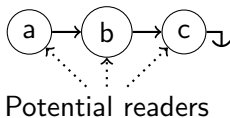
- Initial state of the list; writer wants to insert b.
- Initialize b's next pointer to point to c
- The writer can then “publish” b to node a's next pointer

Example: Relativistic linked list insertion



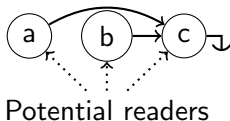
- Initial state of the list; writer wants to insert b.
- Initialize b's next pointer to point to c
- The writer can then “publish” b to node a's next pointer
- Readers can immediately begin observing the new node

Example: Relativistic linked list removal



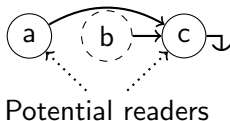
- Initial state of the list; writer wants to remove node b

Example: Relativistic linked list removal



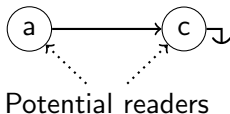
- Initial state of the list; writer wants to remove node b
- Sets a's next pointer to c, removing b from the list for all future readers

Example: Relativistic linked list removal



- 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: Relativistic linked list removal



- 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.

Relativistic data structures

- Linked lists
- Radix trees
- Tries
- Balanced trees
- Hash tables

Relativistic hash tables

- Open chaining with relativistic linked lists
- Insertion and removal supported
- Atomic move operation (see previous work)

Relativistic hash tables

- Open chaining with relativistic linked lists
- Insertion and removal supported
- Atomic move operation (see previous work)
- What about resizing?
- Necessary to maintain constant-time performance and reasonable memory usage

Relativistic hash tables

- Open chaining with relativistic linked lists
- Insertion and removal supported
- Atomic move operation (see previous work)
- What about resizing?
- Necessary to maintain constant-time performance and reasonable memory usage
- Must keep the table consistent at all times

Existing approaches to resizing

- Don't: allocate a fixed-size table and never resize it
 - Poor performance or wasted memory

Existing approaches to resizing

- Don't: allocate a fixed-size table and never resize it
 - Poor performance or wasted memory
- “Dynamic Dynamic Data Structures” (DDDS)
 - Readers must check old and new data structures
 - Readers have to **wait** until no concurrent resizes
 - Slows down the common case
 - Significantly slows lookups while resizing

Existing approaches to resizing

- Don't: allocate a fixed-size table and never resize it
 - Poor performance or wasted memory
- “Dynamic Dynamic Data Structures” (DDDS)
 - Readers must check old and new data structures
 - Readers have to **wait** until no concurrent resizes
 - Slows down the common case
 - Significantly slows lookups while resizing
- Herbert Xu's resizable relativistic hash tables
 - Extra linked-list pointers in every node
 - High memory usage

Defining “consistent”

- A reader traversing a hash bucket must always observe all elements in that bucket

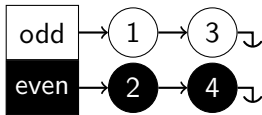
Defining “consistent”

- A reader traversing a hash bucket must always observe all elements in that bucket
- ...but if it observes more, no harm done

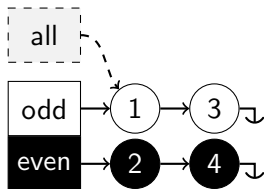
Defining “consistent”

- A reader traversing a hash bucket must always observe all elements in that bucket
- ...but if it observes more, no harm done
- *Imprecise* hash buckets contain elements from other buckets

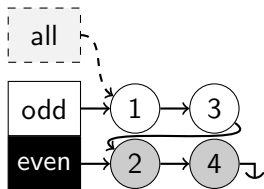
Shrinking: Initial state



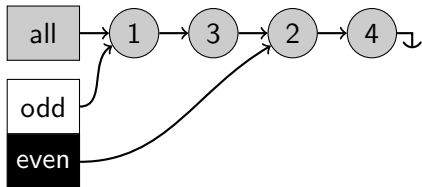
Shrinking: Initialize new buckets



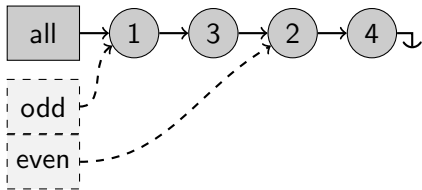
Shrinking: Link old chains



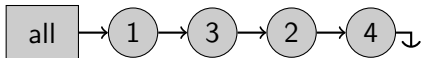
Shrinking: Publish new buckets



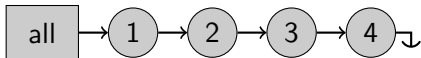
Shrinking: Wait for readers



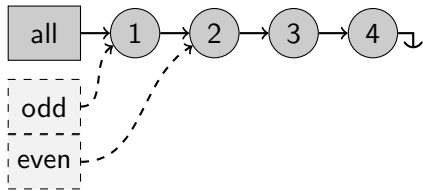
Shrinking: Reclaim



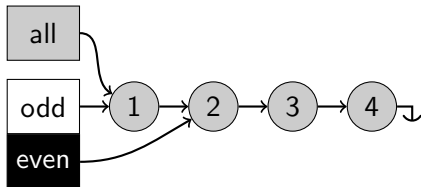
Expanding: Initial state



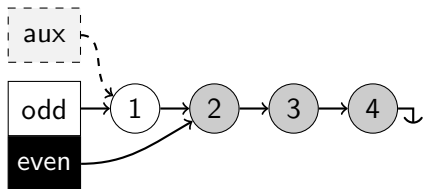
Expanding: Initialize new buckets



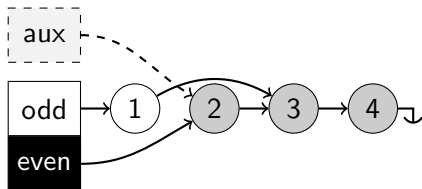
Expanding: Publish new buckets



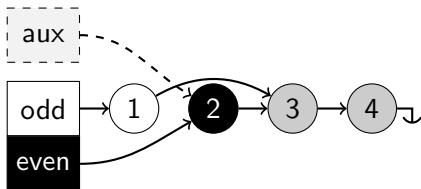
Expanding: Wait for readers



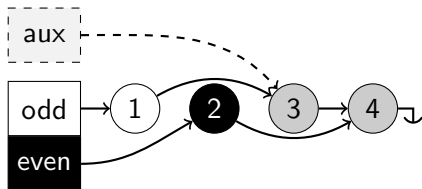
Expanding: Unzip one step



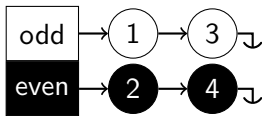
Expanding: Wait for readers



Expanding: Unzip again



Expanding: Final state



Benchmarking methodology

- Implemented a microbenchmark as a Linux kernel module
- Used Linux's Read-Copy Update (RCU) implementation
- Relativistic Programming primitives map to RCU operations

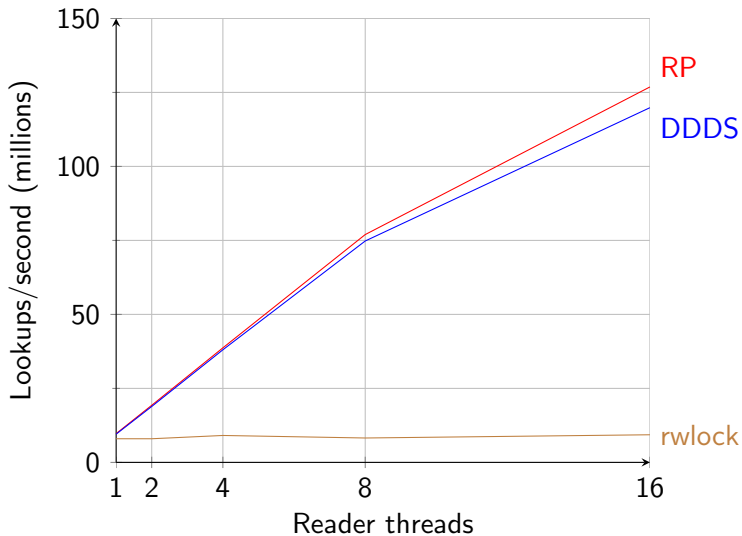
Benchmarking methodology

- Implemented a microbenchmark as a Linux kernel module
- Used Linux's Read-Copy Update (RCU) implementation
- Relativistic Programming primitives map to RCU operations
- Lookups with no resize as a baseline
- Lookups with continuous resizing as a worst-case scenario

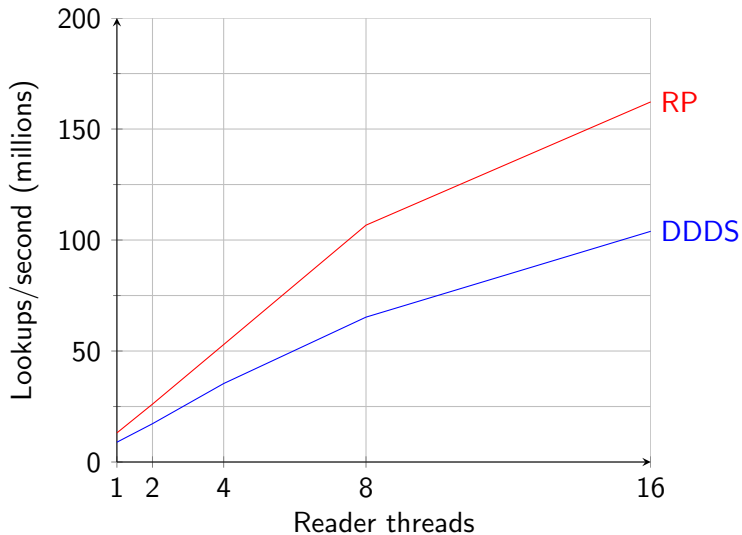
Benchmarking methodology

- Implemented a microbenchmark as a Linux kernel module
- Used Linux's Read-Copy Update (RCU) implementation
- Relativistic Programming primitives map to RCU operations
- Lookups with no resize as a baseline
- Lookups with continuous resizing as a worst-case scenario
- Compared: our algorithm, DDDS, rwlock

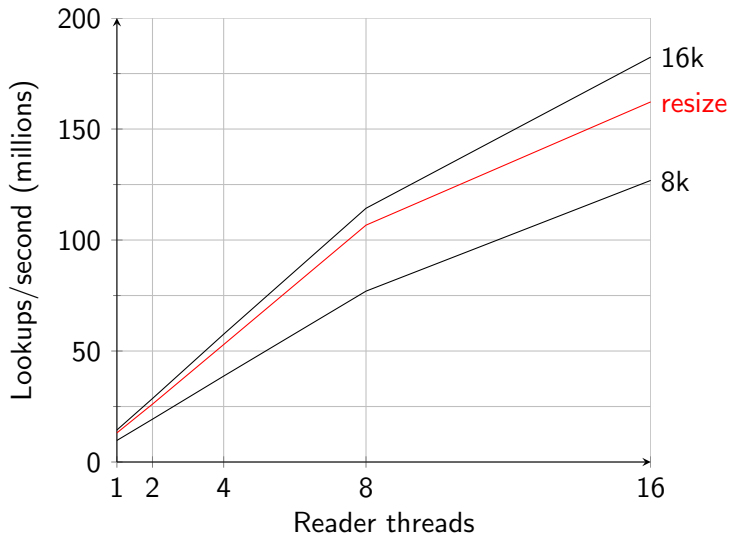
Results: fixed-size table baseline



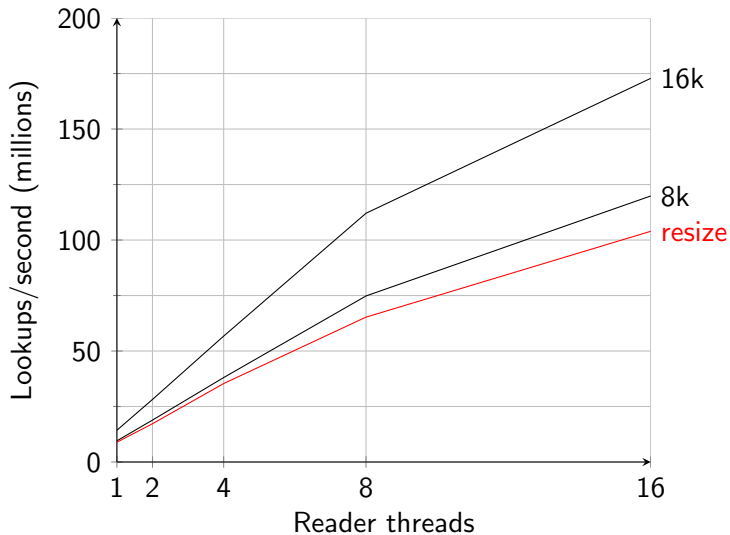
Results - continuous resizing



Results - our resize versus fixed



Results - DDDS resize versus fixed



Hang on a minute...

- This is USENIX!
- We don't settle for microbenchmarks here
- We care about real-world implementations

memcached

- Network-accessible key-value store
- Used for caching
- Performance-critical

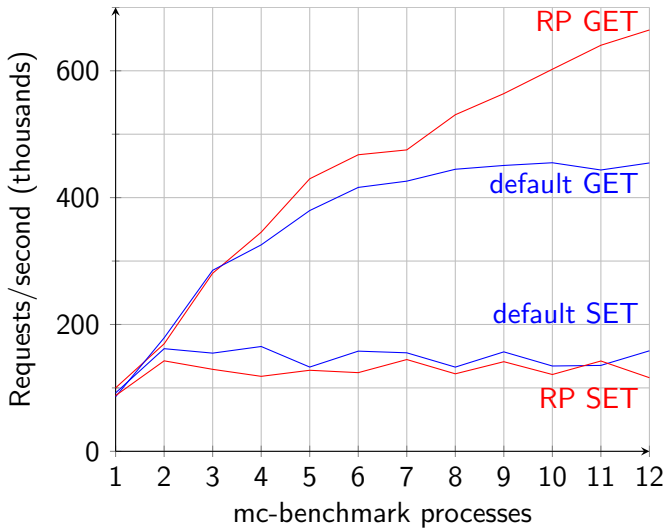
memcached

- Network-accessible key-value store
- Used for caching
- Performance-critical
- ...and it uses a global table lock

memcached with relativistic hash tables

- Uses the userspace RCU implementation, urcu
- Adds a fast path for GET requests using relativistic lookups
- Copies value while still in a relativistic reader
- Falls back to the slow path for expiry, eviction
- Writers use safe relativistic memory reclamation

memcached results



Future work: Relativistic data structures

- New relativistic algorithms currently require careful construction
- We have a general methodology for algorithm construction
 - Write an algorithm assuming our memory model
 - Use this methodology to mechanically place barriers and wait-for-readers operations

Summary

- Relativistic programming allows linearly scalable readers
- Relativistic hash tables support resizing now
 - Now suitable for general-purpose usage
- Real-world code scales better with relativistic programming

Questions?