An HTM-Based Update-side Synchronization for RCU on NUMA systems

SeongJae Park, Paul E. McKenney, Laurent Dufour, Heon Y. Yeom

amazon facebook IBM

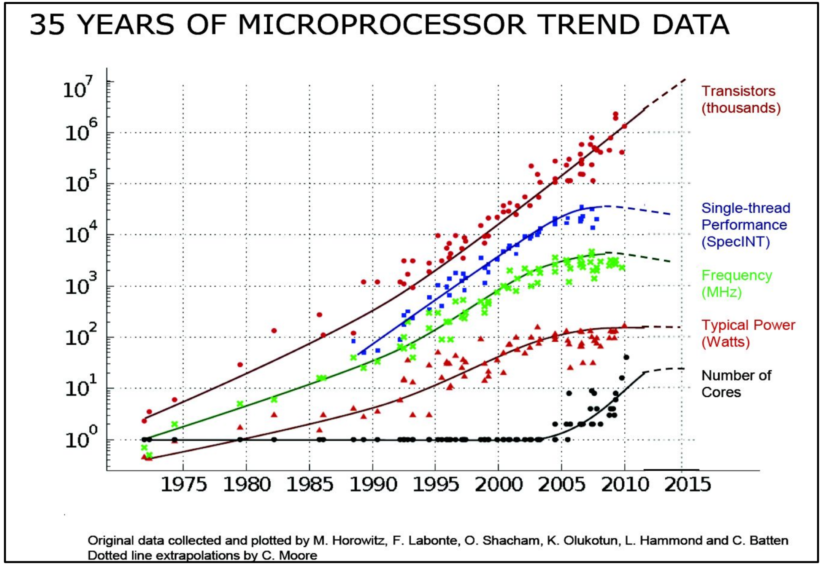
Seoul National University
College of Engineering
Dept. of Computer Science and Engineering

Disclaimer

- This work was done prior to the first author joining Amazon and while the second author was at IBM
- The views expressed herein are those of the authors; they do not reflect the views of their employers

The World Is In NUMA/Multi-CPU Era

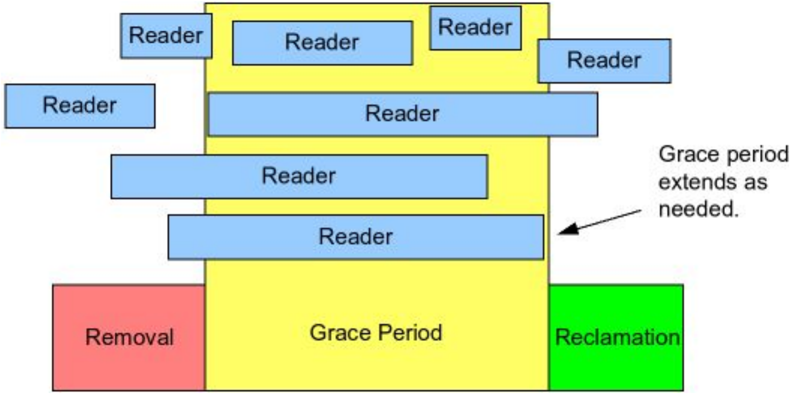
- More than a decade ago, world has changed to multi-CPU era
- Nowadays, huge NUMA systems utilizing hundreds of threads are common
- Efficient synchronization primitives are the key of performance and scalability



https://www.karlrupp.net/wp-content/uploads/2015/06/35years.png

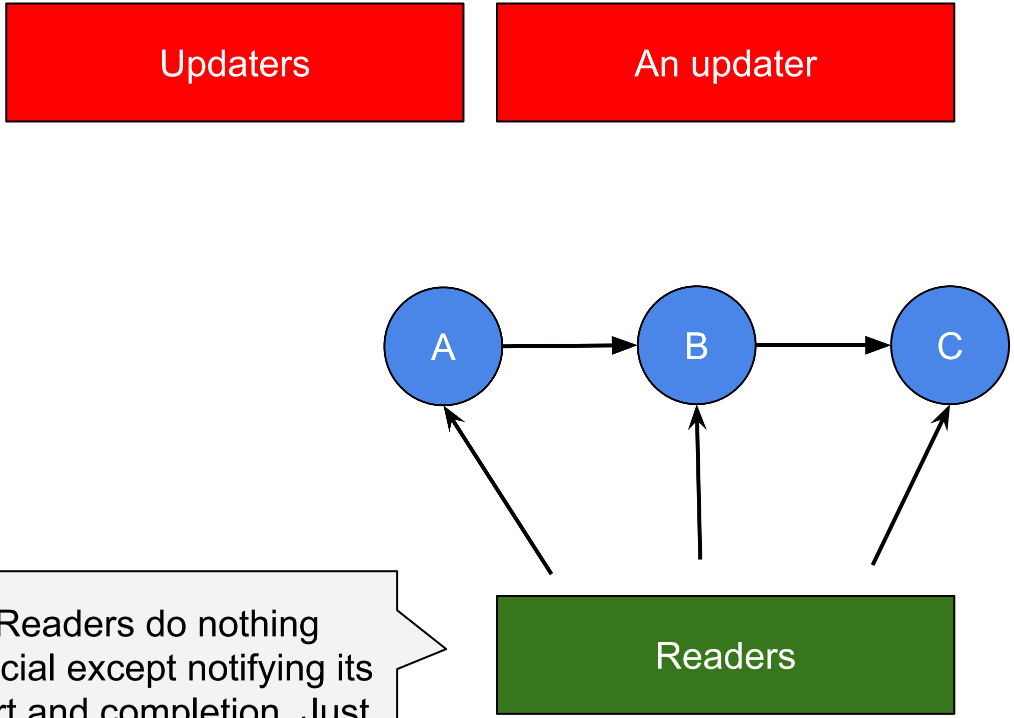
RCU: Read-Copy Update

- A synchronization mechanism for read-mostly workloads
- Provides almost ideal performance and scalability for reads



https://static.lwn.net/images/ns/kernel/rcu/GracePeriodGood.png

RCU-protected Linked List: Reading Items

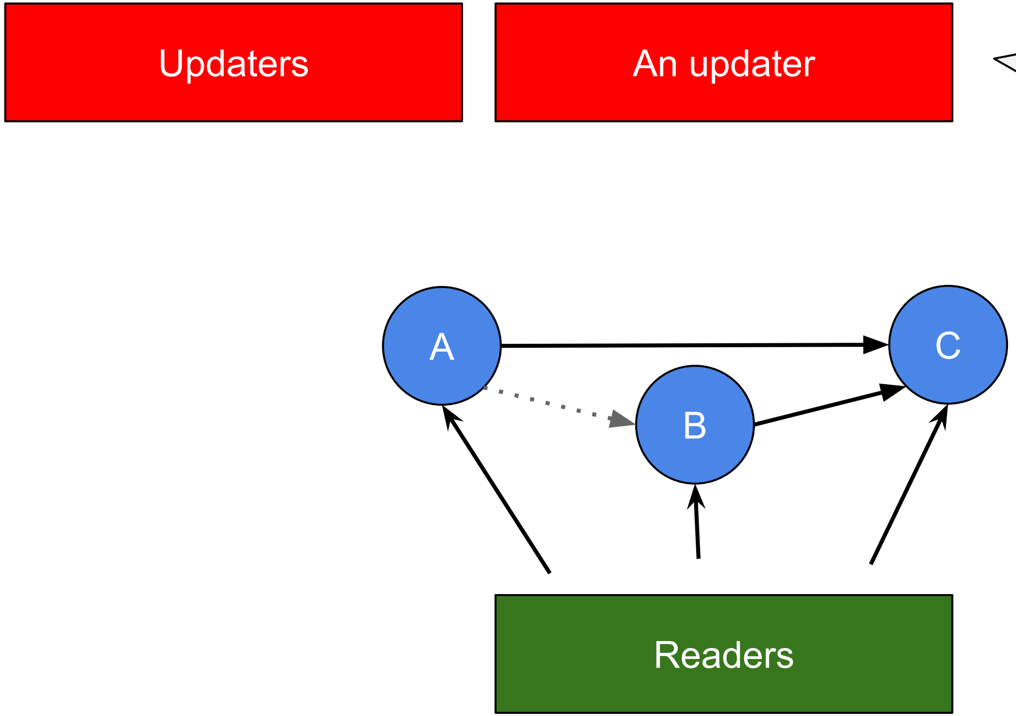


Readers do nothing special except notifying its start and completion. Just traverse the list.

A → B means B is A’s `next` item

X → Y means X can see Y

RCU-protected Linked List: Deletion of B

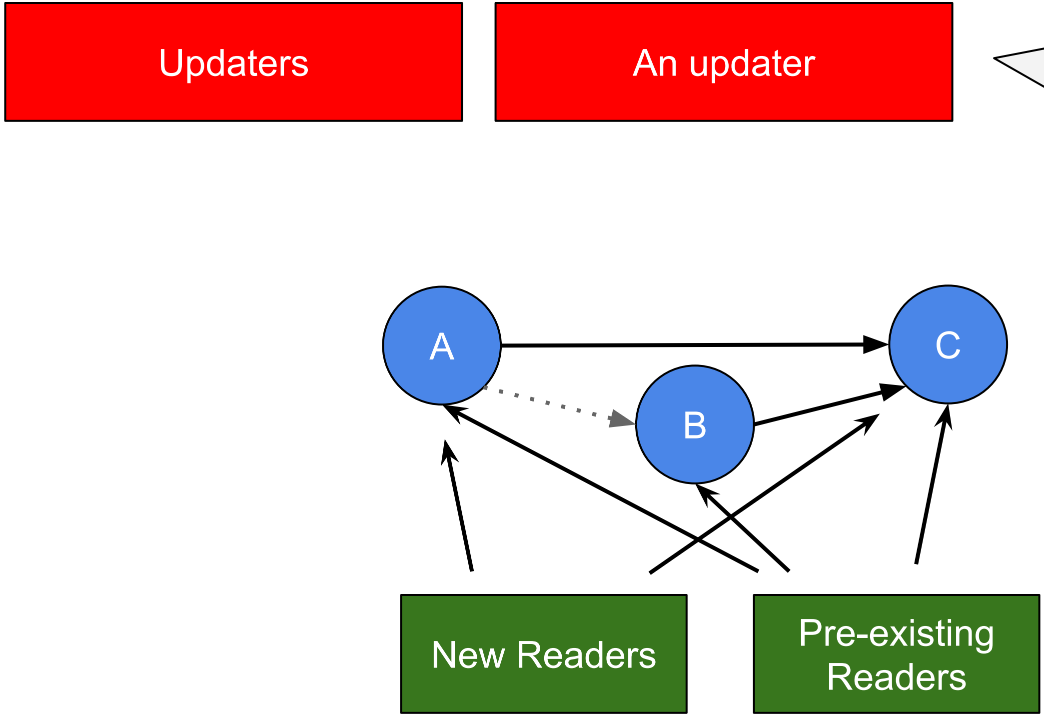



```
lock(update_lock);
a->next = c;
unlock(update_lock);
```

`lock()` is required to avoid the race between concurrent updates.
Use of the **global locking** becomes the scalability bottleneck.

A ⟶ B means B is A's `next` item
X ⟶ Y means X can see Y

RCU-protected Linked List: Deletion of B



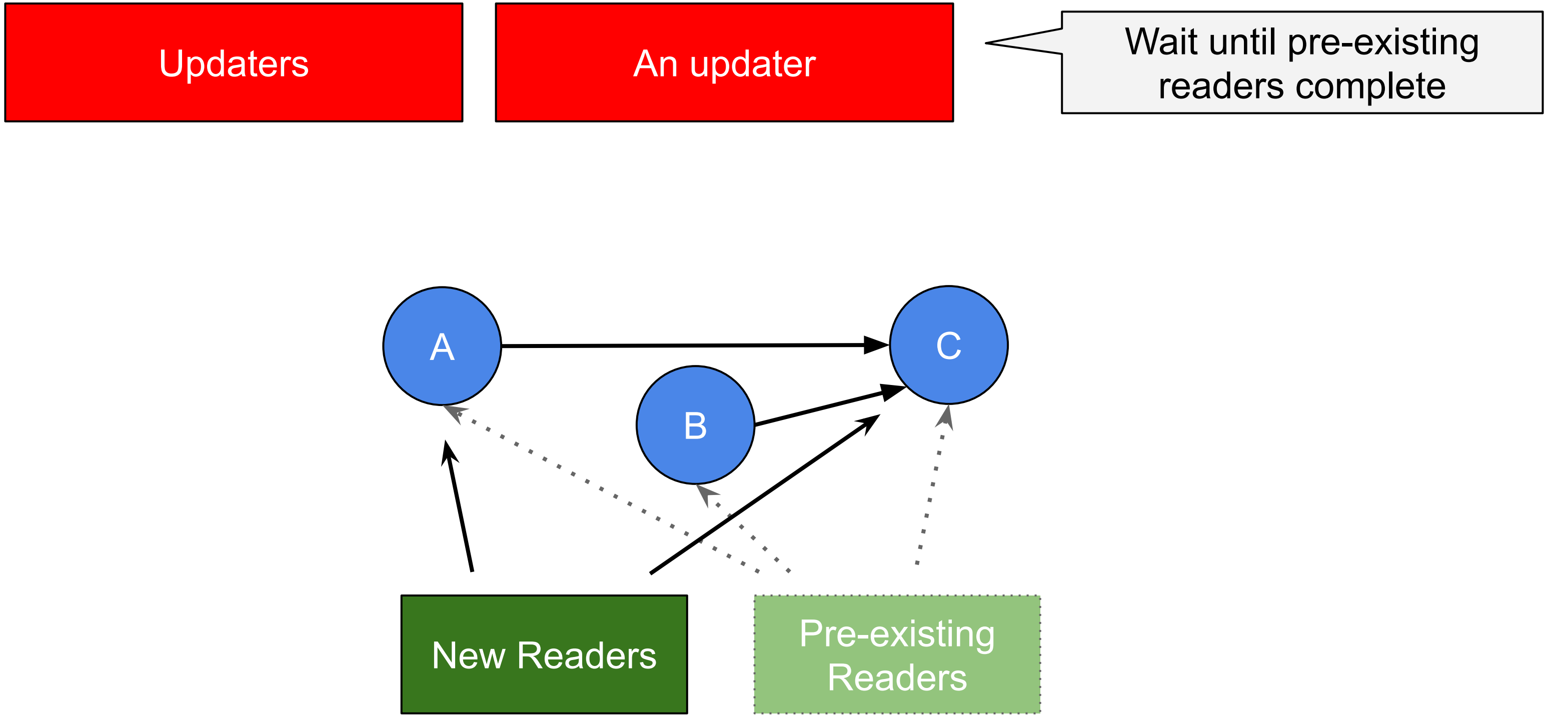

```
lock(update_lock);
a->next = c;
unlock(update_lock);
```

`lock()` is required to avoid the race between concurrent updates.
Use of the **global locking** becomes the scalability bottleneck.

Now there are pre-existing readers and new readers.

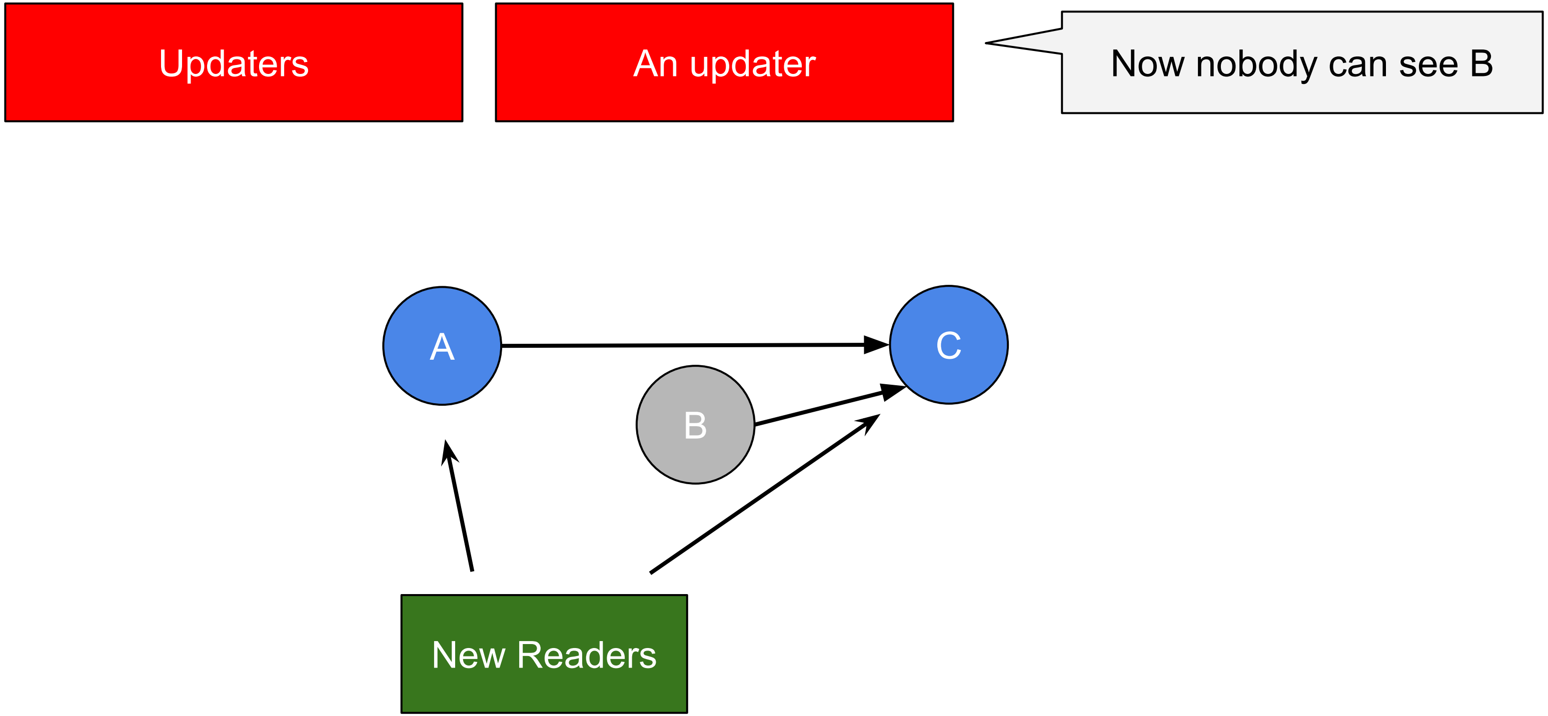
A ⟶ B means B is A's `next` item
X ⟶ Y means X can see Y

RCU-protected Linked List: Deletion of B



A → B means B is A's `next` item
X → Y means X can see Y

RCU-protected Linked List: Deletion of B



A → B means B is A's `next` item
X → Y means X can see Y

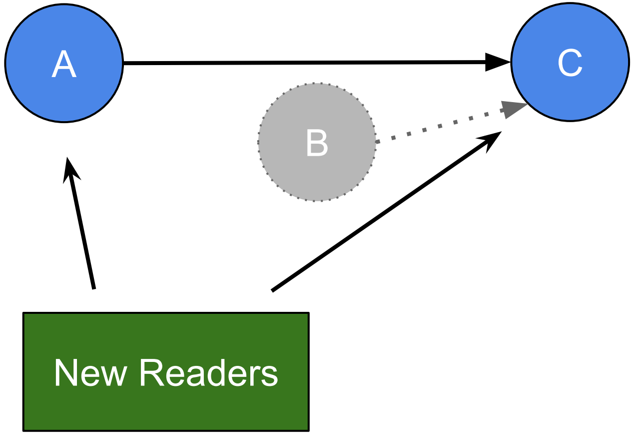
RCU-protected Linked List: Deletion of B

Updaters

An updater

safe to reuse B!
`free(B);`

Called QSBR (Quiescent State Based Reclaim)



A ⟶ B means B is A's `next` item

X ⟶ Y means X can see Y

Lack of RCU-centric update-side synchronization

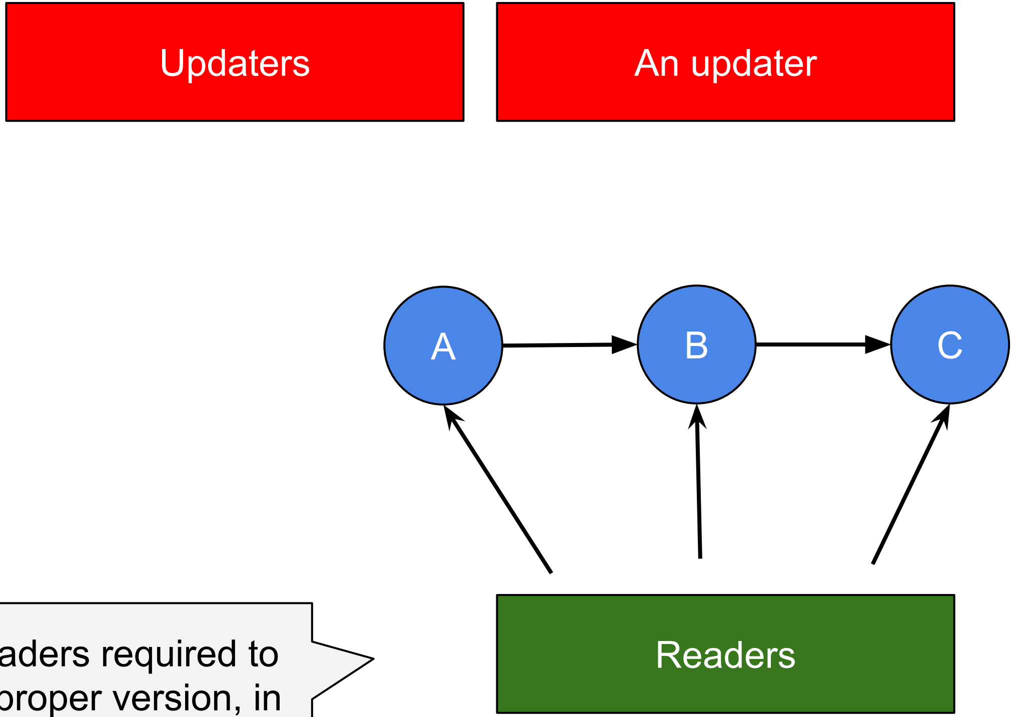
- Intended design
  - allow users selecting or implementing best synchronization mechanism for them
- However, many users use the global locking
  - Simple to apply, but imposes scalability problem
  - To mitigate this problem, several RCU extensions have proposed

Read-Log-Update (RLU)

- Published in SOSP'15[1]
- Adopts a software transactional memory (STM) like logging mechanism

[1] Matveev, Alexander, et al. "Read-log-update: a lightweight synchronization mechanism for concurrent programming." *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 2015.

RLU-protected Linked List: Reading Items



RLU Readers required to find out proper version, in addition to notifying its start and completion

A ⟶ B means B is A’s `next` item

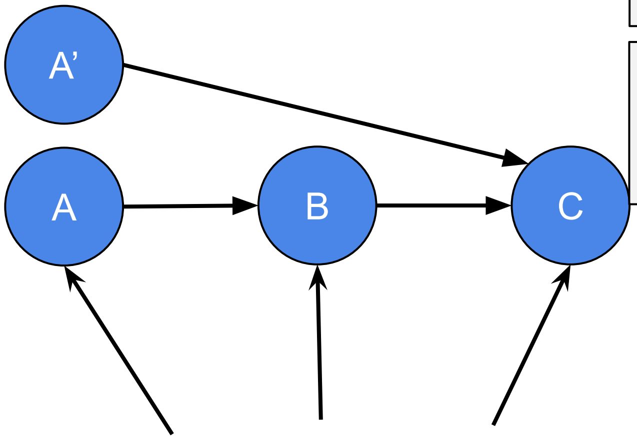
X ⟶ Y means X can see Y

RLU-protected Linked List: Deletion of B

Updaters

An updater

```
rlu_lock();
create new version A';
rlu_unlock();
```



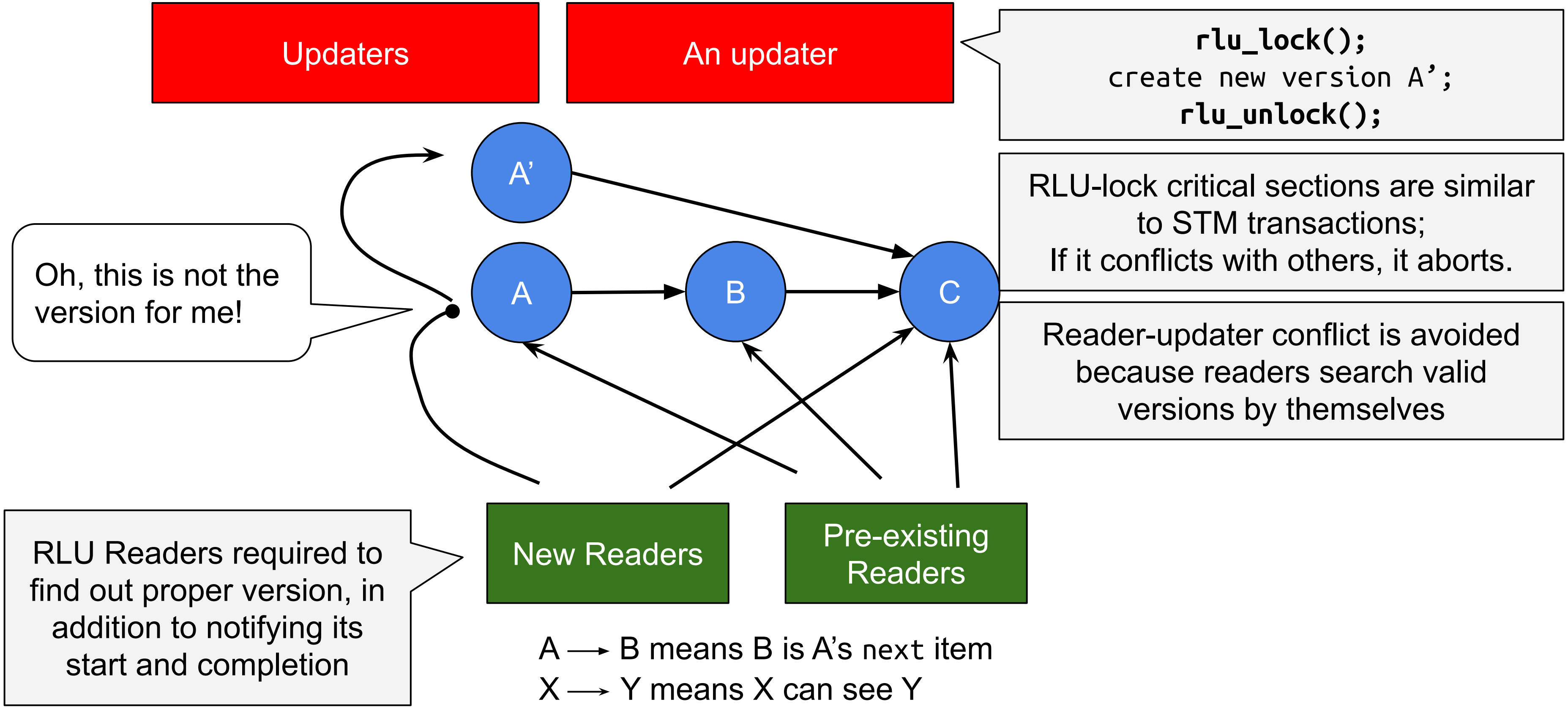
RLU-lock critical sections are similar to STM transactions;
If it conflicts with others, it aborts.

RLU Readers required to find out proper version, in addition to notifying its start and completion

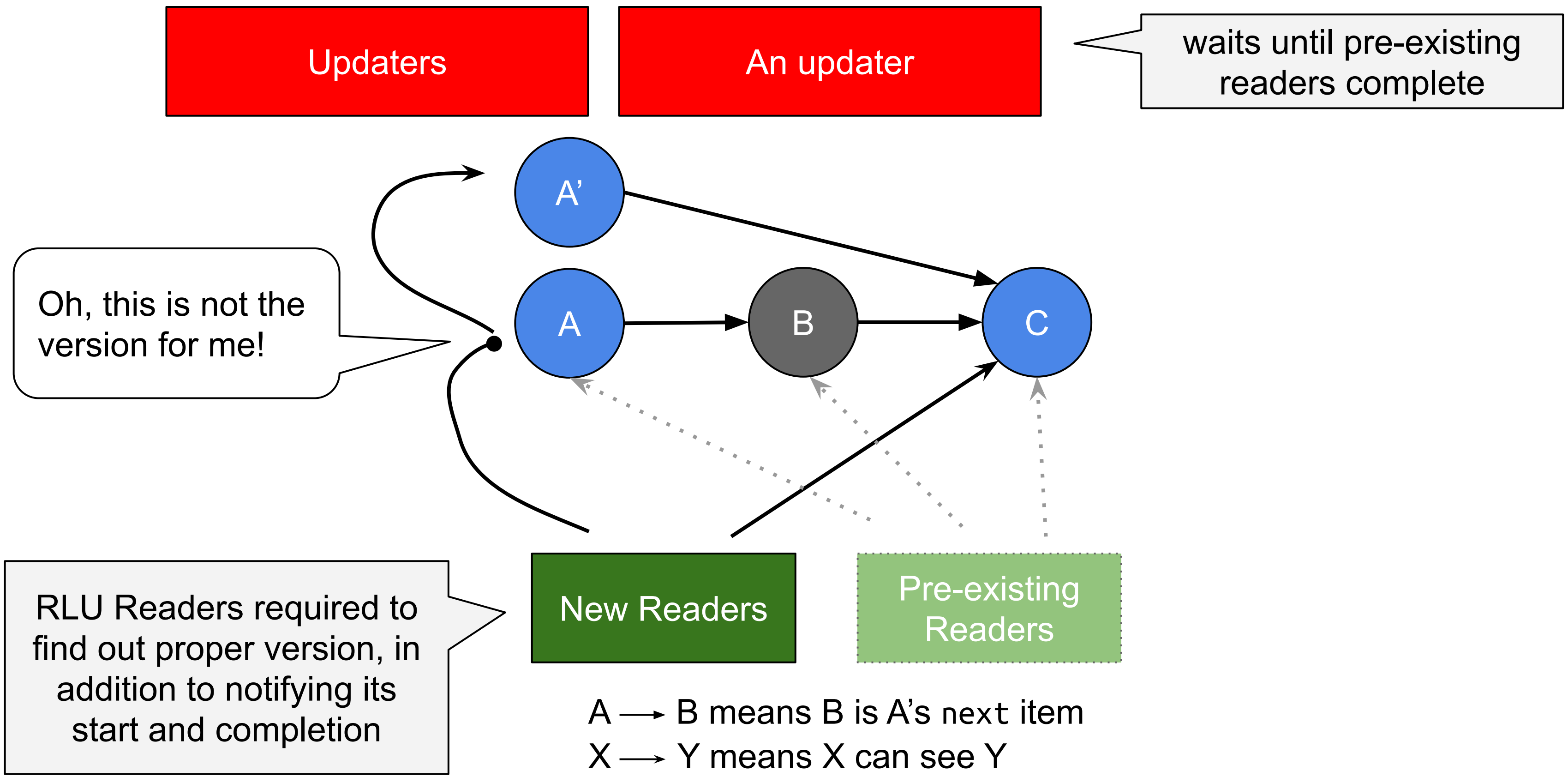
Readers

A → B means B is A's `next` item
X → Y means X can see Y

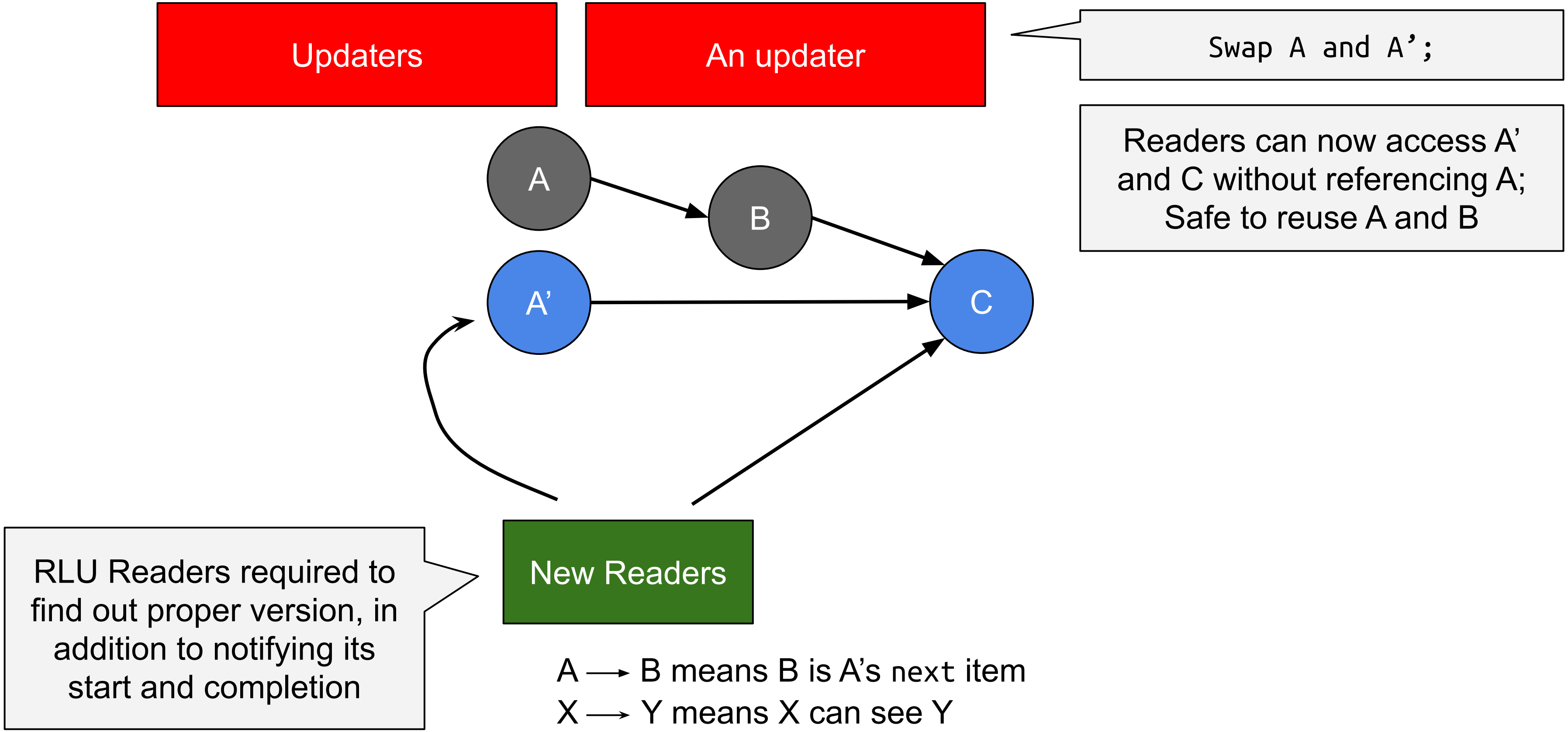
RLU-protected Linked List: Deletion of B



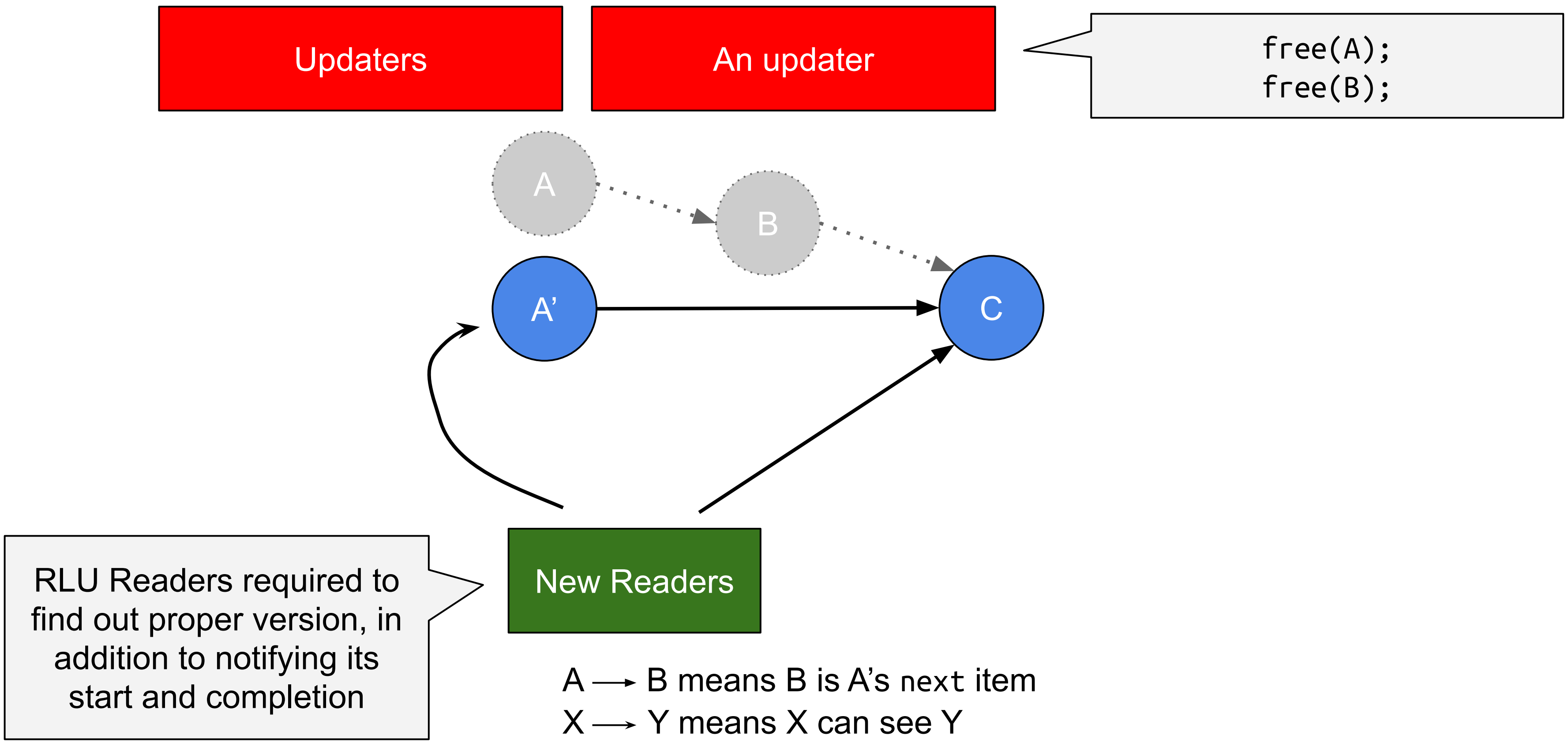
RLU-protected Linked List: Deletion of B



RLU-protected Linked List: Deletion of B



RLU-protected Linked List: Deletion of B

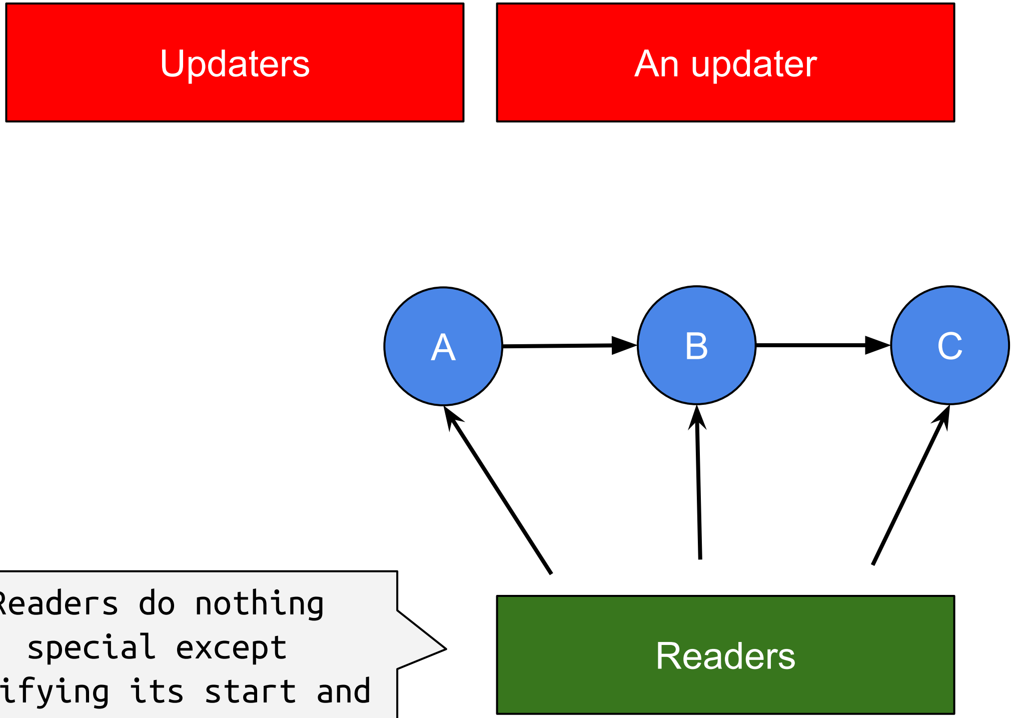


RCU-HTM

- Published in PACT'17[1]
- Encapsulates each update in an HTM transaction

[1] Siakavaras, Dimitrios, et al. "RCU-HTM: combining RCU with HTM to implement highly efficient concurrent binary search trees." *Parallel Architectures and Compilation Techniques (PACT), 2017 26th International Conference on*. IEEE, 2017.

RCU-HTM-protected Linked List: Reading Items

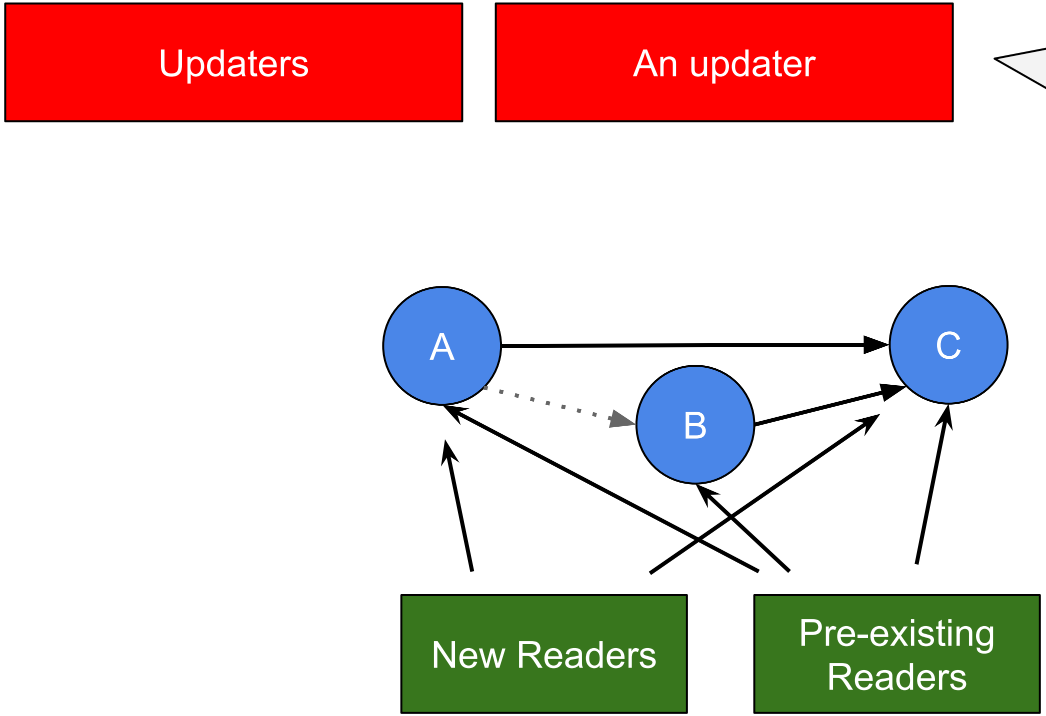


Readers do nothing special except notifying its start and completion. Just traverse the list.

A → B means B is A's next item

X → Y means X can see Y

RCU-HTM-protected Linked List: Deletion of B




```
begin_htm_trx();
a->next = c;
commit_htm_trx();
```

Encapsulate data updates within HTM transaction; HTM guarantees consistency and scalability

Else are same to QSBR; Wait until safe and dealloc

A → B means B is A's `next` item

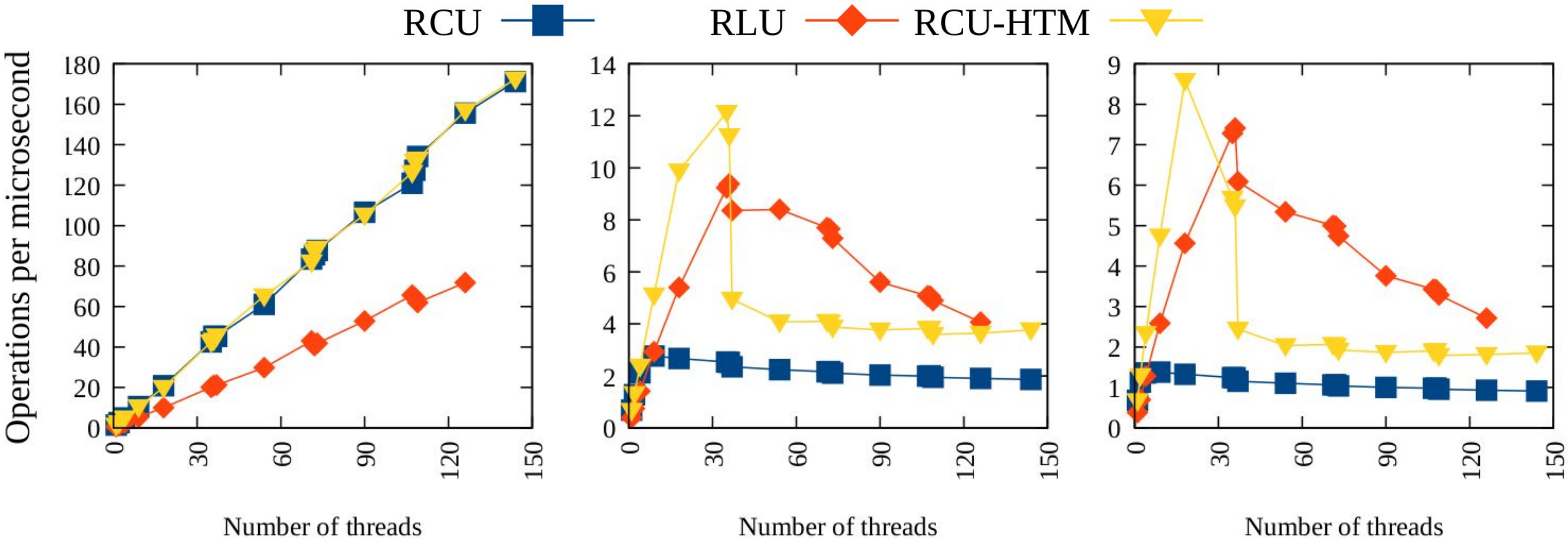
X → Y means X can see Y

Will Those Scale On NUMA Machines?

- Both RLU and RCU-HTM had not evaluated on huge NUMA machine
  - RLU was evaluated with single socket machine utilizing 16 threads
  - RCU-HTM evaluated with single socket machine utilizing 44 threads
- Server: 4 sockets, 18 cores, hyper-threaded (total 144 h/w threads)
  - Every following evaluation uses this server
- Workload: Random reads, inserts, and deletes to kernel space linked lists
  - Each of the linked lists are protected by RCU, RLU, and RCU-HTM, respectively
  - 256 initial items pre-loaded (sufficient to scale with 144 threads)
  - Measure operations per second with varying number of threads and update rate

Unexpected Poor Scalability Revealed

- RLU imposes significant **overhead to reads**
- With updates, RLU and RCU-HTM **degrade** as multiple NUMA nodes used



(a) Read-only (b) 20% updates (c) 40% updates

Root-causes and Implications of The Results

- RLU’s read overhead apparently comes from the valid version searching
  - Read-mostly performance-sensitive workloads would not use RLU instead of RCU!
- NUMA-oblivious designs of RLU and RCU-HTM degrade update scalability
- In case of RCU-HTM, amplification of HTM aborts on NUMA impacts
  - Long latency between NUMA makes transaction time long and thus easy to be aborted
  - The dominate readers conflict with HTM transactions of update threads and aborts them
- HTM benefit is clear, we need NUMA-aware HTM use for read-mostly works

	Read	Update on single NUMA node	Update on multiple NUMA nodes
RCU	Almost Ideal	Bad (Global locking)	Bad (Global locking)
RLU	Far from ideal (Version check overhead)	Good	Bad (NUMA oblivious)
RCU-HTM	Almost Ideal	Best (No software locking overhead)	Horrible (HTM aborts amplification)

Our Design Principles for New RCU Extension

We design new RCU extension called RCX with our principles

Our Design Principles for New RCU Extension

We design new RCU extension called RCX with our principles

1. Do fine-grained update-side synchronization

	Principle #1
RCU	X
RLU	O
RCU-HTM	O

Our Design Principles for New RCU Extension

We design new RCU extension called RCX with our principles

1. Do fine-grained update-side synchronization
2. Use pure RCU read mechanism for the ideal read performance and scalability

	Principle #1	Principle #2
RCU	X	O
RLU	O	X
RCU-HTM	O	O

Our Design Principles for New RCU Extension

We design new RCU extension called RCX with our principles

1. Do fine-grained update-side synchronization
2. Use pure RCU read mechanism for the ideal read performance and scalability
3. Use HTM; Only HTM provides H/W-oriented high performance

	Principle #1	Principle #2	Principle #3
RCU	X	O	X
RLU	O	X	X
RCU-HTM	O	O	O

Our Design Principles for New RCU Extension

We design new RCU extension called RCX with our principles

1. Do fine-grained update-side synchronization
2. Use pure RCU read mechanism for the ideal read performance and scalability
3. Use HTM; Only HTM provides H/W-oriented high performance
4. Access only NUMA-local data objects within HTM transaction
    a. Otherwise, abort rates exponentially increase

	Principle #1	Principle #2	Principle #3	Principle #4
RCU	X	O	X	N/A
RLU	O	X	X	N/A
RCU-HTM	O	O	O	X

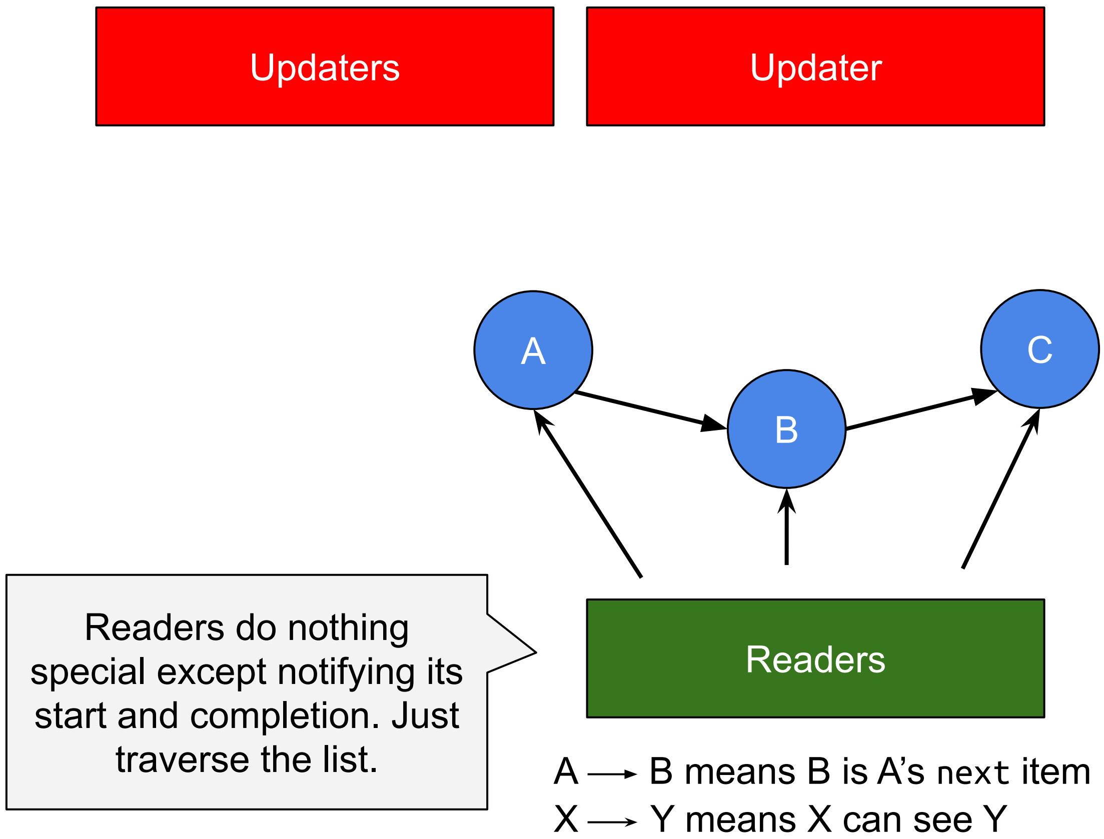
Our Design Principles for New RCU Extension

We design new RCU extension called RCX with our principles

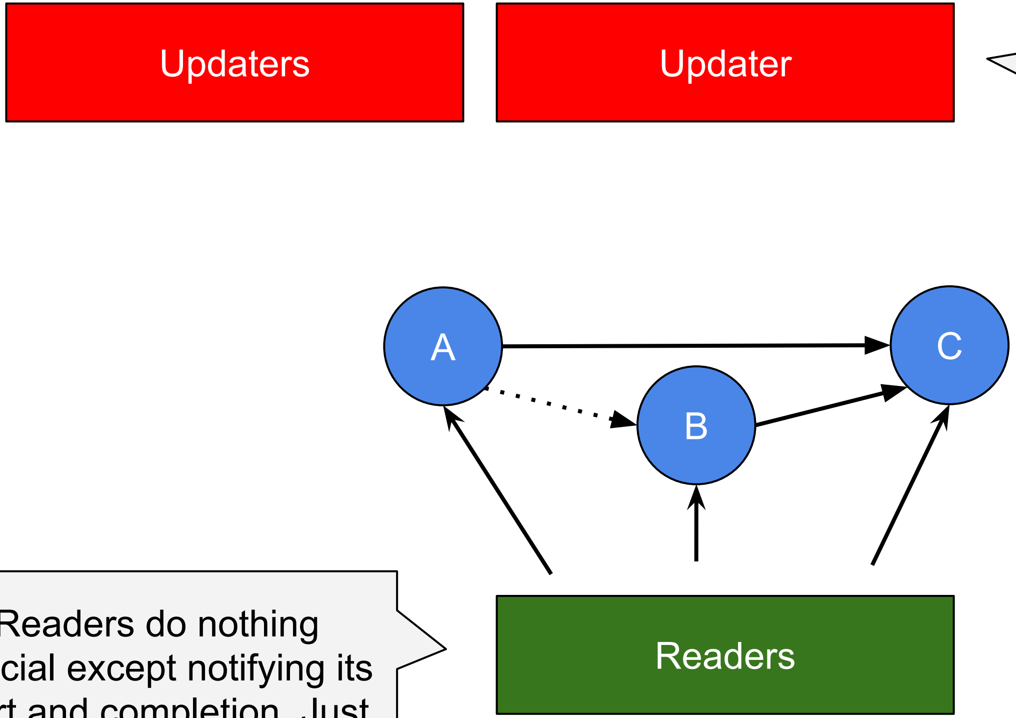
1. Do fine-grained update-side synchronization
2. Use pure RCU read mechanism for the ideal read performance and scalability
3. Use HTM; Only HTM provides H/W-oriented high performance
4. Access only NUMA-local data objects within HTM transaction
   a. Otherwise, abort rates exponentially increase
5. Isolate the HTM working set from the dominant readers
   a. Otherwise, the readers abort HTM transactions

	Principle #1	Principle #2	Principle #3	Principle #4	Principle #5
RCU	X	O	X	N/A	N/A
RLU	O	X	X	N/A	N/A
RCU-HTM	O	O	O	X	X

RCX Interface



RCX Interface




```
rcx_lock(A,B,C);
a->next = c;
rcx_unlock(A,B,C);
```

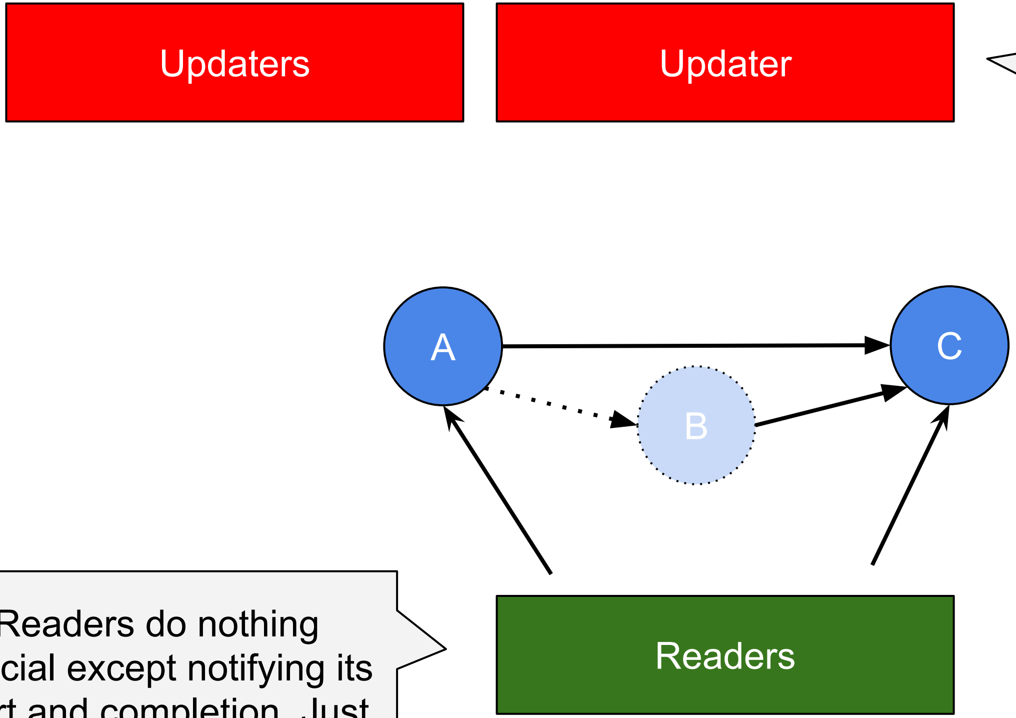
In RCX, update critical sections should specify items to update

Readers do nothing special except notifying its start and completion. Just traverse the list.

A ⟶ B means B is A's `next` item

X ⟶ Y means X can see Y

RCX Interface




```
rcx_lock(A,B,C);
a->next = c;
rcx_unlock(A,B,C);
```

In RCX, update critical sections should specify items to update

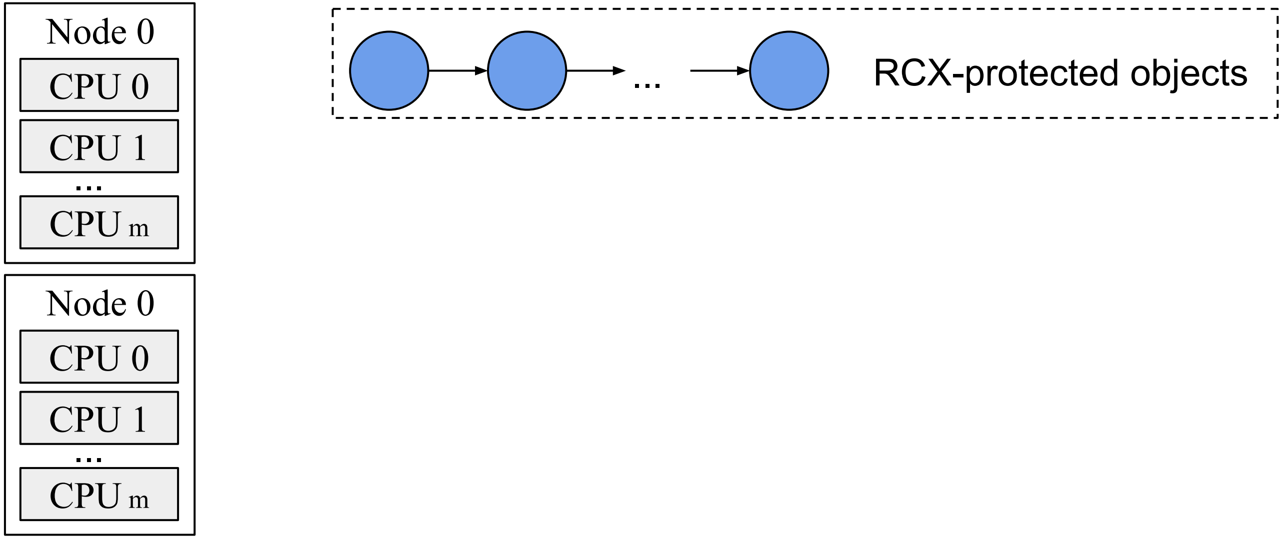
Else are same to QSBR; Wait until safe and dealloc

Readers do nothing special except notifying its start and completion. Just traverse the list.

A → B means B is A's `next` item

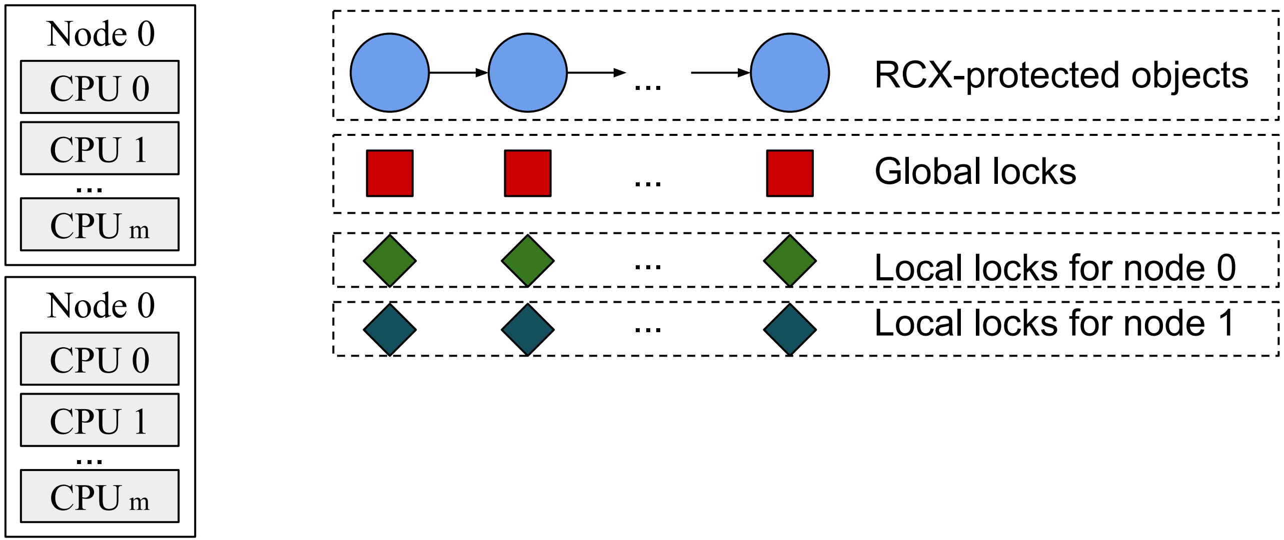
X → Y means X can see Y

`rcx_lock()` in Detail



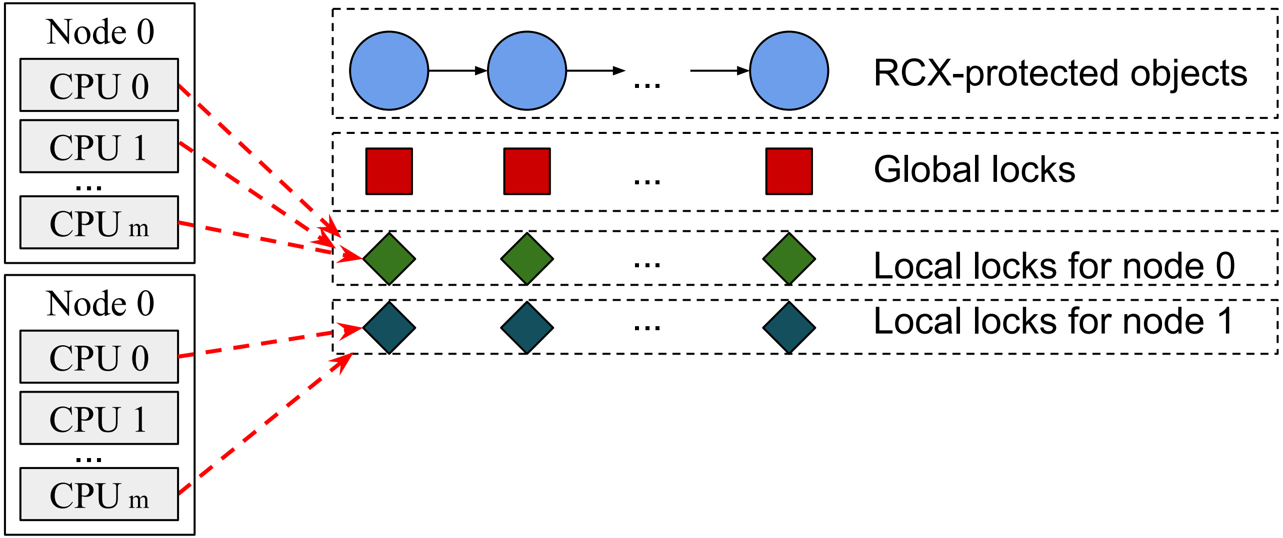
`rcx_lock()` in Detail

- Embed node-local locks and a global lock to each object



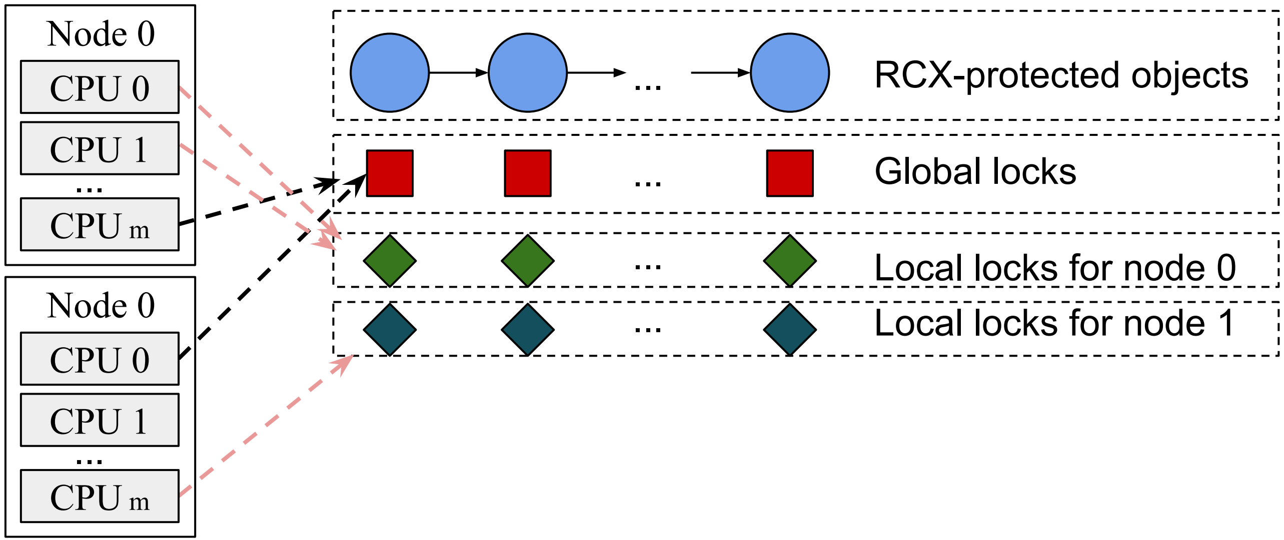
`rcx_lock()` in Detail

- Embed node-local locks and a global lock to each object
- Updaters first acquire the per-node **local lock using HTM**



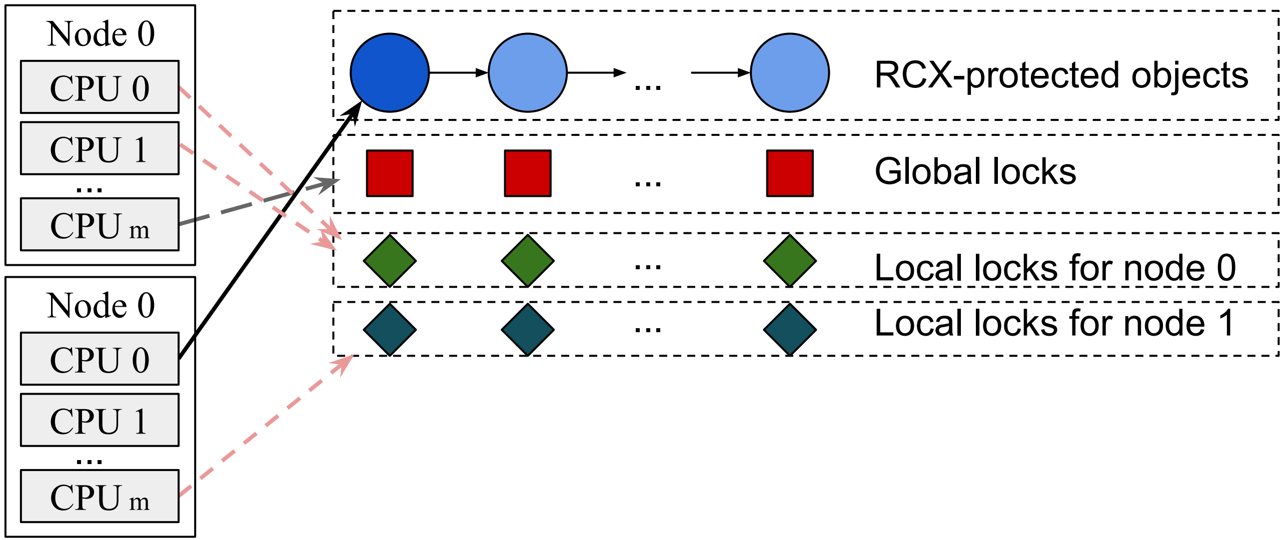
`rcx_lock()` in Detail

- Embed node-local locks and a global lock to each object
- Updaters first acquire the per-node **local lock using HTM**
- Than, commit the transaction and acquire the **global lock using spinlock**



`rcx_lock()` in Detail

- Embed node-local locks and a global lock to each object
- Updaters first acquire the per-node **local lock using HTM**
- Than, commit the transaction and acquire the **global lock using spinlock**
- Updaters who acquired both locks can update the items



RCX and The Principles

RCX and The Principles

- Do fine-grained update-side synchronization
  - Compete with threads accessing same objects only

	Principle #1
RCU	X
RLU	O
RCU-HTM	O
RCX	O

RCX and The Principles

- Do fine-grained update-side synchronization
  - Compete with threads accessing same objects only
- Use pure RCU read mechanism

	Principle #1	Principle #2
RCU	X	O
RLU	O	X
RCU-HTM	O	O
RCX	O	O

RCX and The Principles

- Do fine-grained update-side synchronization
  - Compete with threads accessing same objects only
- Use pure RCU read mechanism
- Use HTM

	Principle #1	Principle #2	Principle #3
RCU	X	O	X
RLU	O	X	X
RCU-HTM	O	O	O
RCX	O	O	O

RCX and The Principles

- Do fine-grained update-side synchronization
  - Compete with threads accessing same objects only
- Use pure RCU read mechanism
- Use HTM
- Access only NUMA-local data objects within HTM transaction

	Principle #1	Principle #2	Principle #3	Principle #4
RCU	X	O	X	N/A
RLU	O	X	X	N/A
RCU-HTM	O	O	O	X
RCX	O	O	O	O

RCX and The Principles

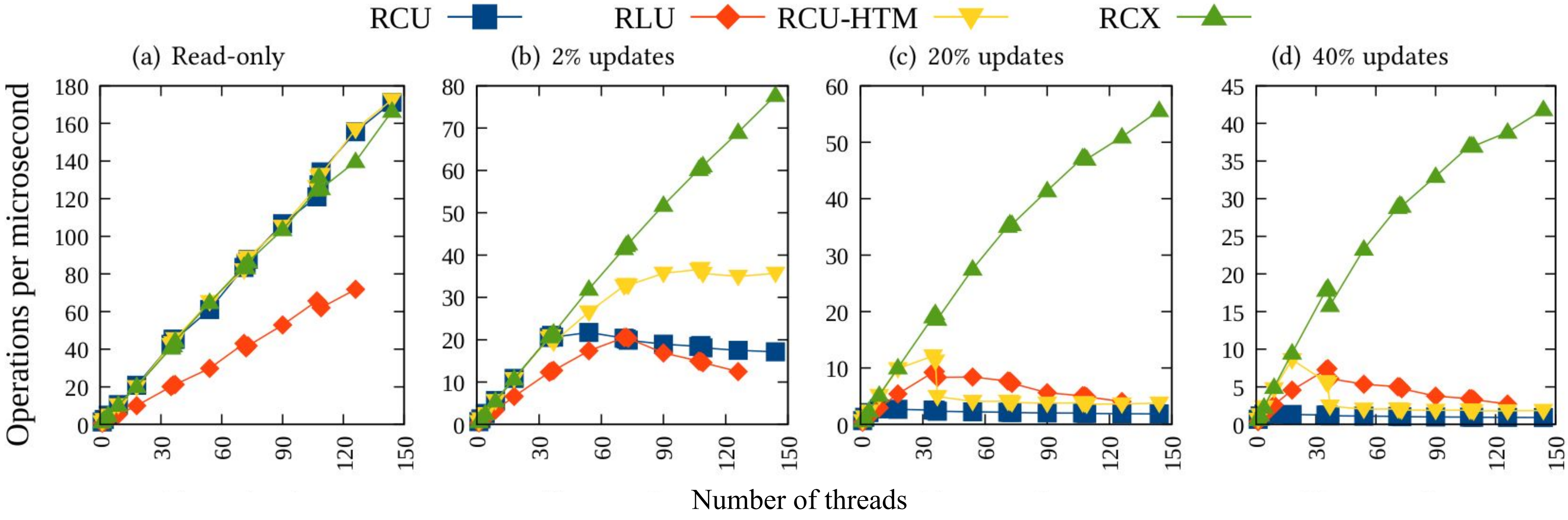
- Do fine-grained update-side synchronization
  - Compete with threads accessing same objects only
- Use pure RCU read mechanism
- Use HTM
- Access only NUMA-local data objects within HTM transaction
- Isolate the working set of HTM from the dominant Readers
  - HTM in RCX touches local locks only, which is invisible to readers

	Principle #1	Principle #2	Principle #3	Principle #4	Principle #5
RCU	X	O	X	N/A	N/A
RLU	O	X	X	N/A	N/A
RCU-HTM	O	O	O	X	X
RCX	O	O	O	O	O

Evaluations

RCU Variants-Protected Linked Lists

- RCX Performs best, for both read only and updates mixed workload
- Similar results with hash tables



Macro Benchmarks

- We further applied RCX to systems having scalability problems
  - Virtual memory management system of Linux
  - In-memory DBMS

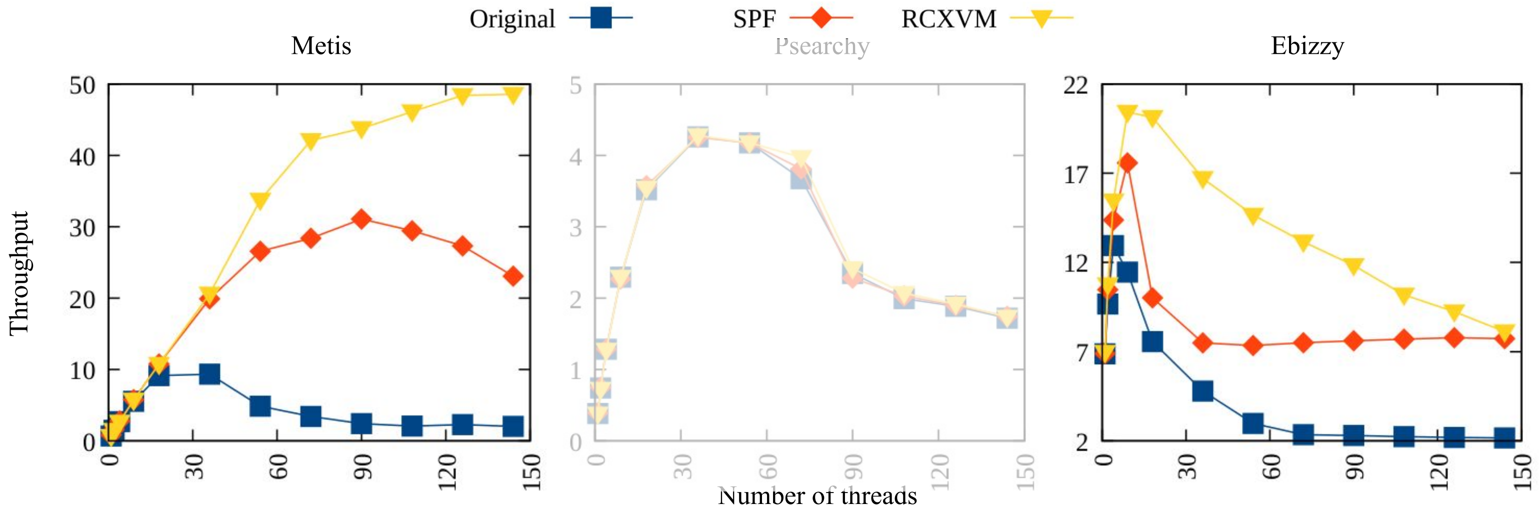
RCU-protected VMA-tree

- Linux protects each VMA-tree with a **global** reader-writer lock (`mmap_sem`)
- Two similar RCU approaches proposed: RCUVM[1] and SPF[2]
- However, VMA-tree update intensive workloads receive no benefit
- We further apply RCX on top of SPF and call it RCXVM

[1] Clements, Austin T., M. Frans Kaashoek, and Nickolai Zeldovich.
"Scalable address spaces using RCU balanced trees." in *ACM SIGPLAN Notices* 47.4 (2012): 199-210.
[2] H USSEIN , N. "Another attempt at speculative page-fault handling." https://lwn.net/Articles/730531/, 2017.

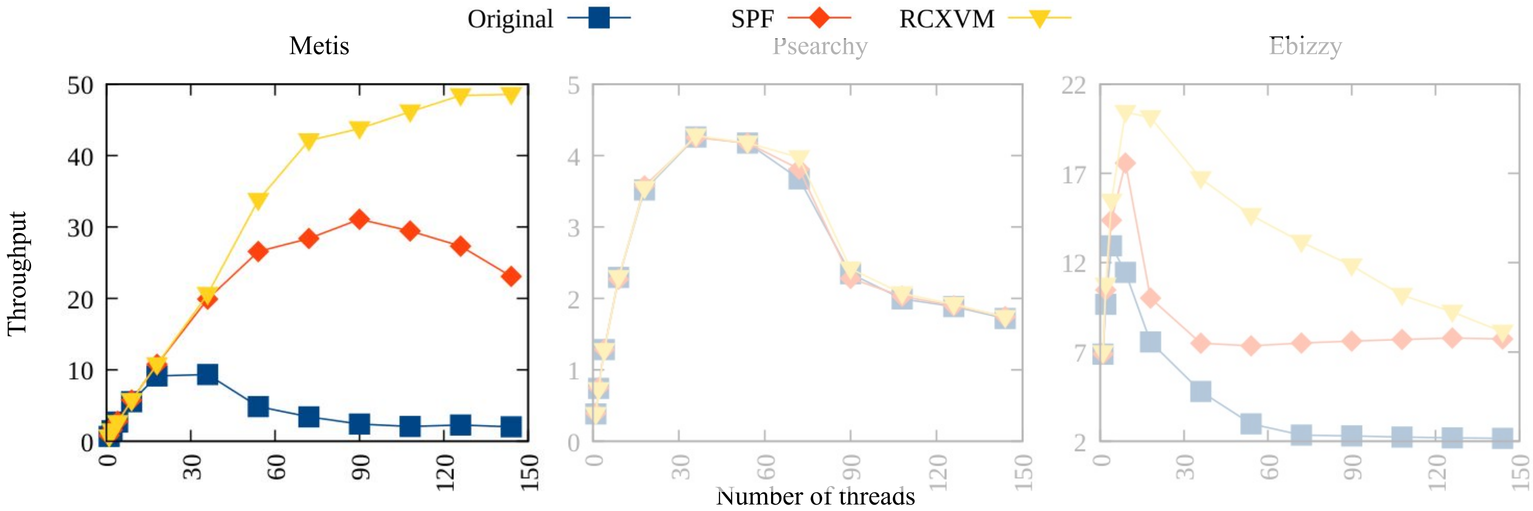
Virtual Memory Scalability Evaluation Result

- RCXVM further improves Metis and Ebizzy



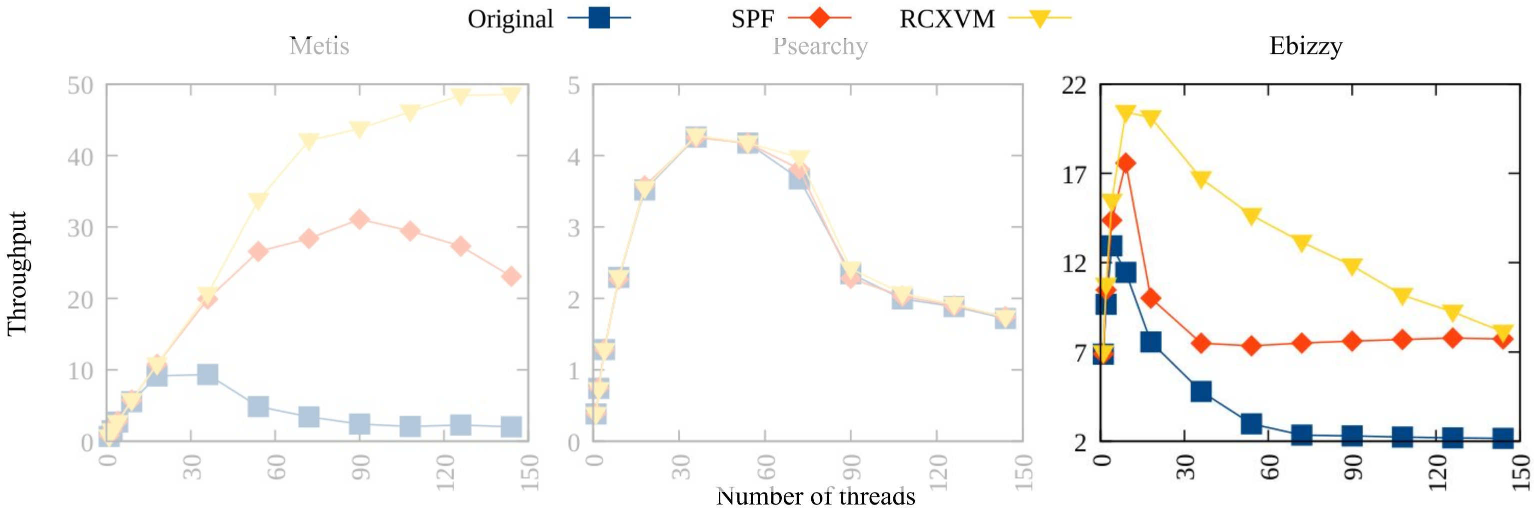
Virtual Memory Scalability Evaluation Result

- RCXVM further improves Metis and Ebizzy
  - Metis: Up to 24.03x of Original, 2.10x of SPF (144 threads)



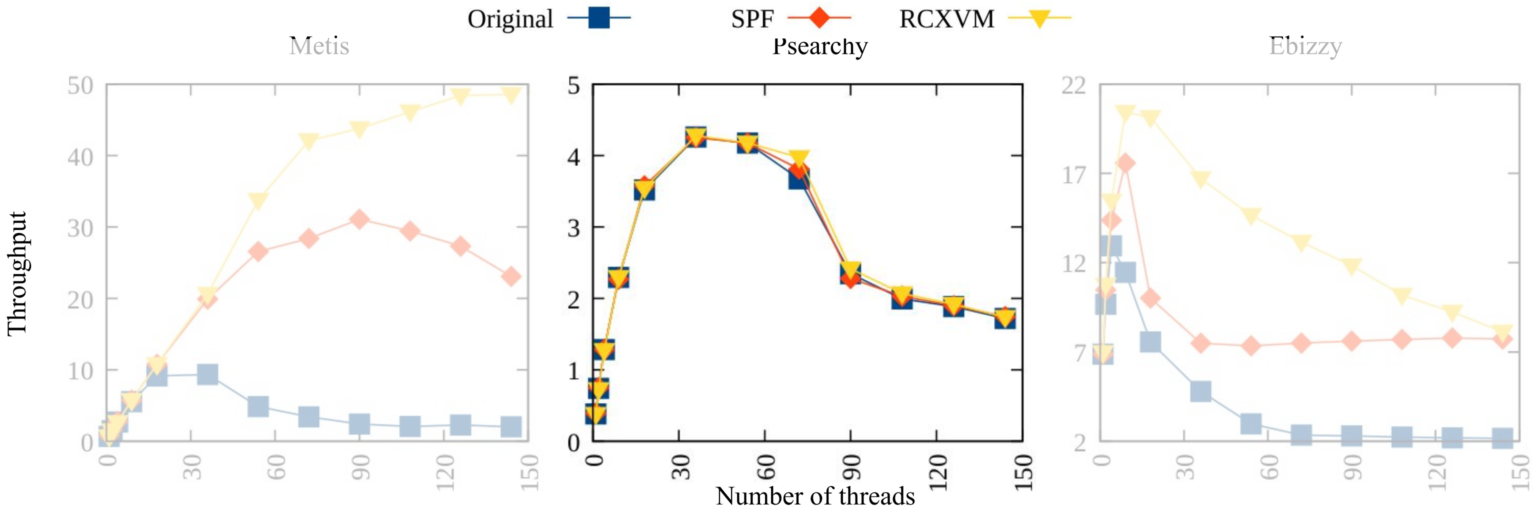
Virtual Memory Scalability Evaluation Result

- RCXVM further improves Metis and Ebizzy
  - Metis: Up to 24.03x of Original, 2.10x of SPF (144 threads)
  - Ebizzy: Up to 5.60x of Original (72 threads), 2.23x of SPF (36 threads)



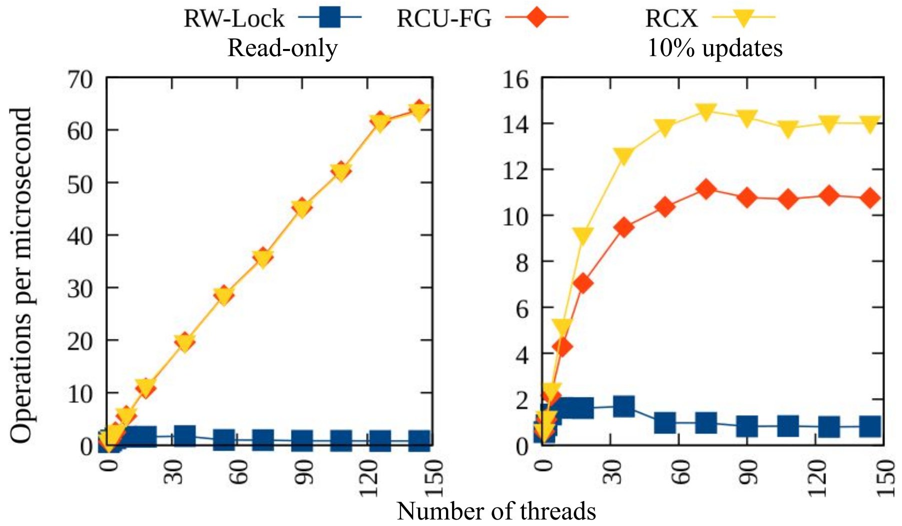
Virtual Memory Scalability Evaluation Result

- RCXVM further improves Metis and Ebizzy
  - Metis: Up to 24.03x of Original, 2.10x of SPF (144 threads)
  - Ebizzy: Up to 5.60x of Original (72 threads), 2.23x of SPF (36 threads)
- Psearchy and Ebizzy with many threads show no benefit
  - The bottleneck (tlb flushes) is out of RCXVM coverage



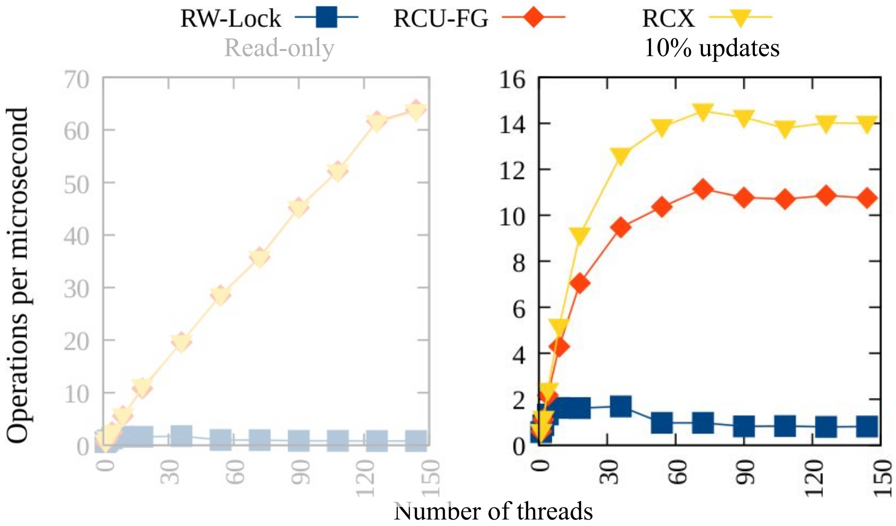
In-memory DBMS Scalability

- Kyoto CacheDB uses global reader-writer lock; We implement two variants substituting it with fine-grained RCU and RCX, respectively
- With 20 million records evaluation, RCX shows improvements
  - Up to 17.28x of Original and 1.3x of fine-grained RCU with 10% updates



In-memory DBMS Scalability

- Kyoto CacheDB uses global reader-writer lock; We implement two variants substituting it with fine-grained RCU and RCX, respectively
- With 20 million records evaluation, RCX shows improvements
  - Up to 17.28x of Original and 1.3x of fine-grained RCU with 10% updates



Conclusion

- RCX achieves best update while preserving the almost ideal read in terms of performance and scalability, owing to its NUMA-aware use of HTM
- Many details and additional things in the paper
  - Detailed investigations of state-of-the-arts including an HMCS lock and RCX variants
  - Optimization of RCX for memory efficiency and HTM implementation details
- The source code is available: https://github.com/rcx-sync

	Read	Single node update	Multiple NUMA node update
RCU	Almost Ideal	Bad (Global locking)	Bad (Global locking)
RLU	Far from ideal (Version check overhead)	Good	Bad (NUMA oblivious)
RCU-HTM	Almost Ideal	Best (No software locking overhead)	Horrible (HTM aborts amplification)
RCX	Almost Ideal	Best	Best

Thank You