경량 로그 기반 지연 업데이트 기법을 활용한 리눅스 커널 확장성 향상

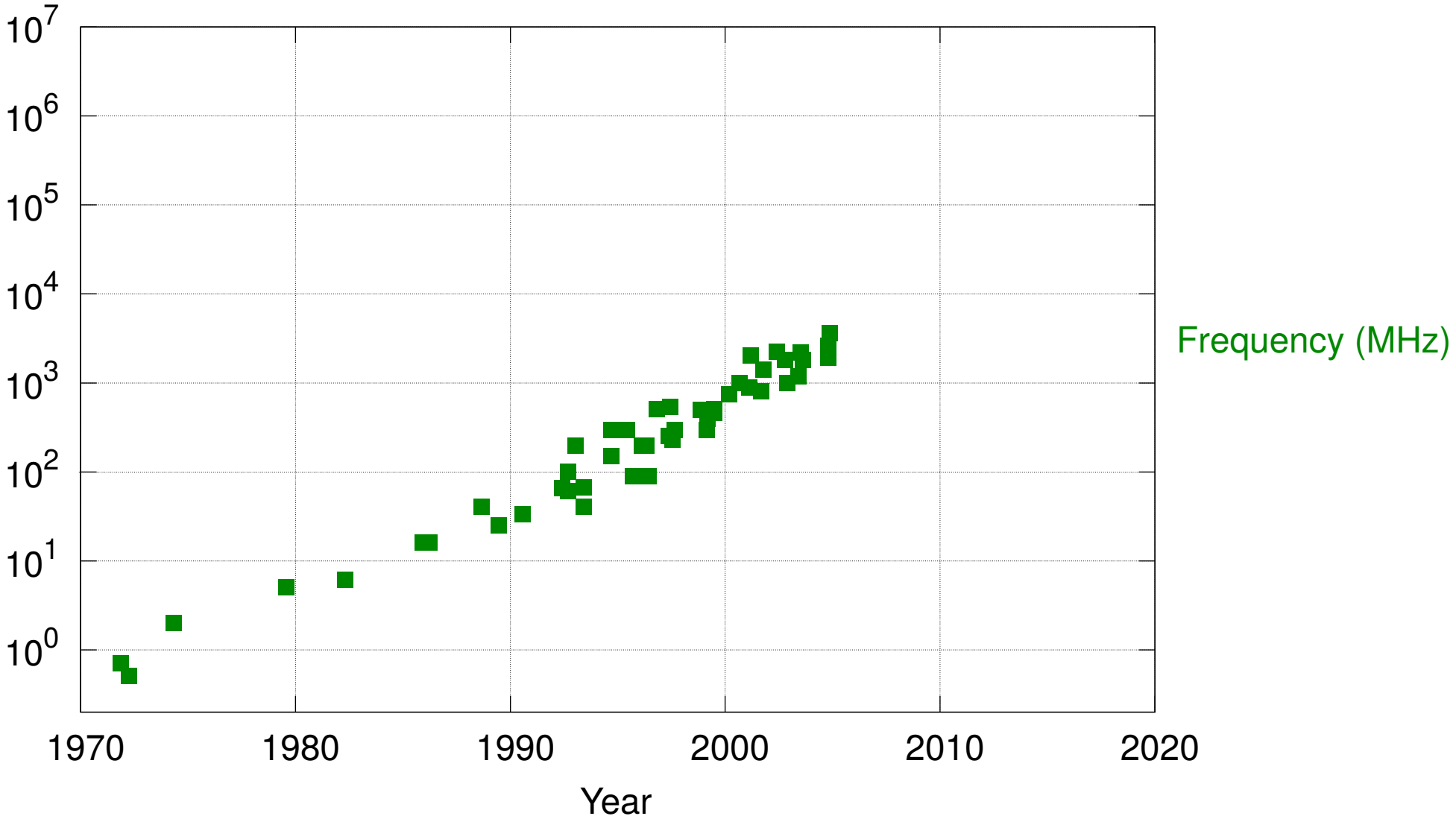
A Lightweight Log-based Deferred Update for Linux Kernel Scalability

Joohyun Kyong

School of Computer Science
Kookmin University
Thesis advisor: Sung-soo Lim

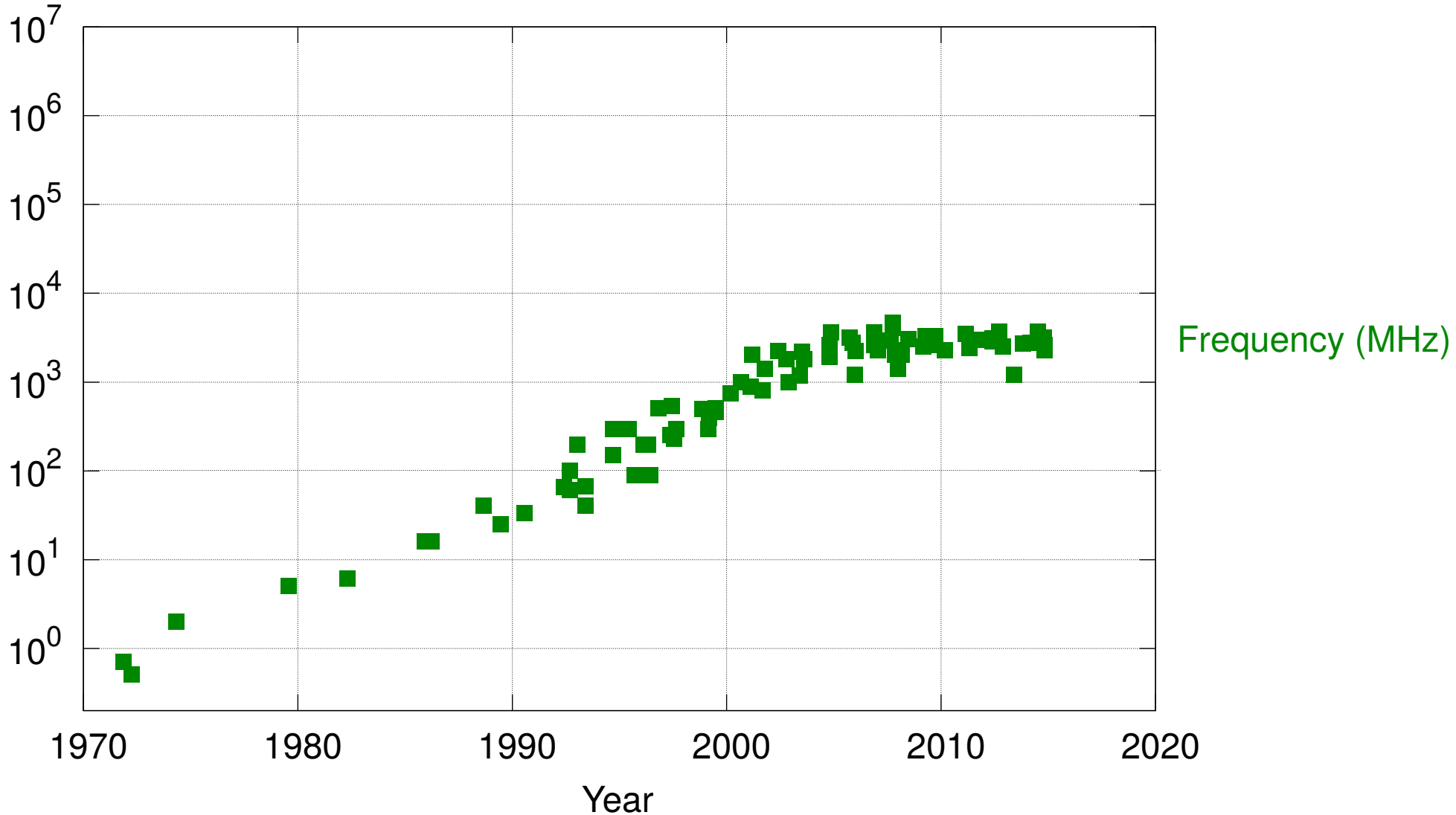
November 21, 2016

40 Years of Microprocessor Trend Data



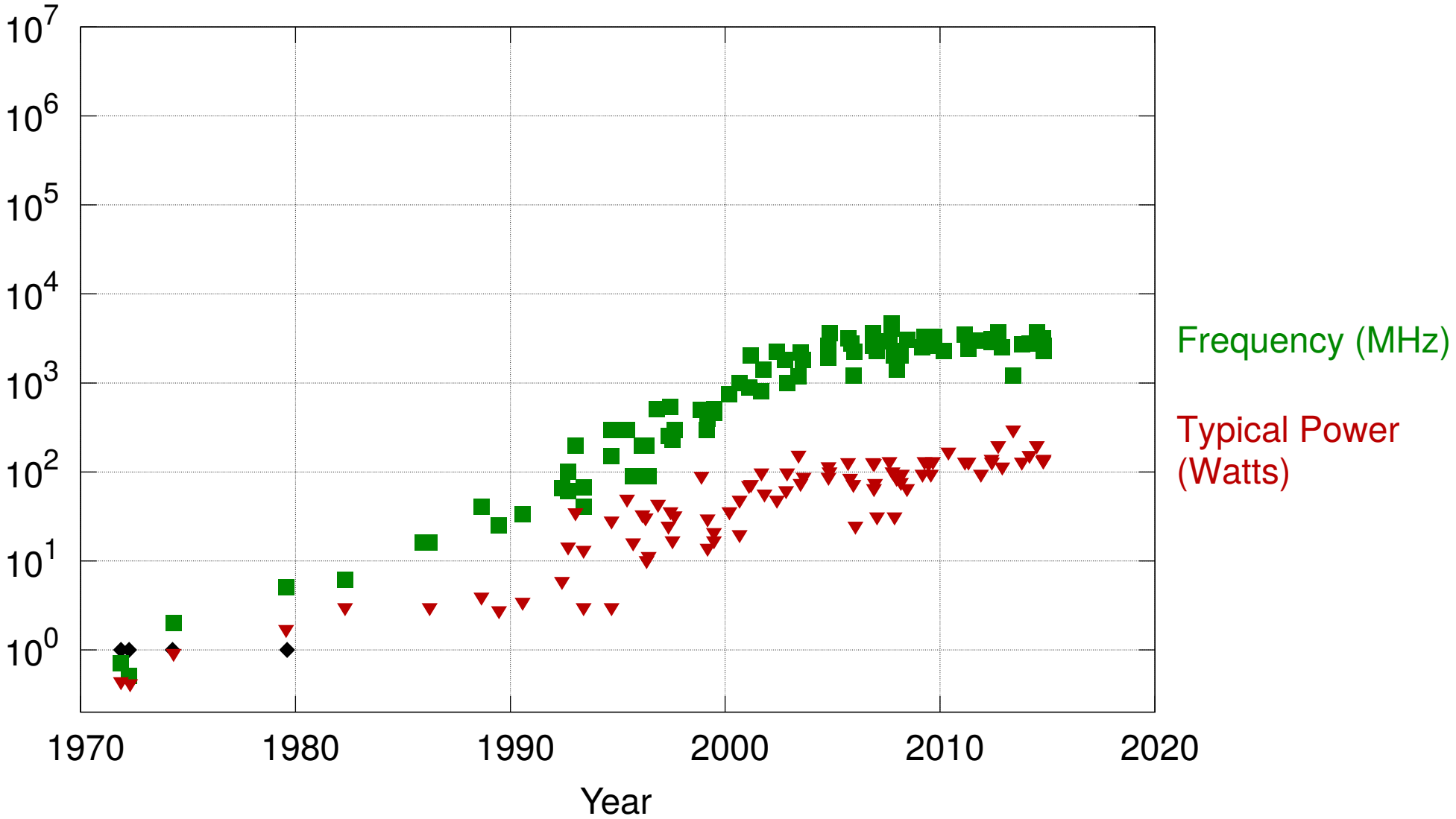
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

40 Years of Microprocessor Trend Data



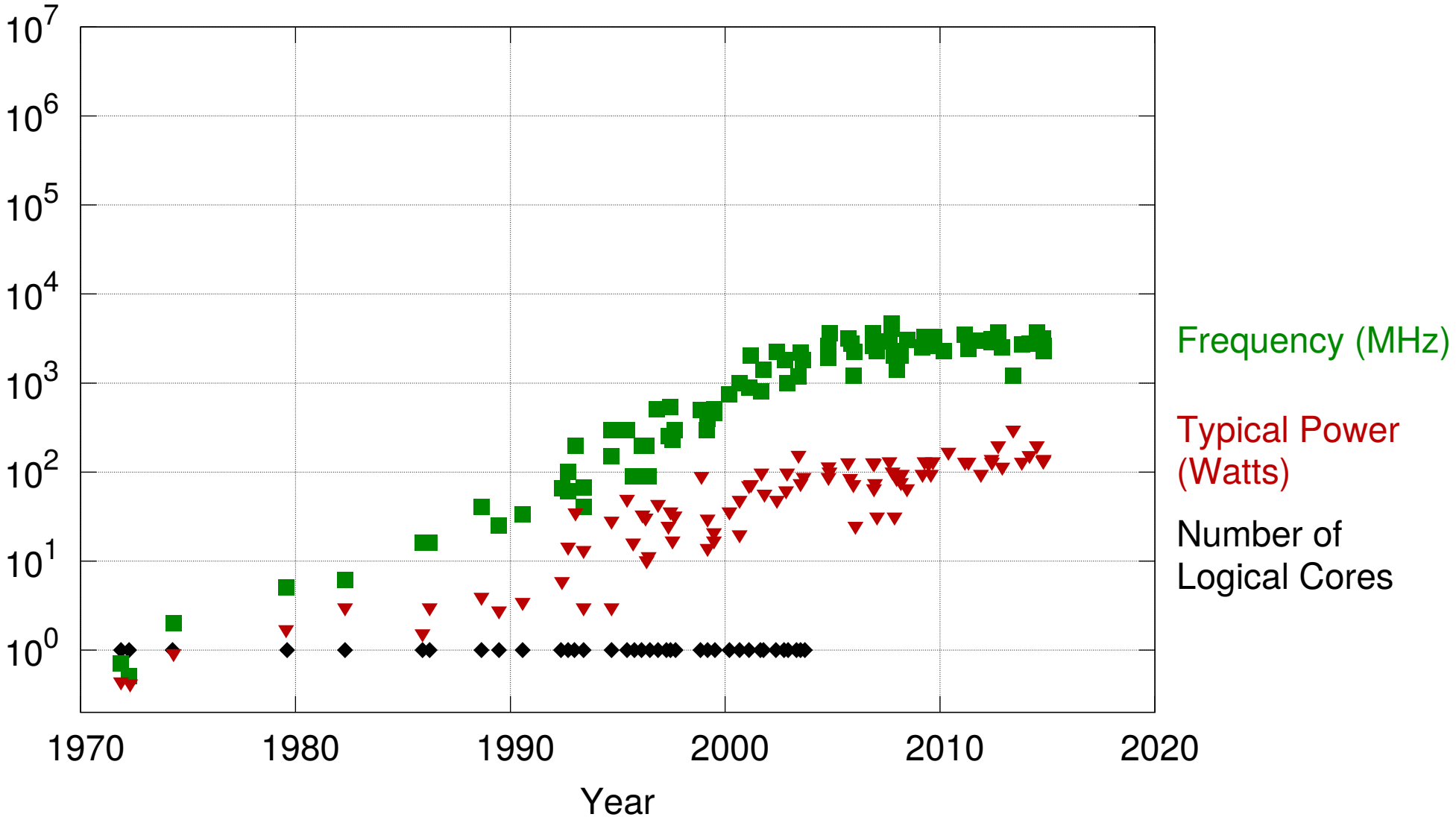
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

40 Years of Microprocessor Trend Data



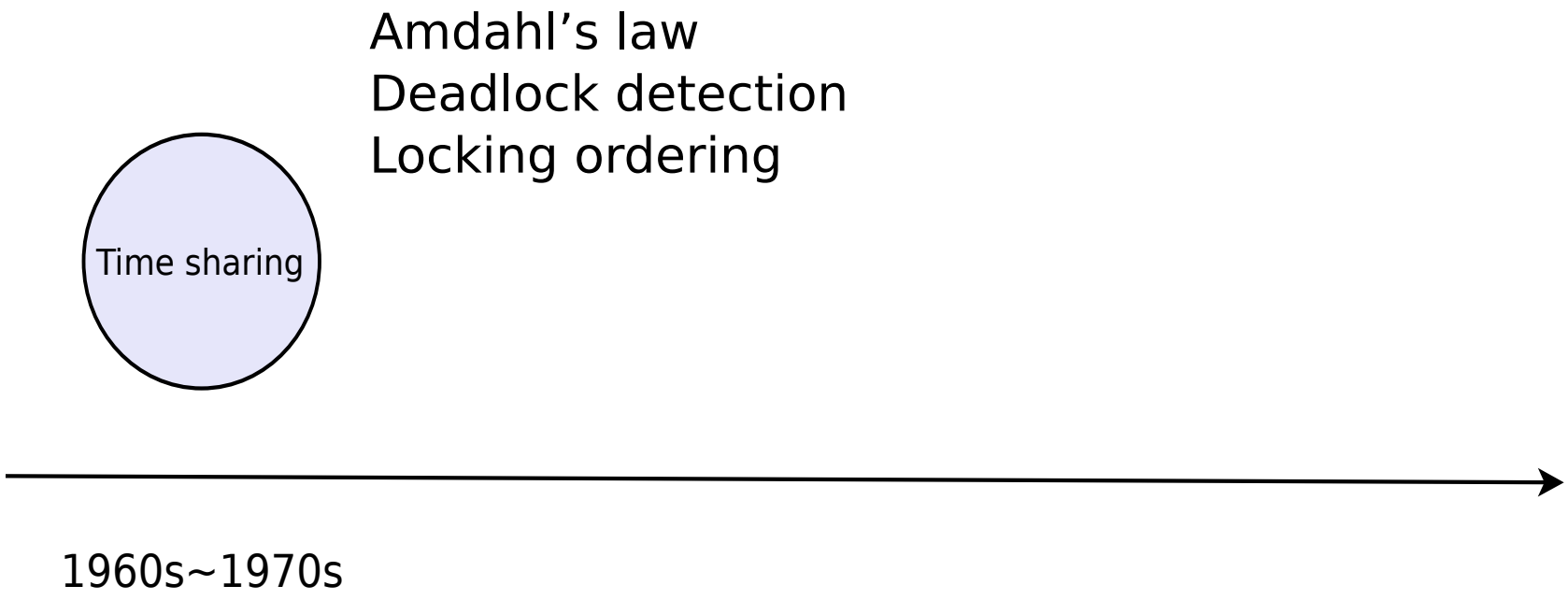
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

40 Years of Microprocessor Trend Data



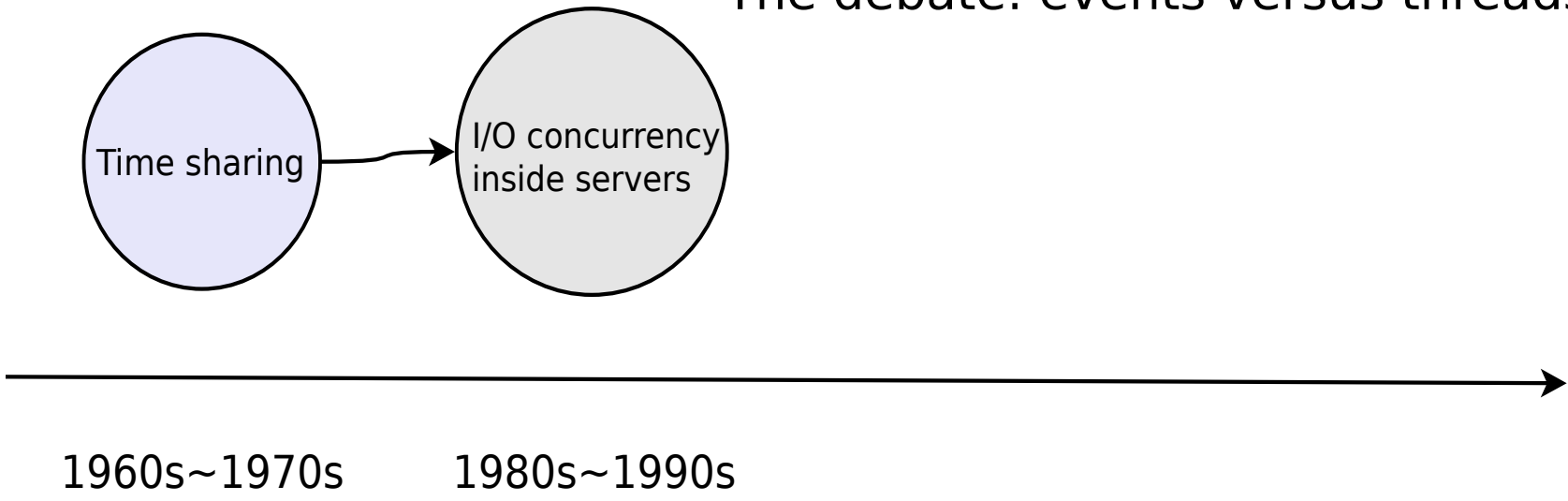
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

OS Kernel Scalability History



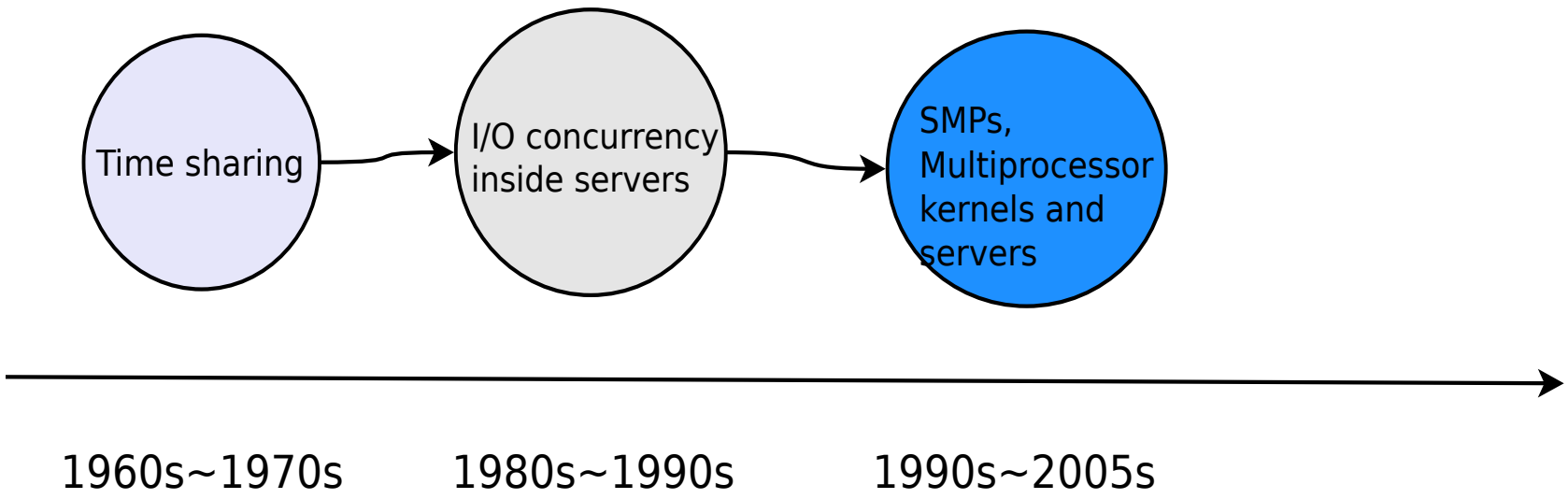
OS Kernel Scalability History

Threads
Locks
Condition variables
programming languages
The debate: events versus threads

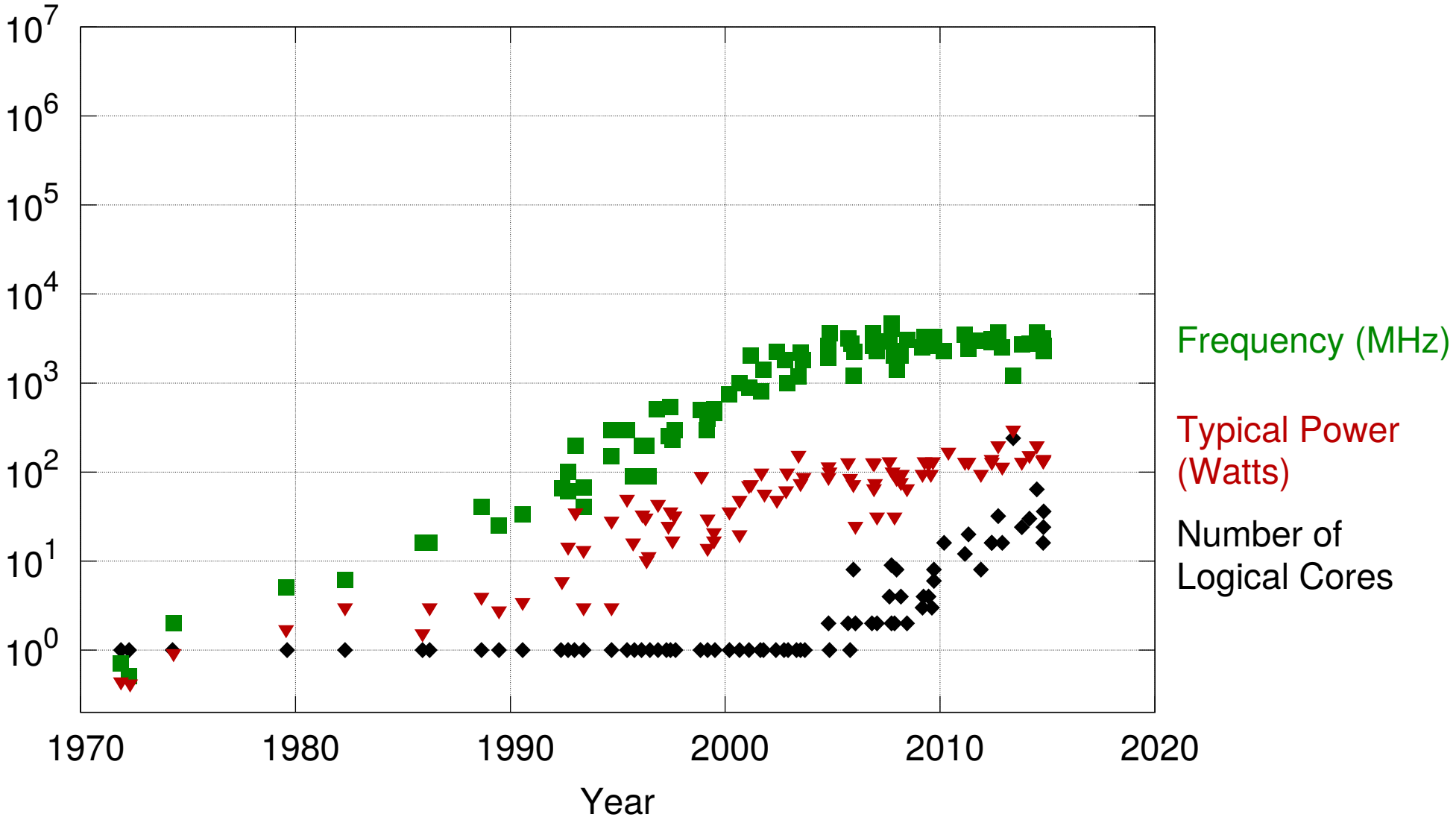


OS Kernel Scalability History

Scalable locks
Efficient user-level threading
NUMA memory management
Read-copy update (RCU)

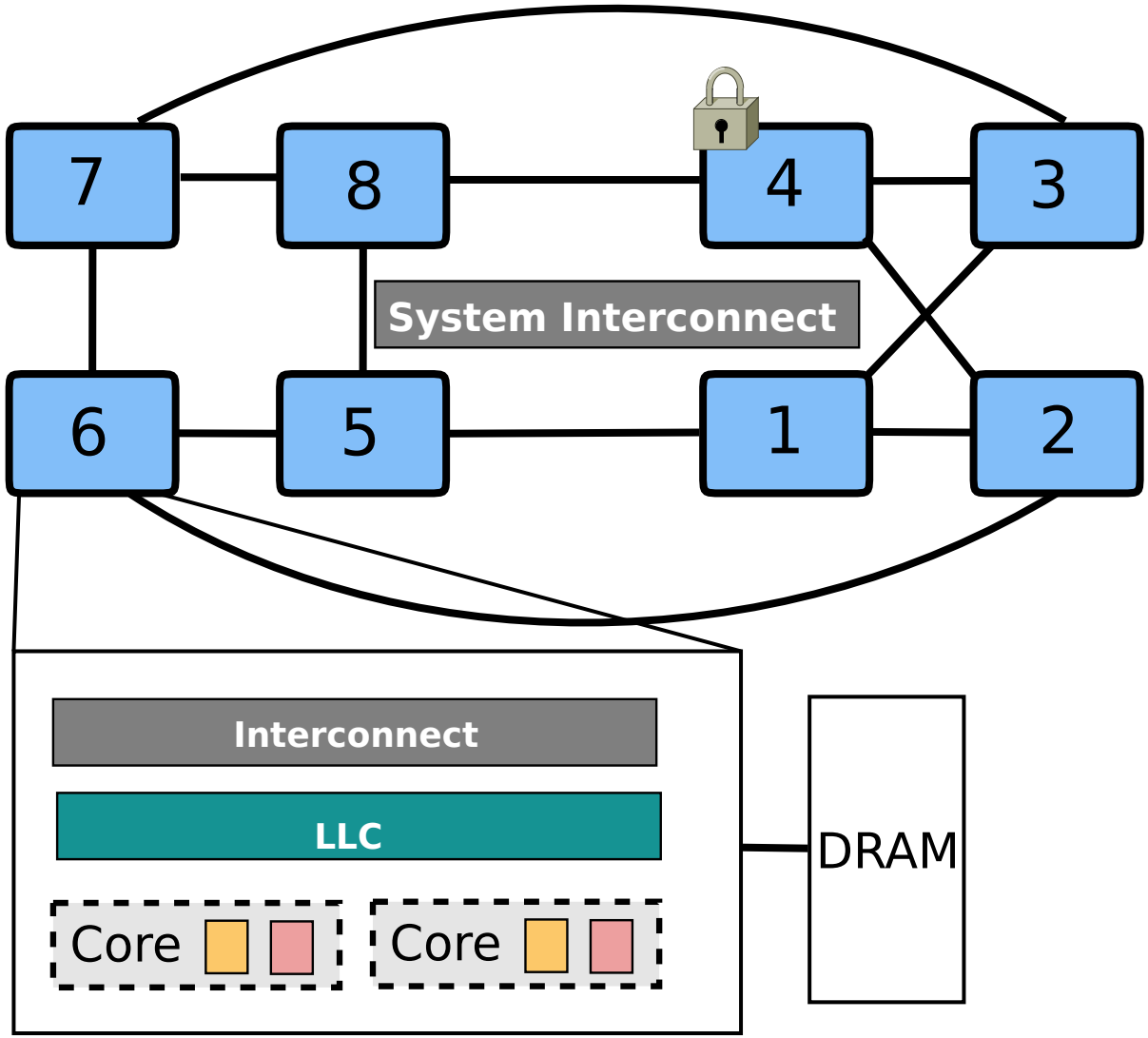


40 Years of Microprocessor Trend Data

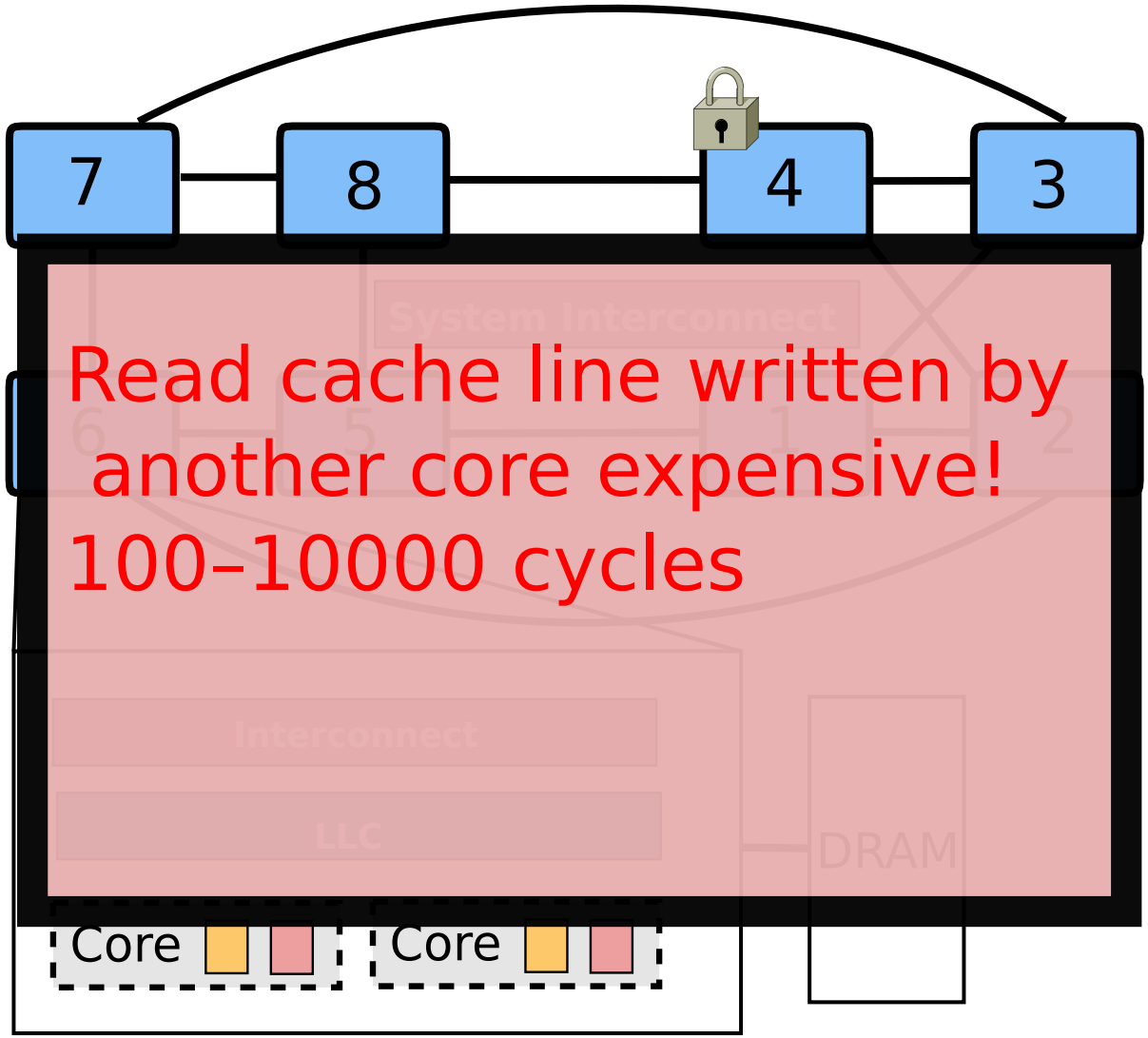


Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

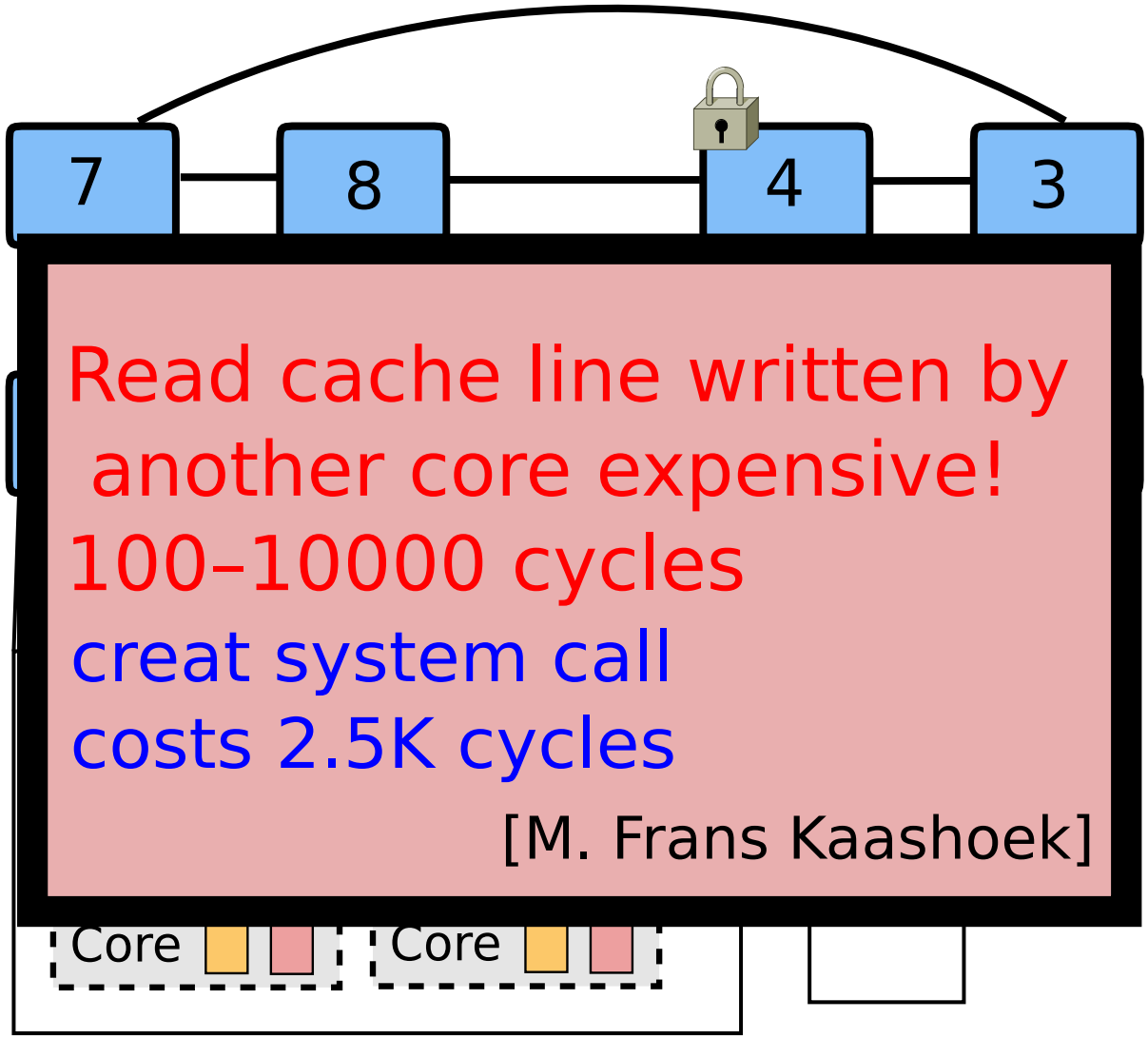
Cache-Coherence System



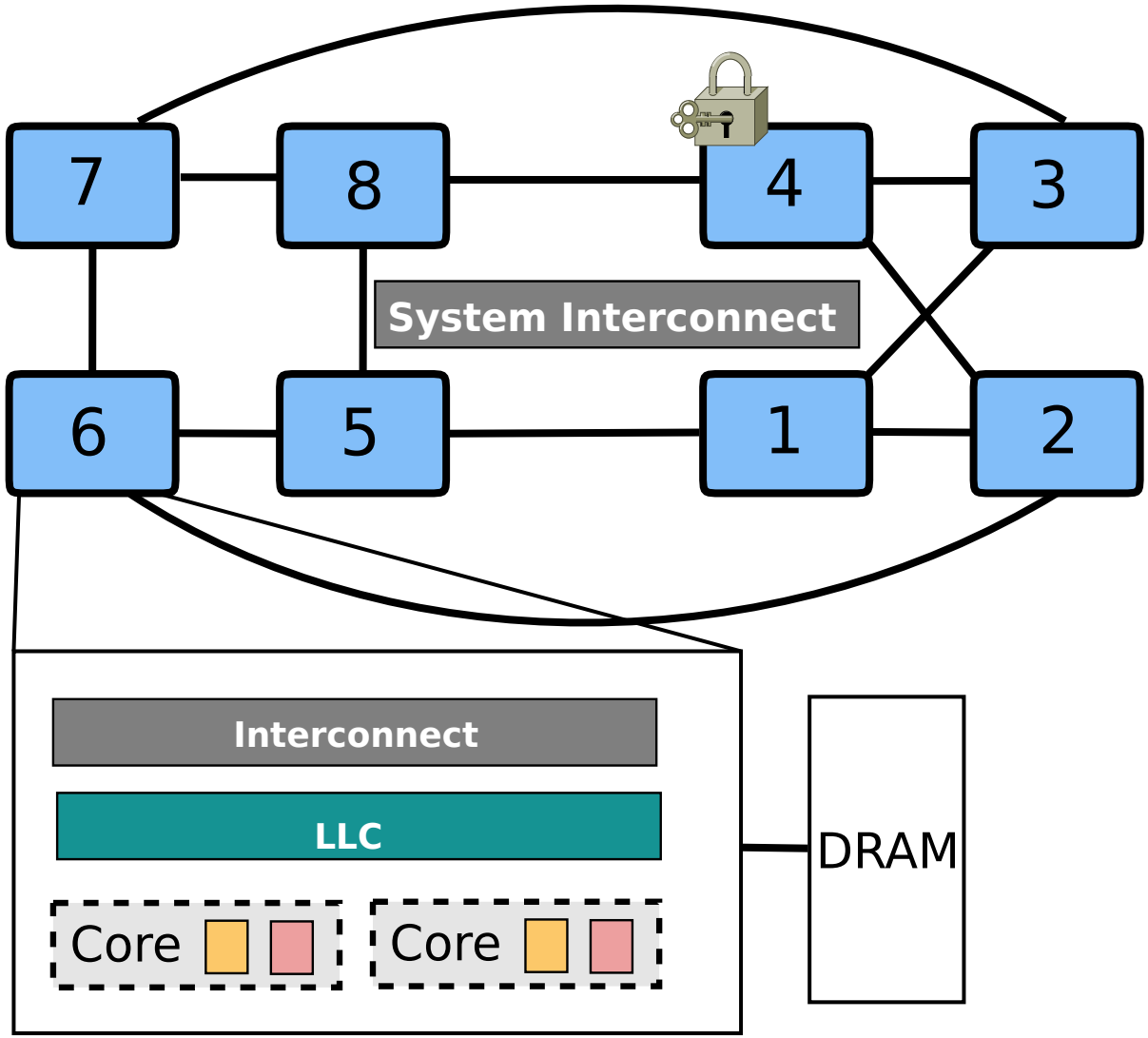
Cache-Coherence System



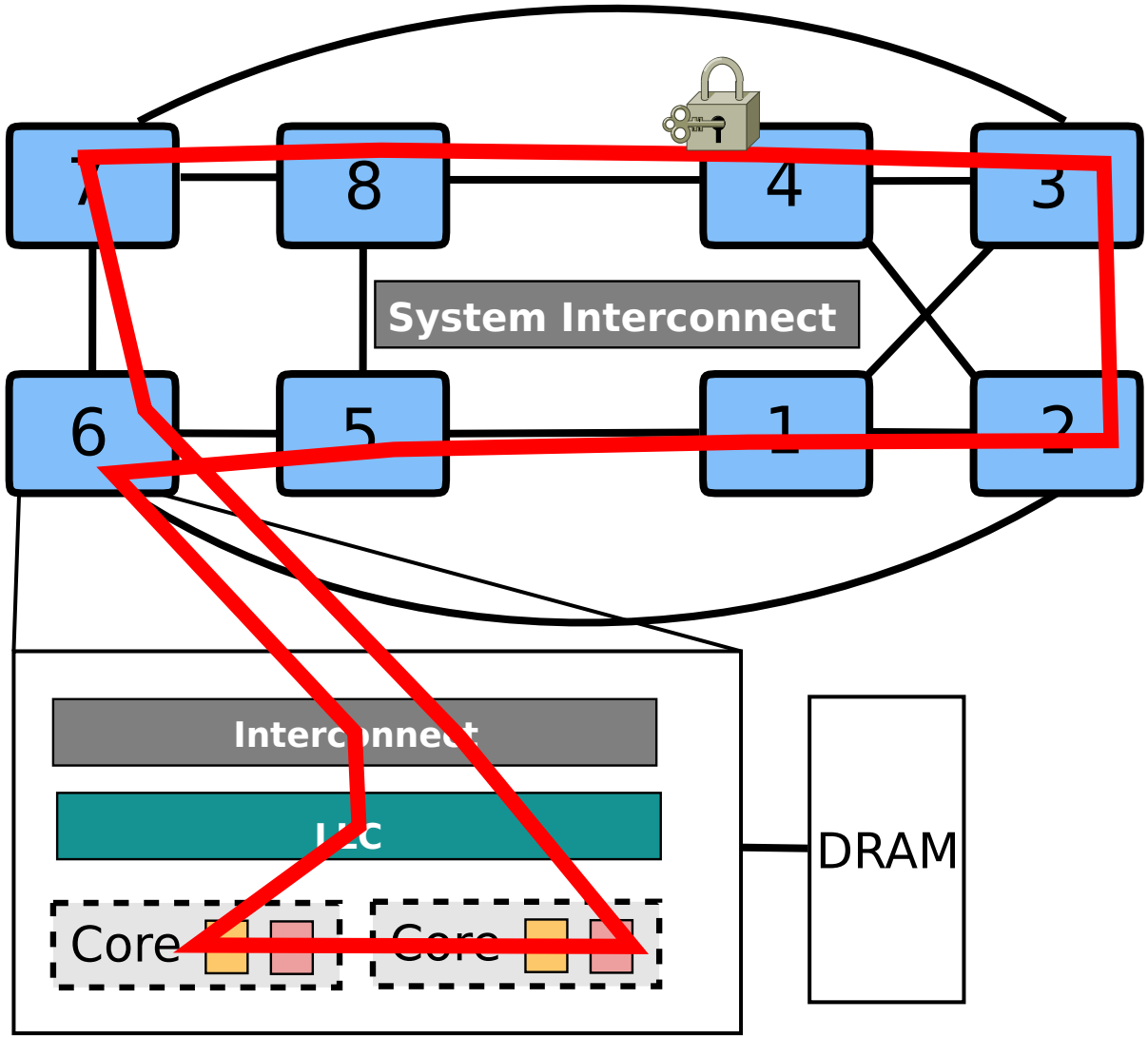
Cache-Coherence System



Cache-Coherence System

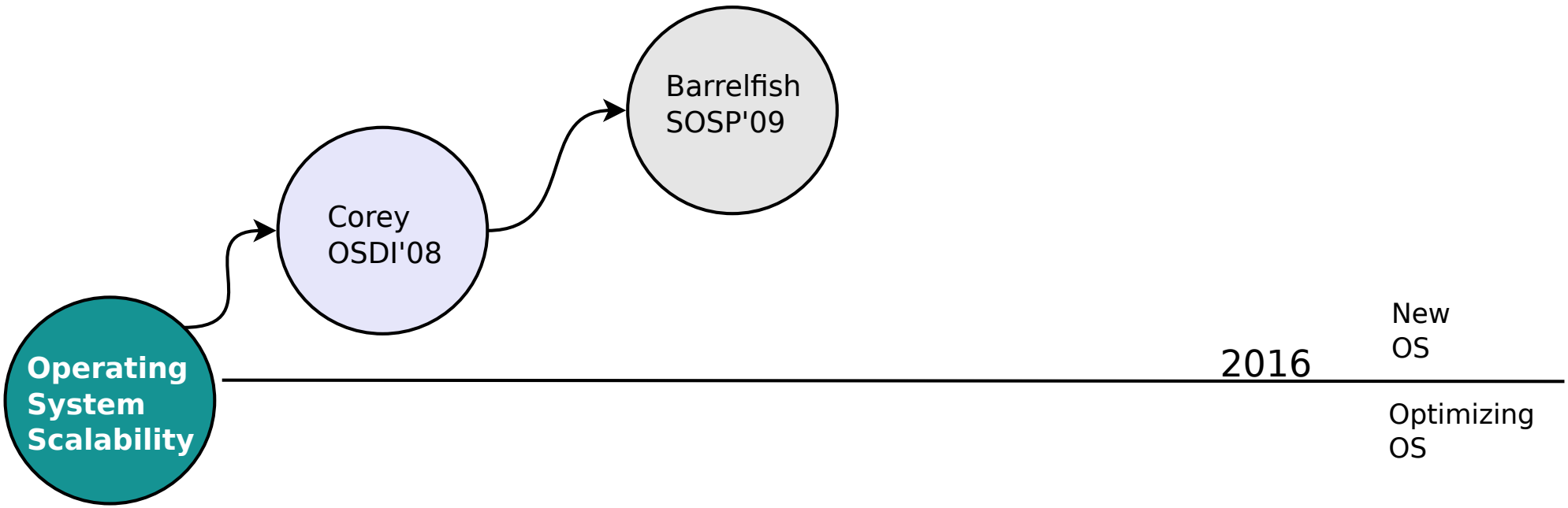


Cache-Coherence System

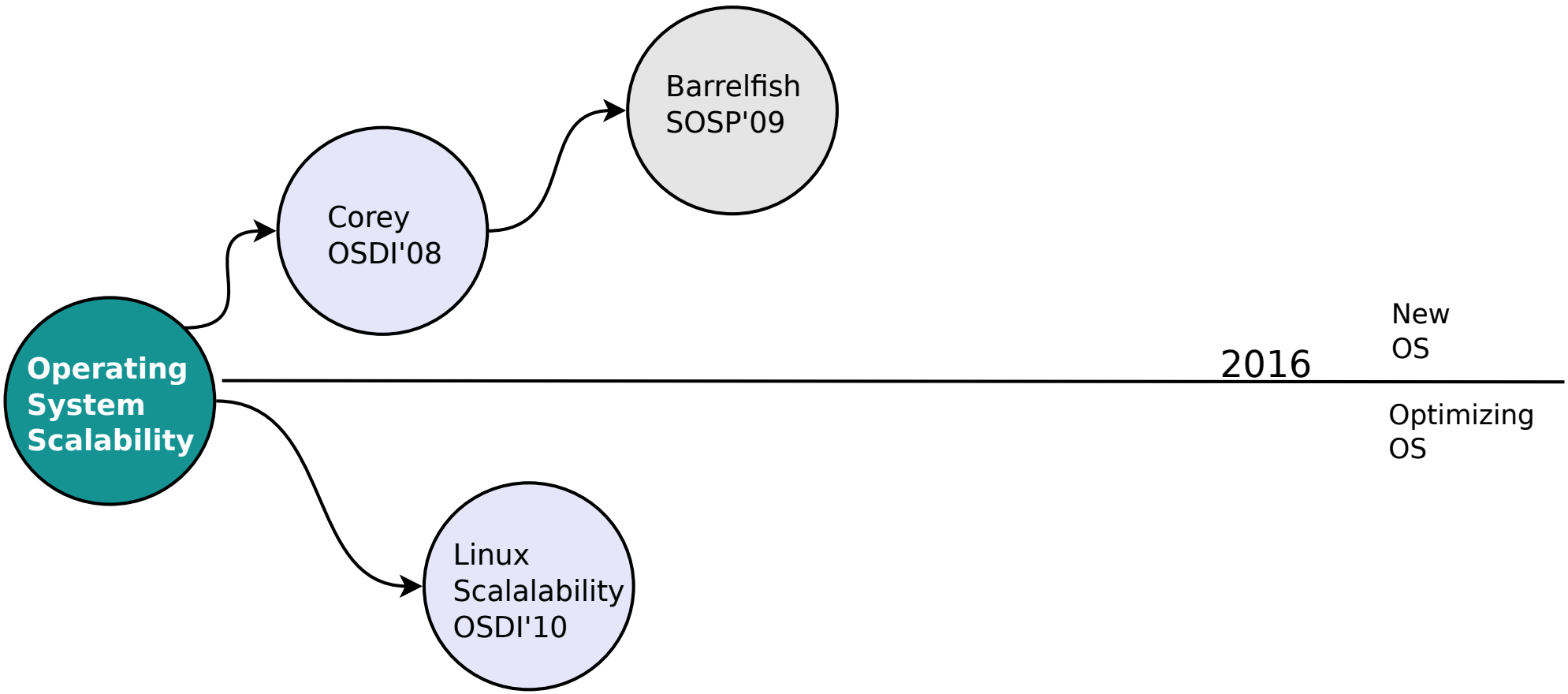


OS Kernel Scalability History

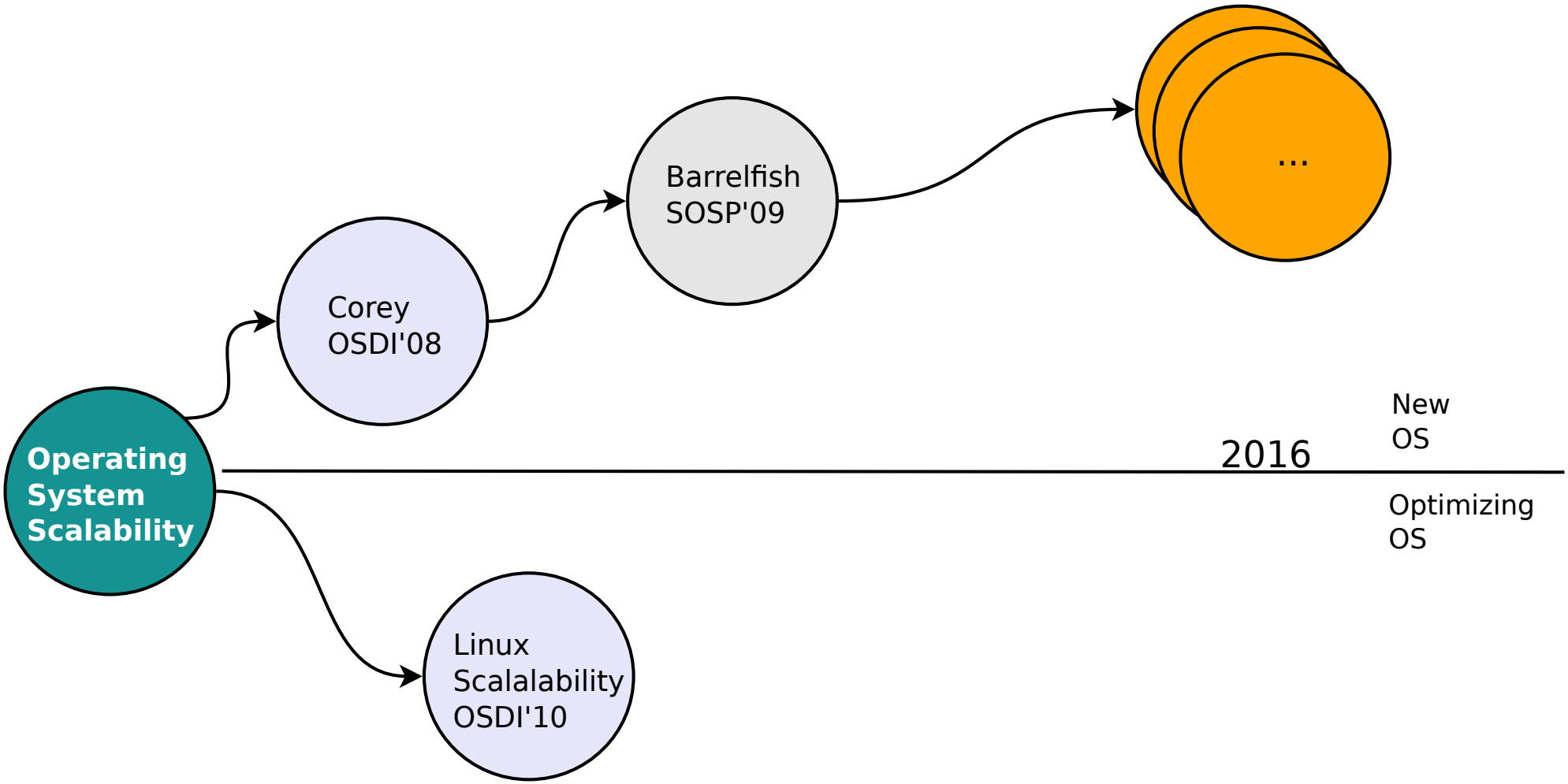
OS Kernel Scalability History



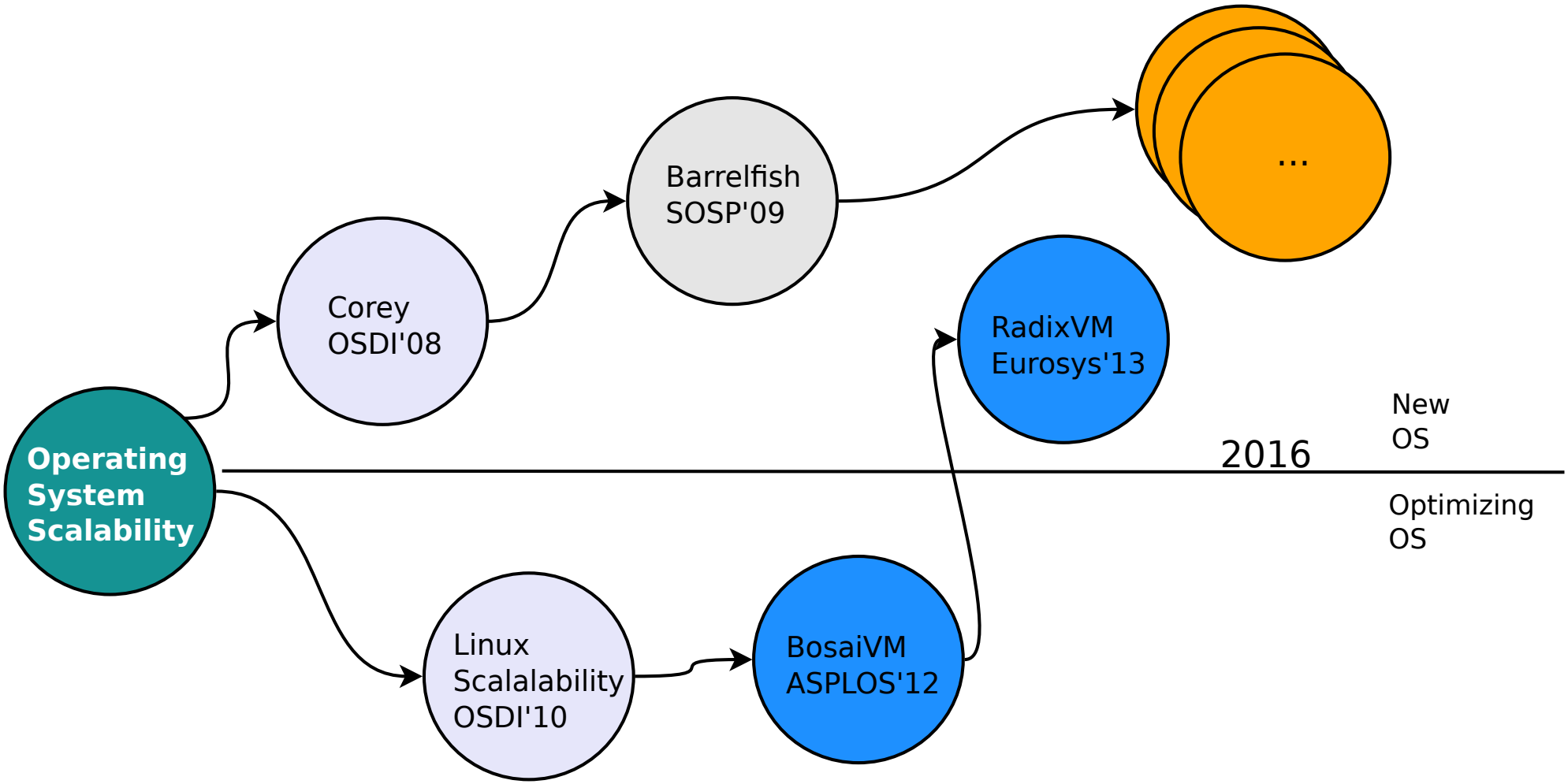
OS Kernel Scalability History



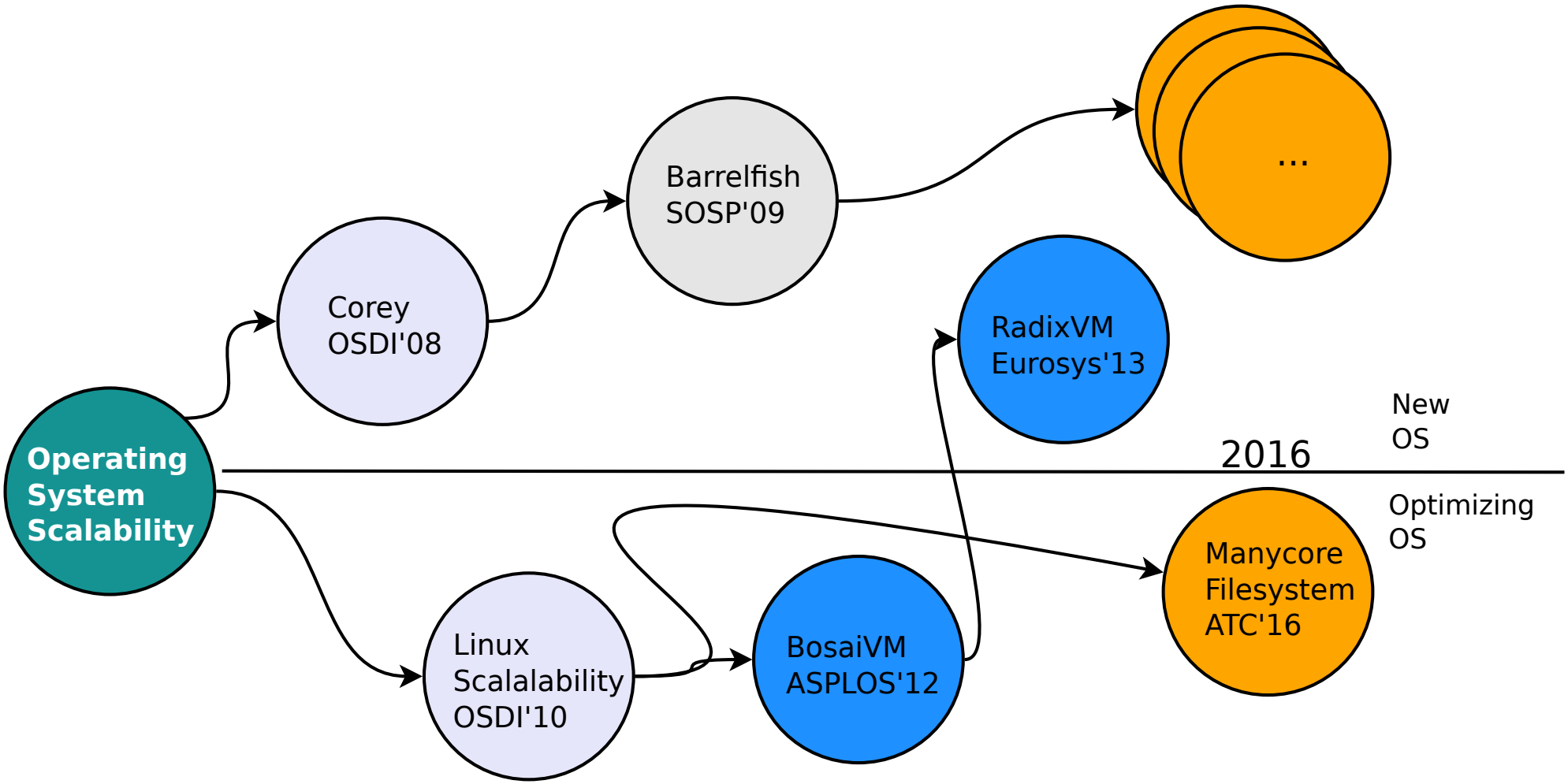
OS Kernel Scalability History



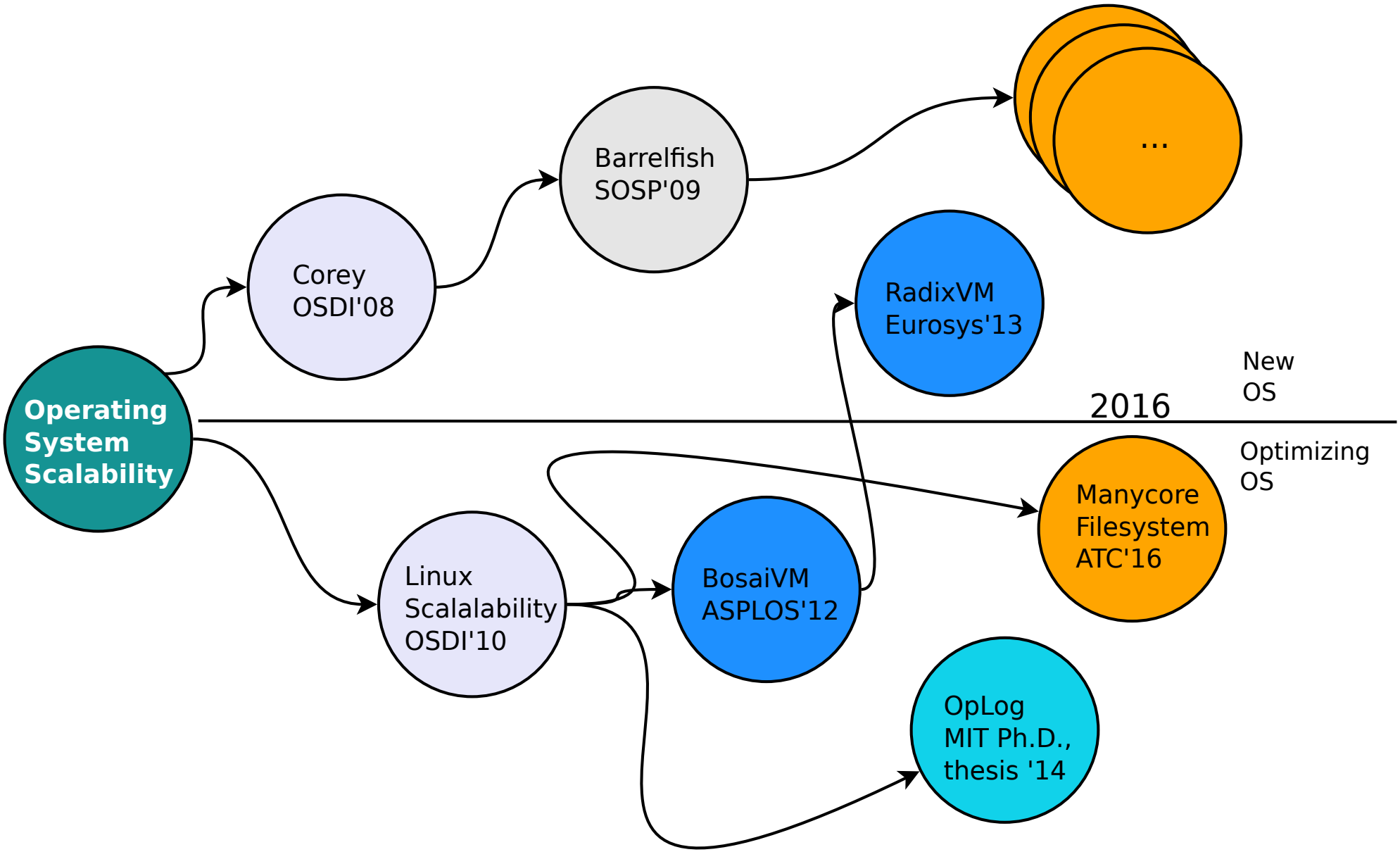
OS Kernel Scalability History



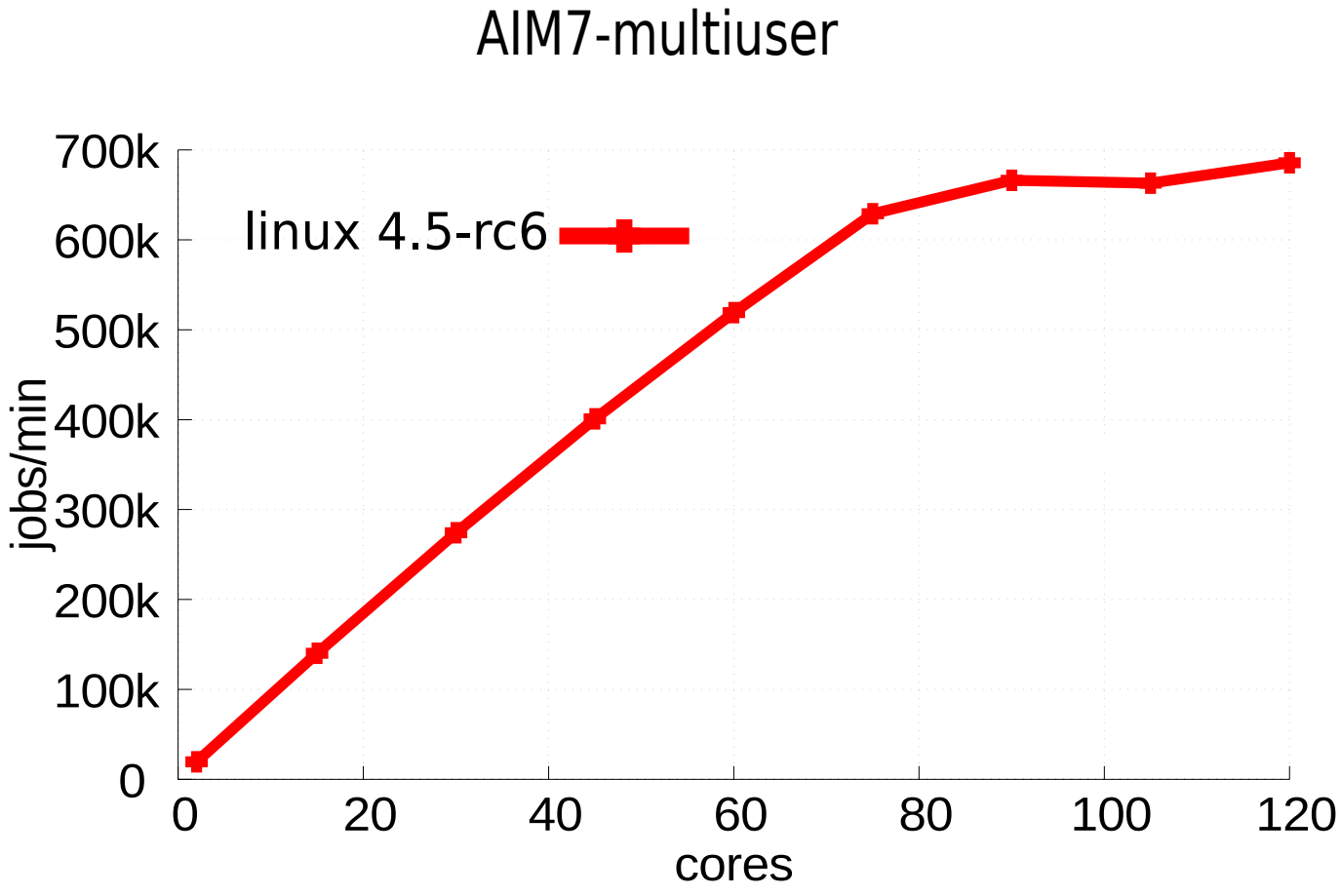
OS Kernel Scalability History



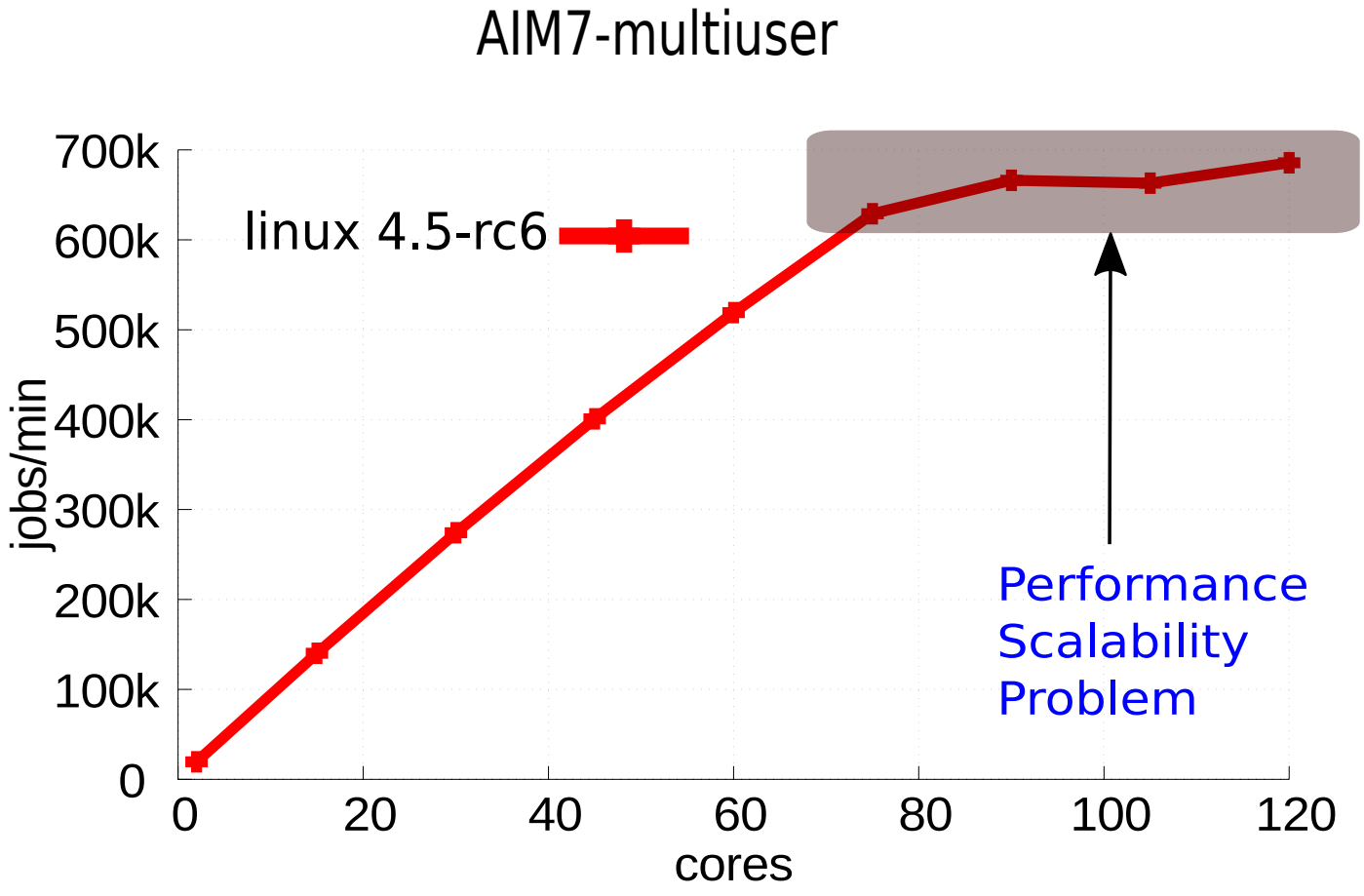
OS Kernel Scalability History



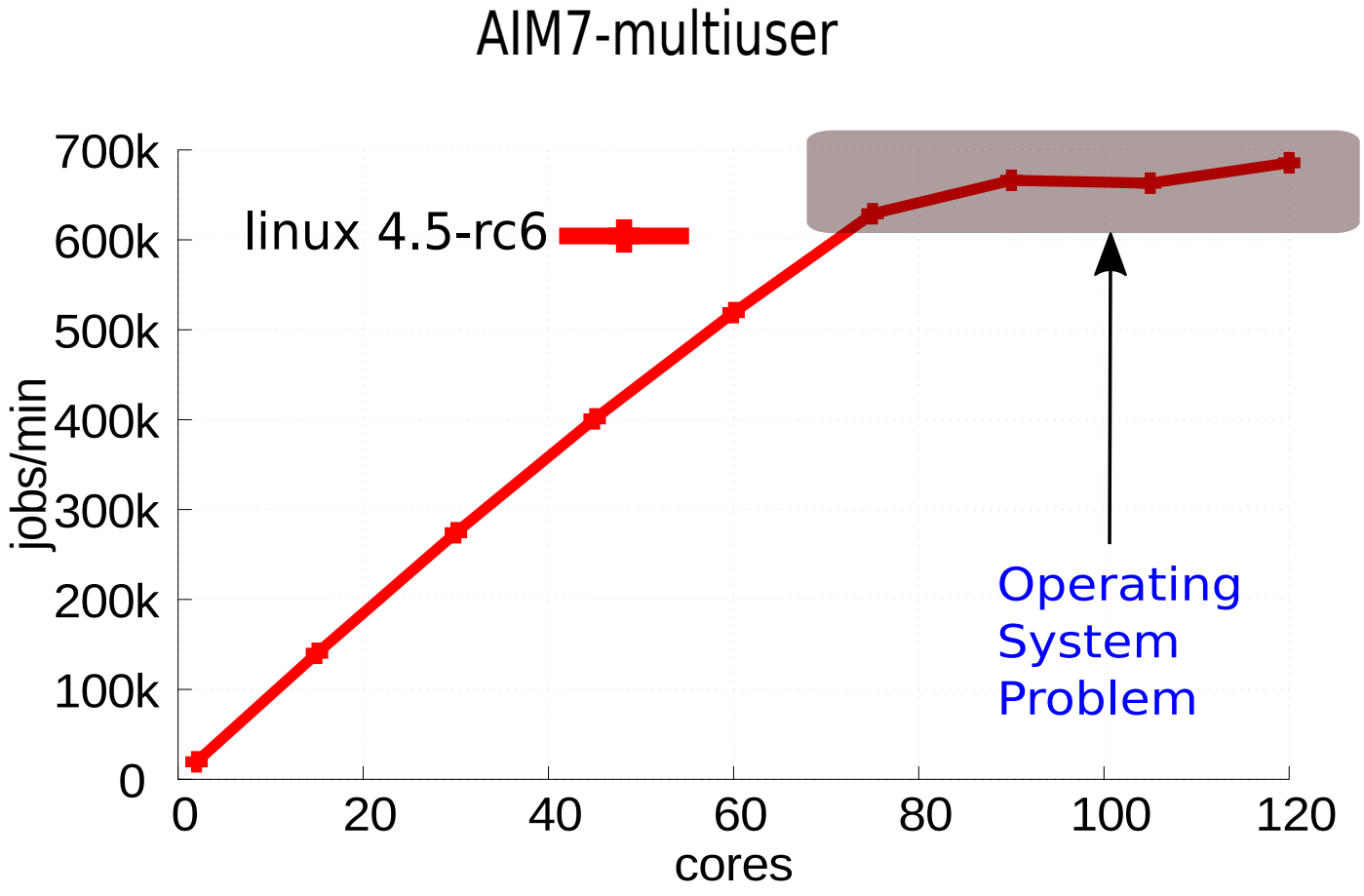
Performance Scalability



Performance Scalability



Performance Scalability



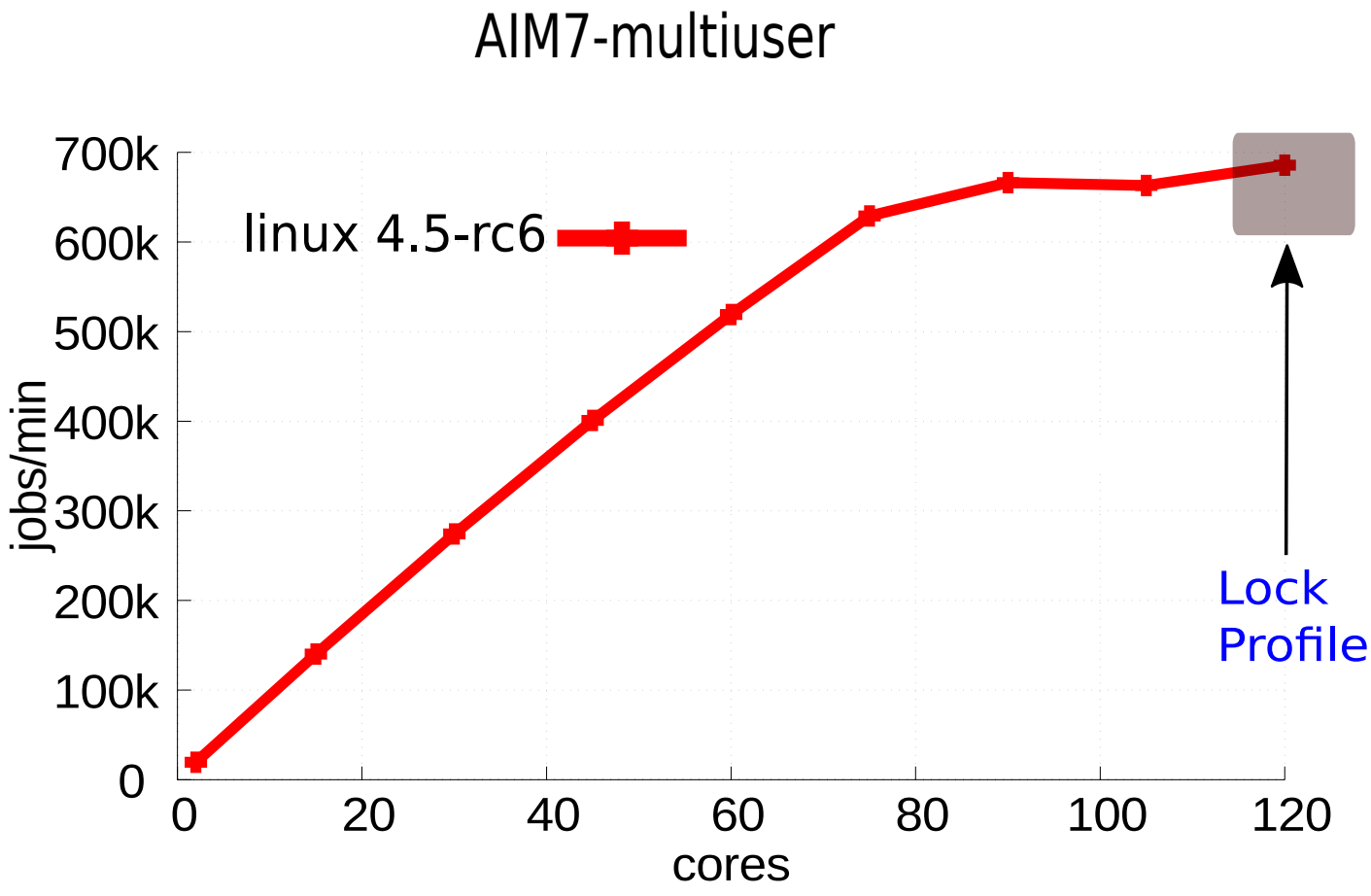
OS Kernel Scalability

- OS kernel scalability is an important part for the whole the system parallelism.
- If the kernel does not scale, applications will not scale.

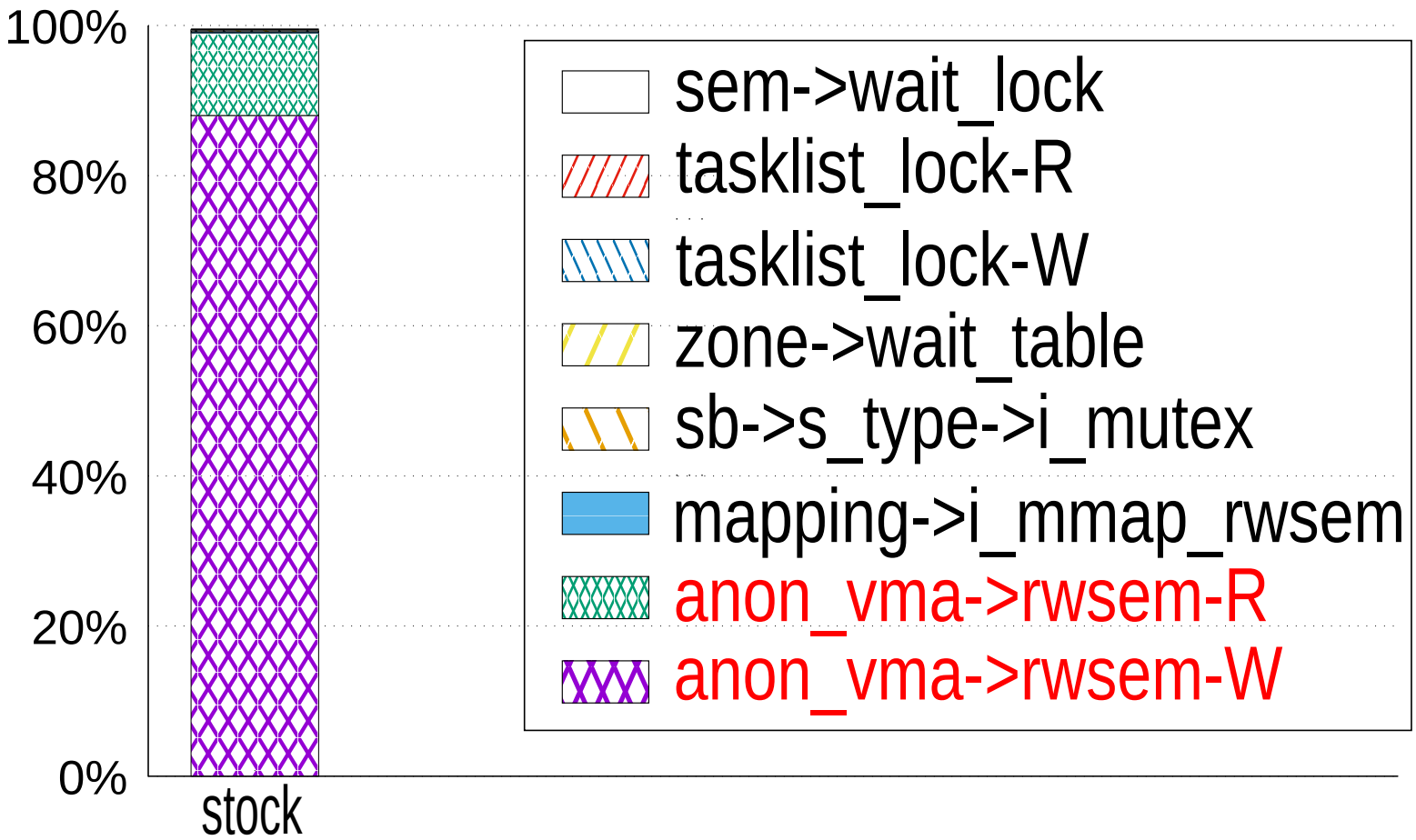
OS Kernel Scalability

- OS kernel scalability is an important part for the whole the system parallelism.
- If the kernel does not scale, applications will not scale.

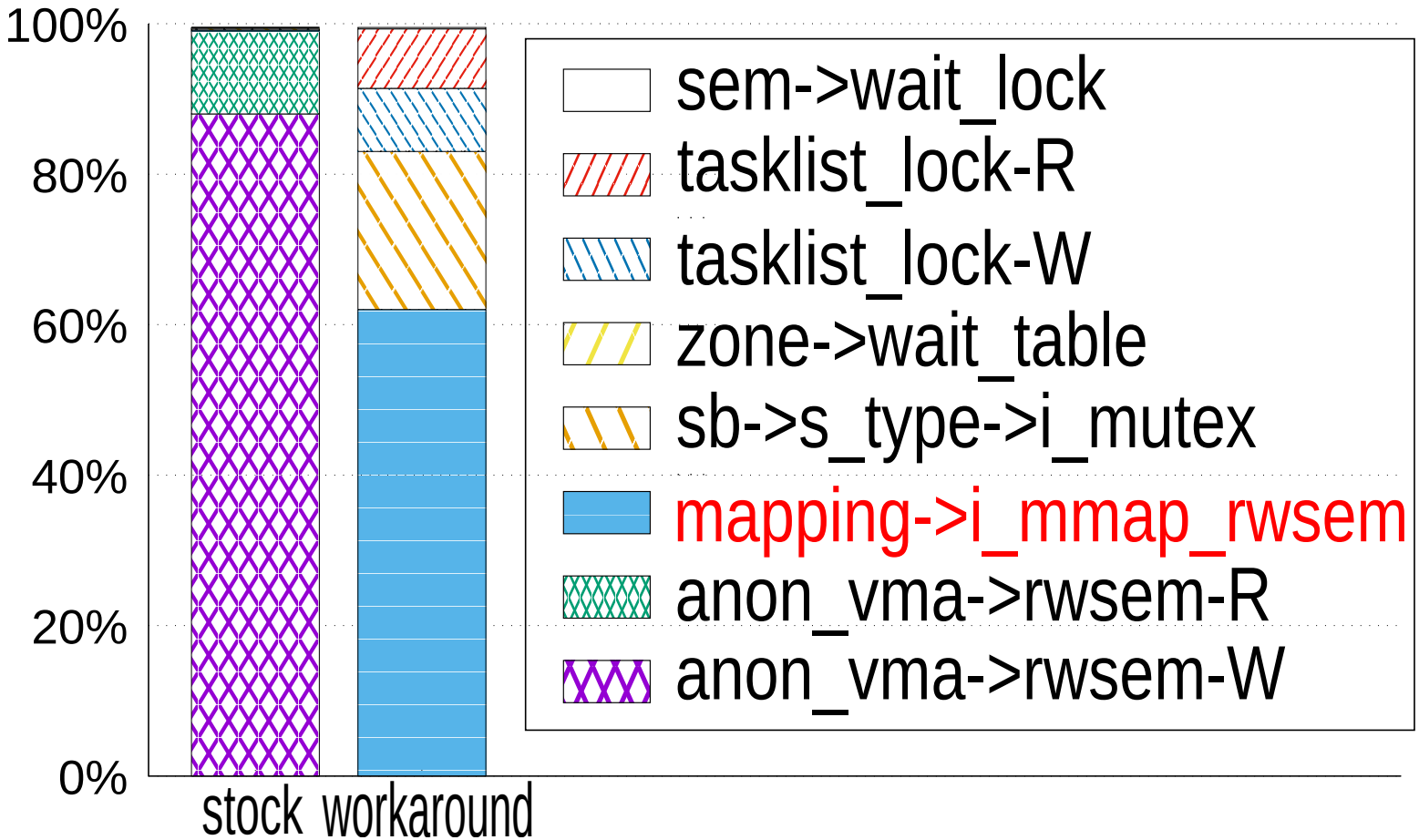
Lock Profile on 120 core



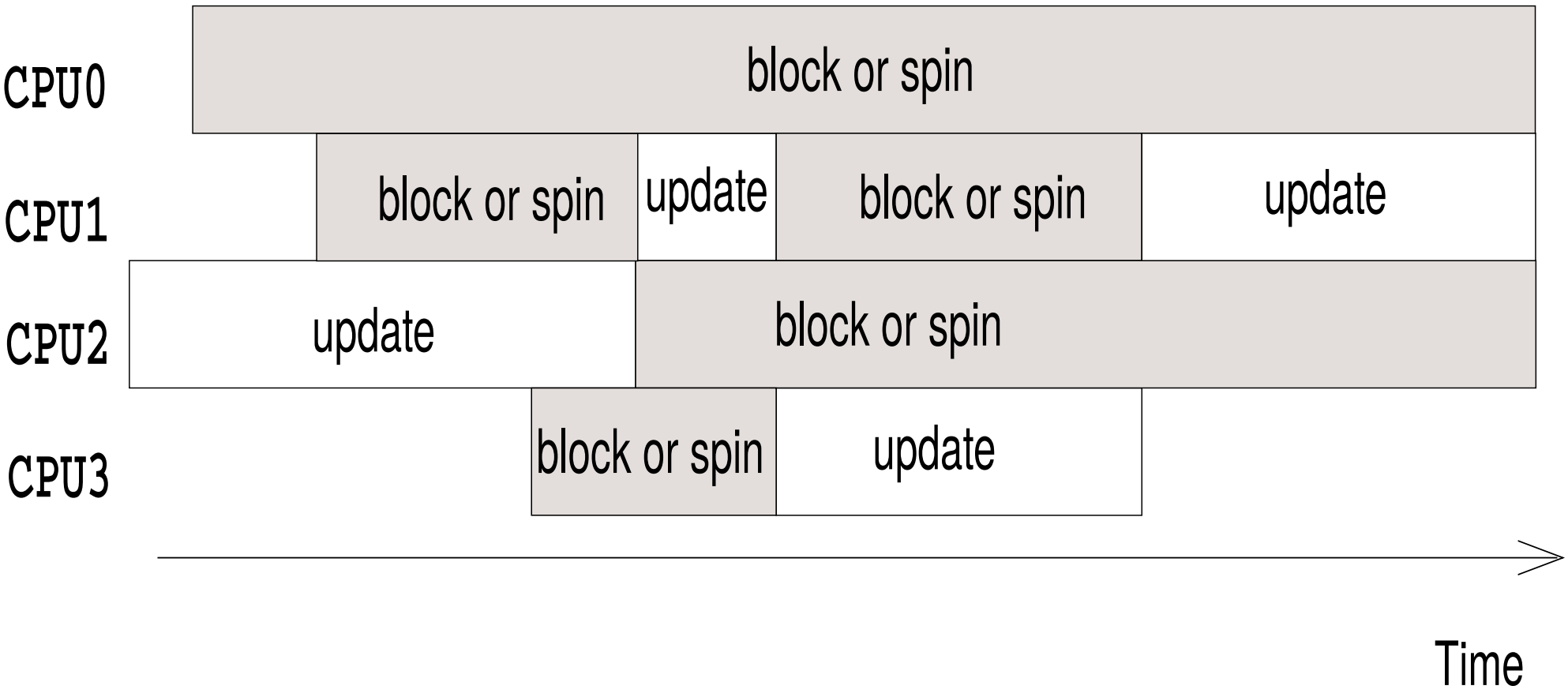
Wait time to acquire the lock



Wait time to acquire the lock



Update Serialization



Solution for High Update Rate Data Structure

- Non-blocking Data Structure.
  - Harris, Fraser.
- Log-based Algorithms.

Solution for High Update Rate Data Structure

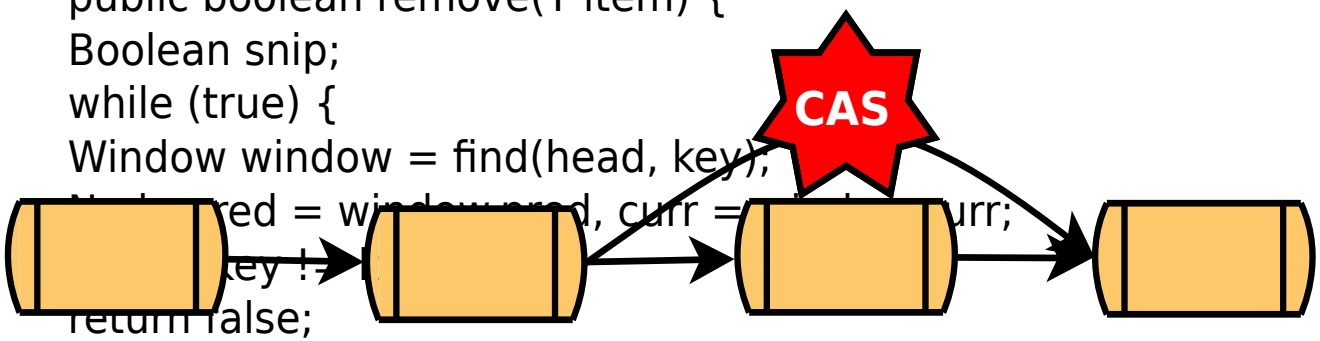
- Non-blocking Data Structure.
  - Harris, Fraser.
- Log-based Algorithms.

Solution for High Update Rate Data Structure

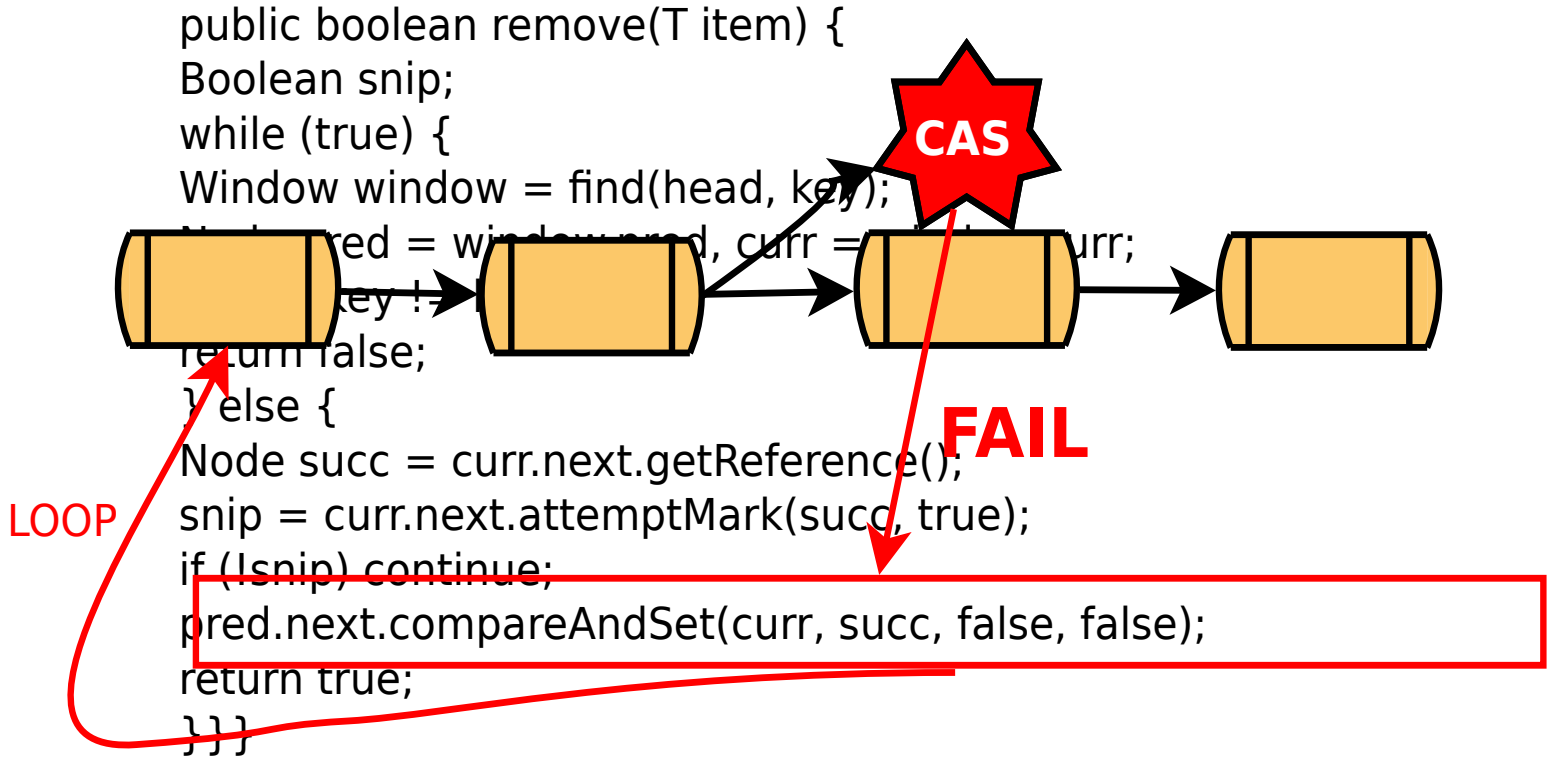
- Non-blocking Data Structure.
  - Harris, Fraser.
- Log-based Algorithms.

Non-blocking Data Structure

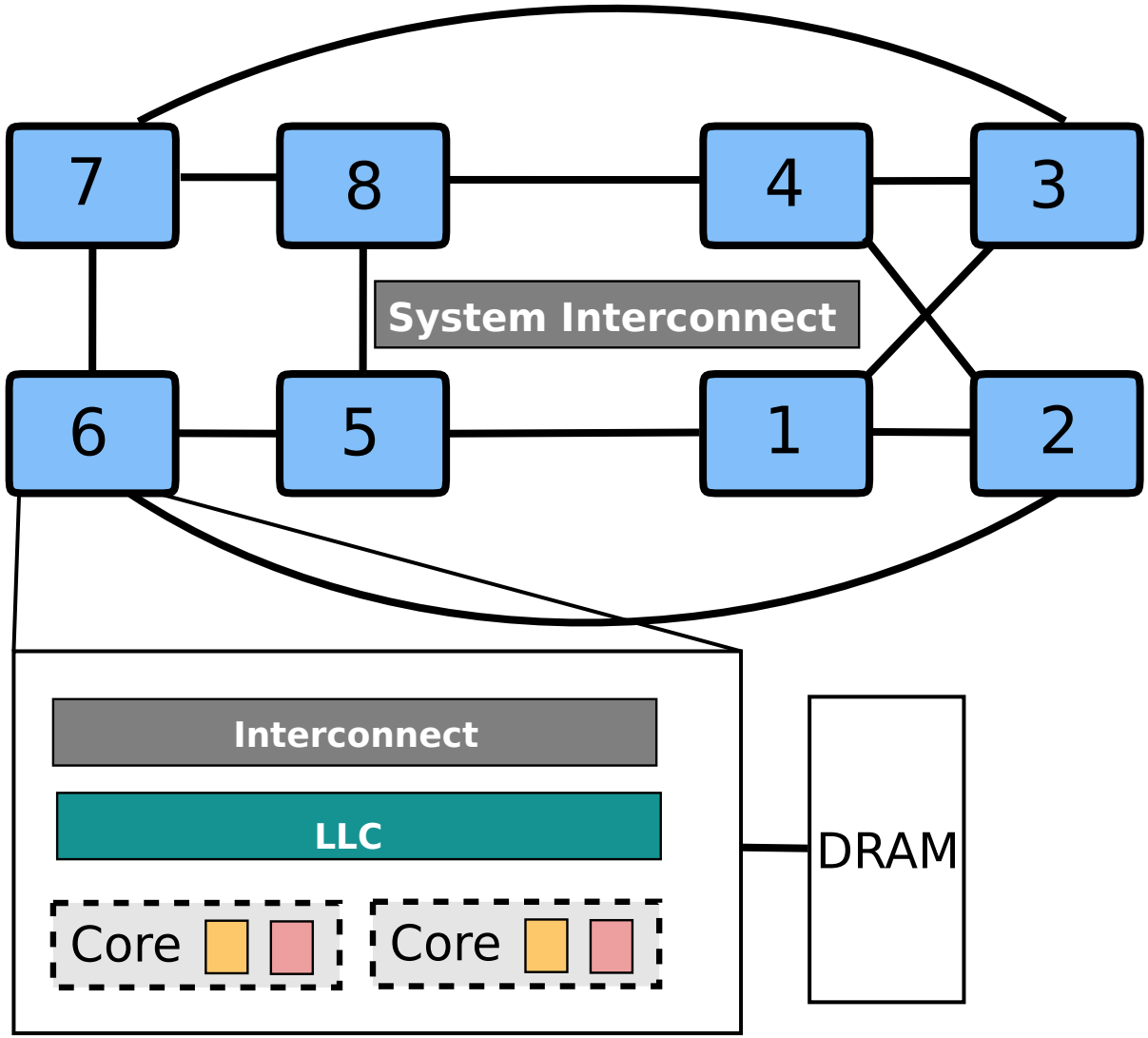
```
public boolean remove(T item) {
Boolean snip;
while (true) {
Window window = find(head, key);
N   red = w   d, curr =   urr;
   ey !=
return false;
} else {
Node succ = curr.next.getReference();
snip = curr.next.attemptMark(succ, true);
if (!snip) continue;
pred.next.compareAndSet(curr, succ, false, false);
return true;
}}}
```



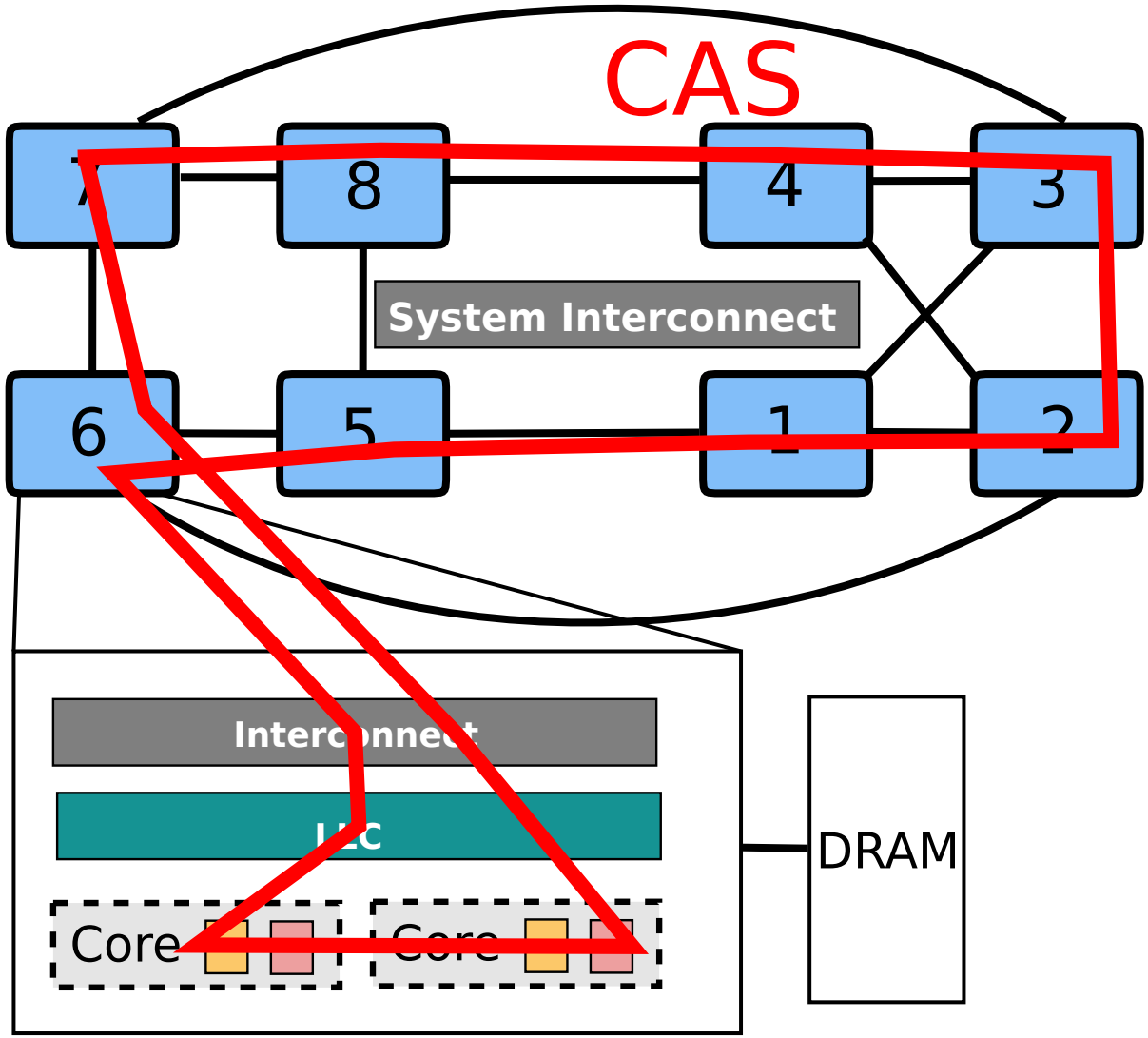
Non-blocking Data Structure



Cache communication bottlenecks



Cache communication bottlenecks



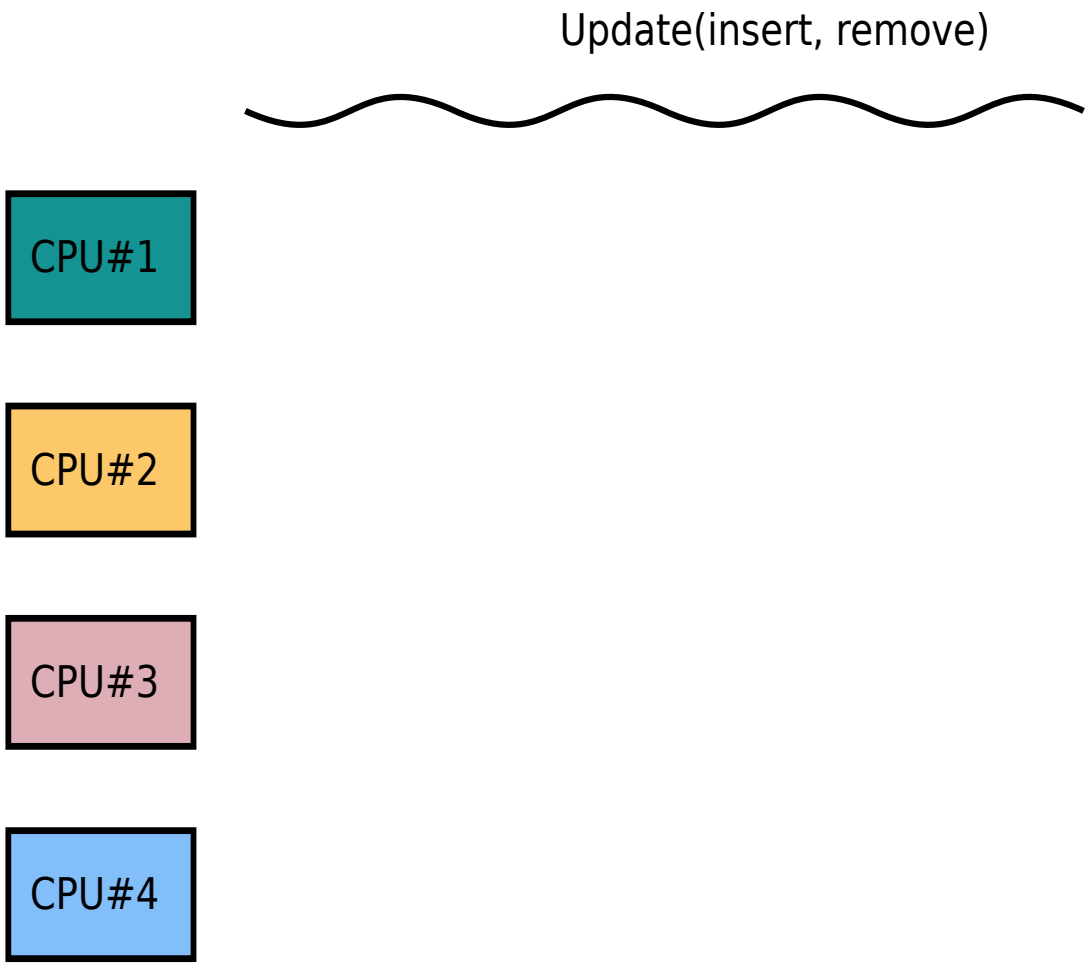
Log-based Algorithms

- Flat combining(SPAA '10).
- OpLog(MIT Ph.D., thesis 2013).

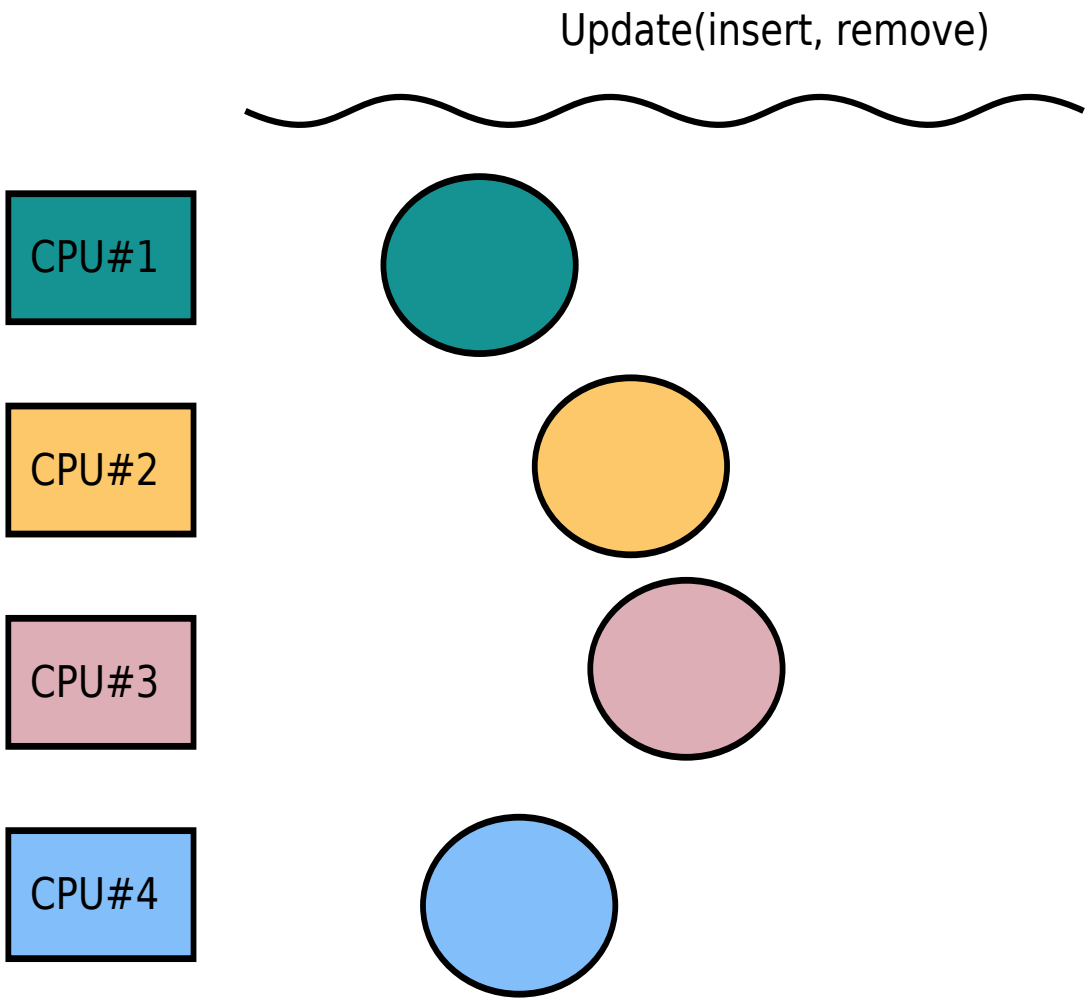
Log-based Algorithms

- Flat combining(SPAA '10).
- OpLog(MIT Ph.D., thesis 2013).

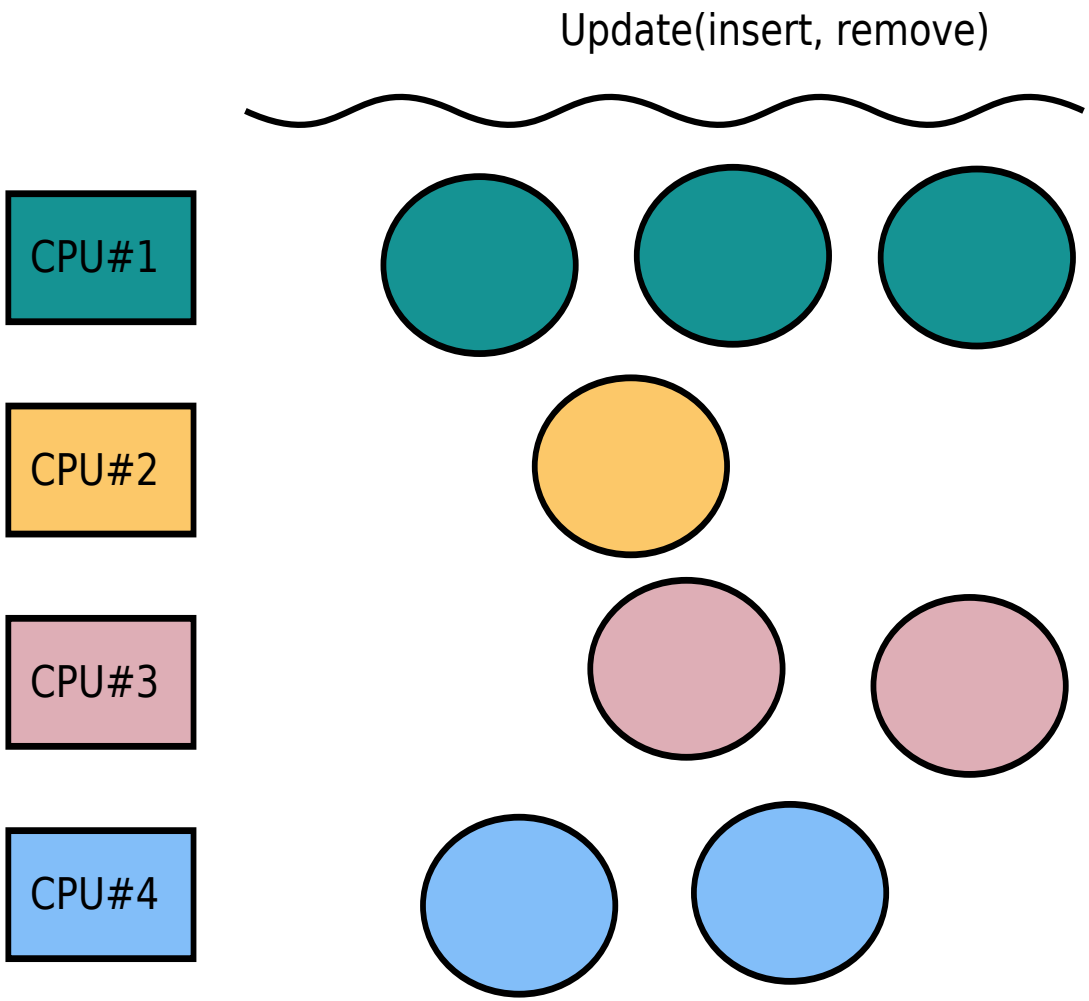
Log-based Algorithms: OpLog



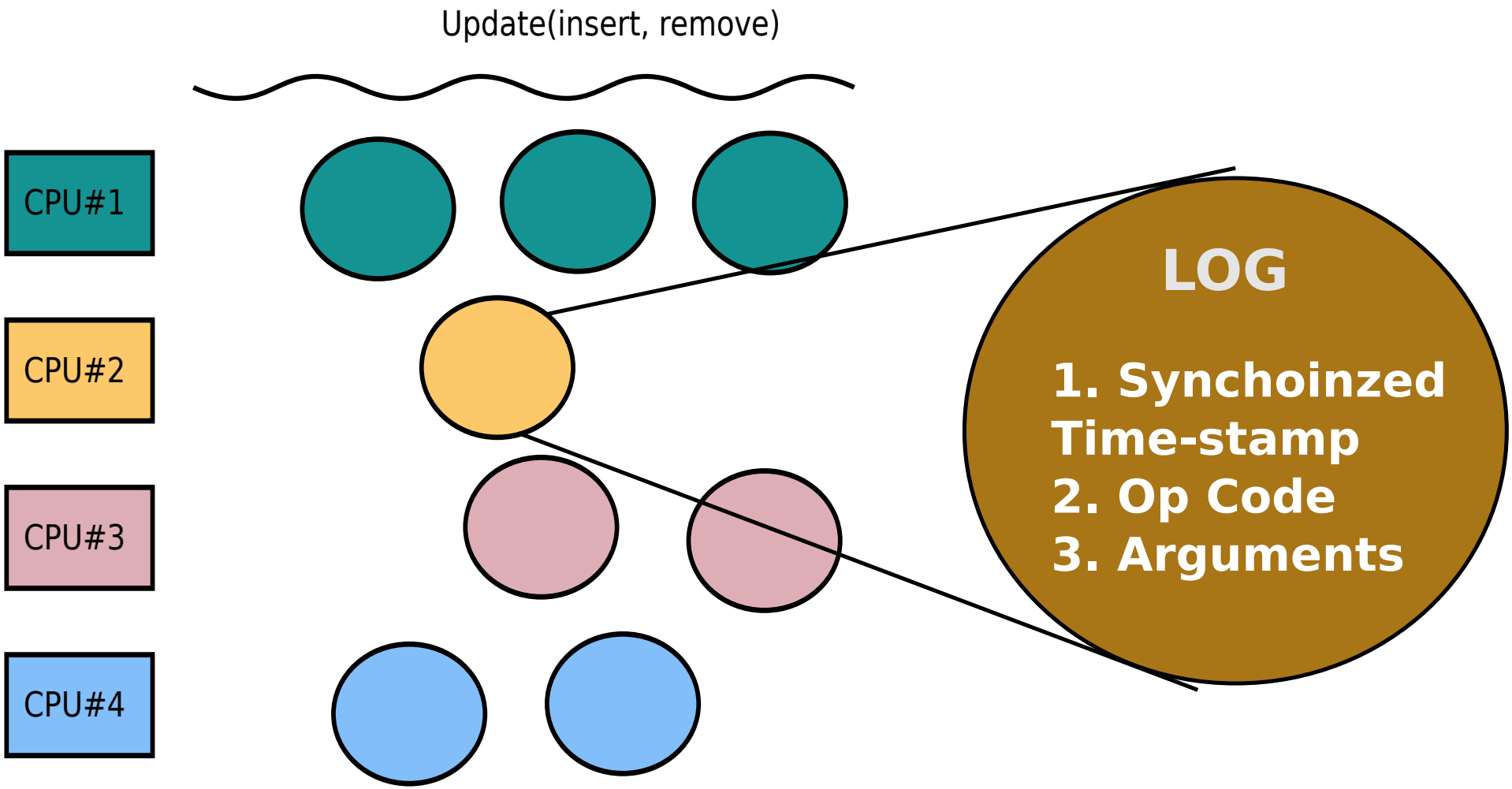
Log-based Algorithms: OpLog



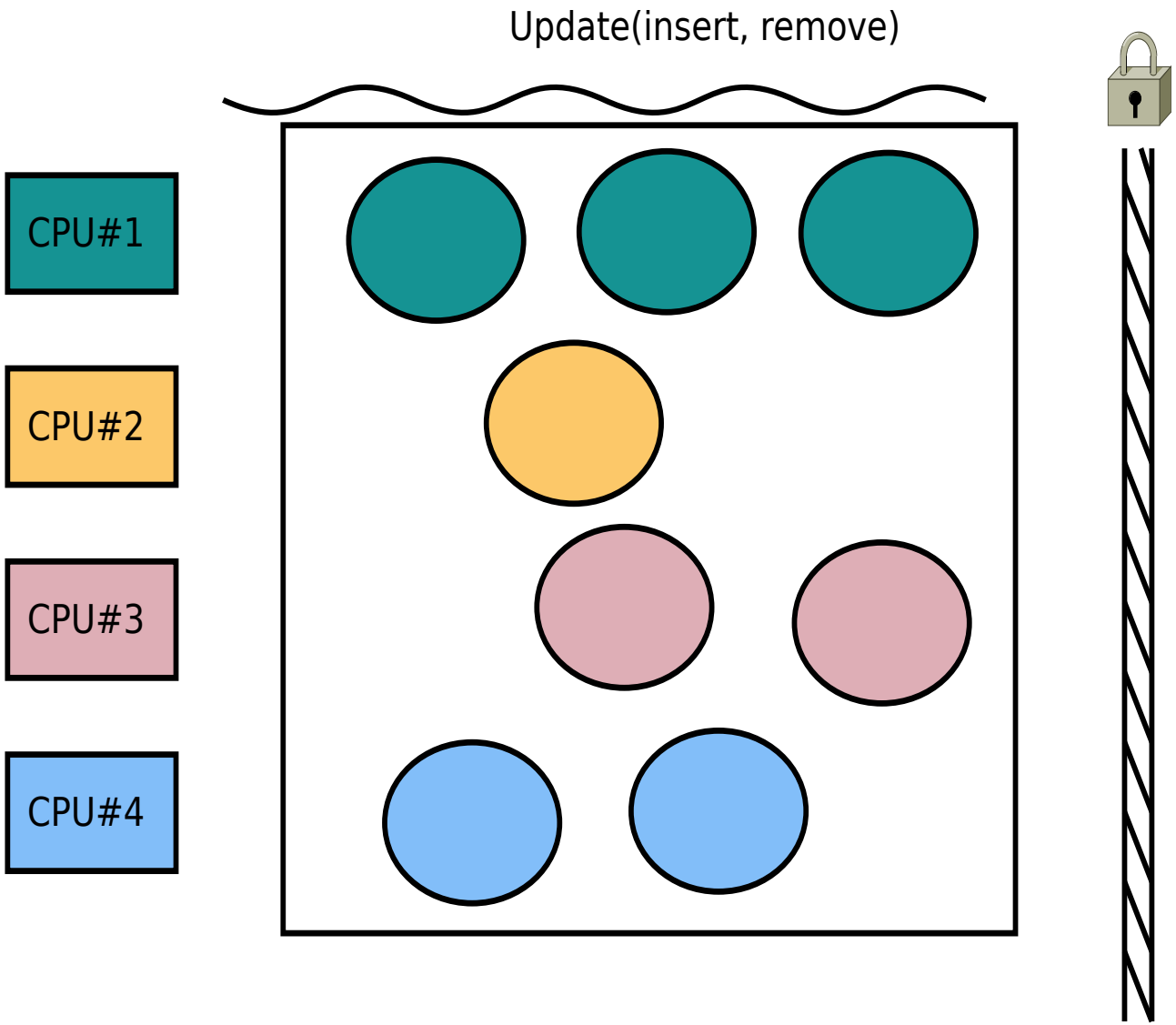
Log-based Algorithms: OpLog



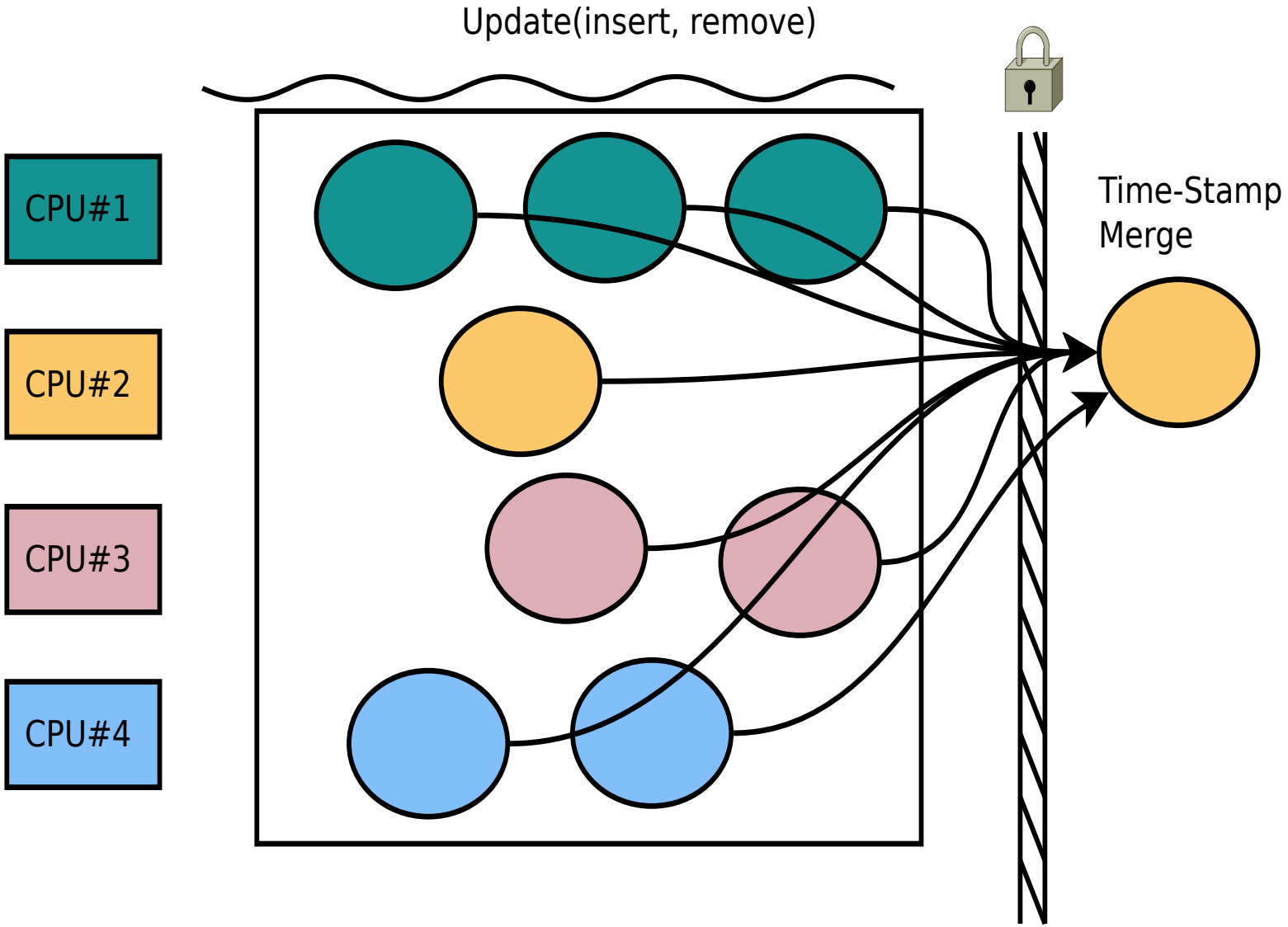
Log-based Algorithms: OpLog



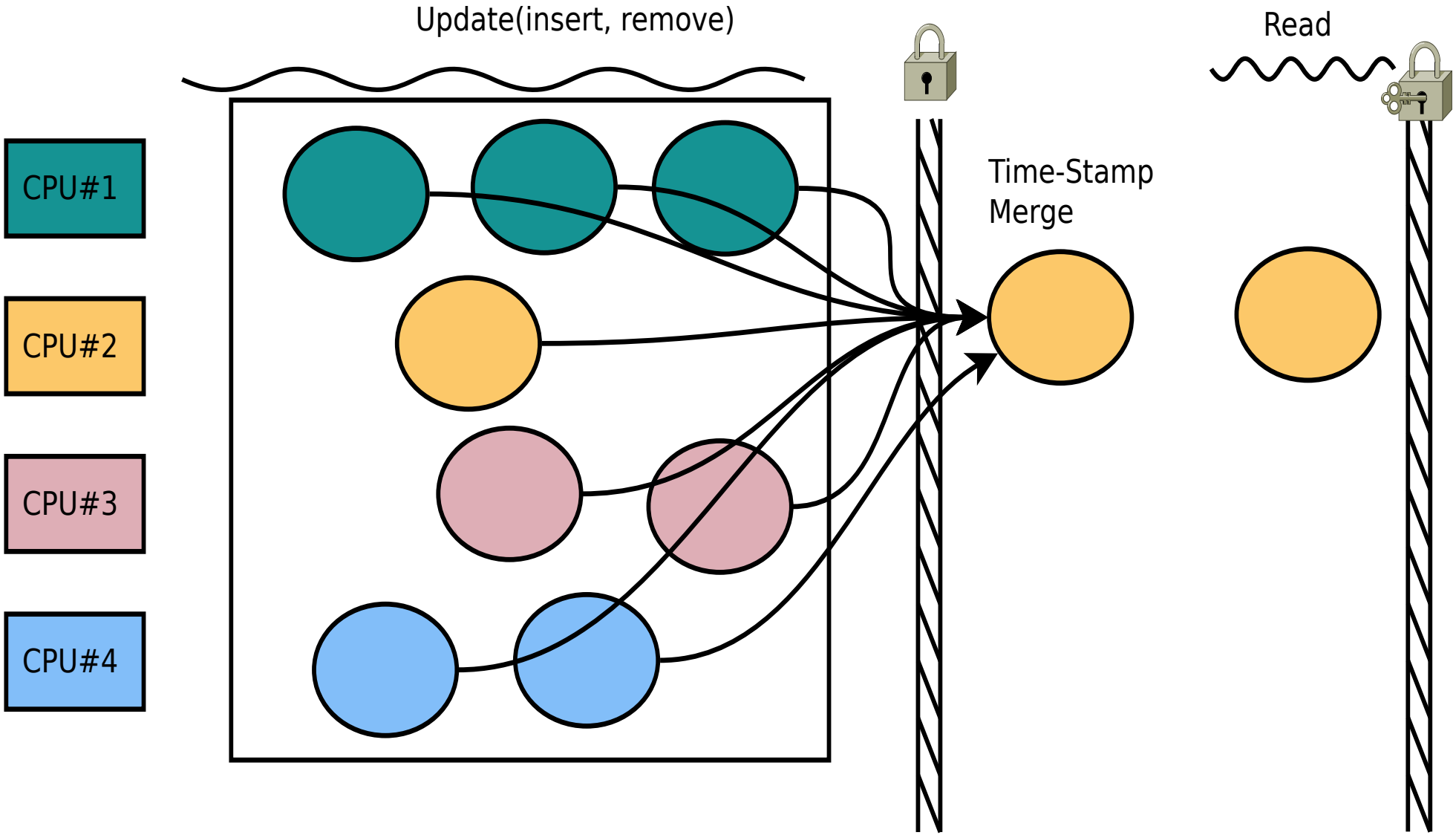
Log-based Algorithms: OpLog



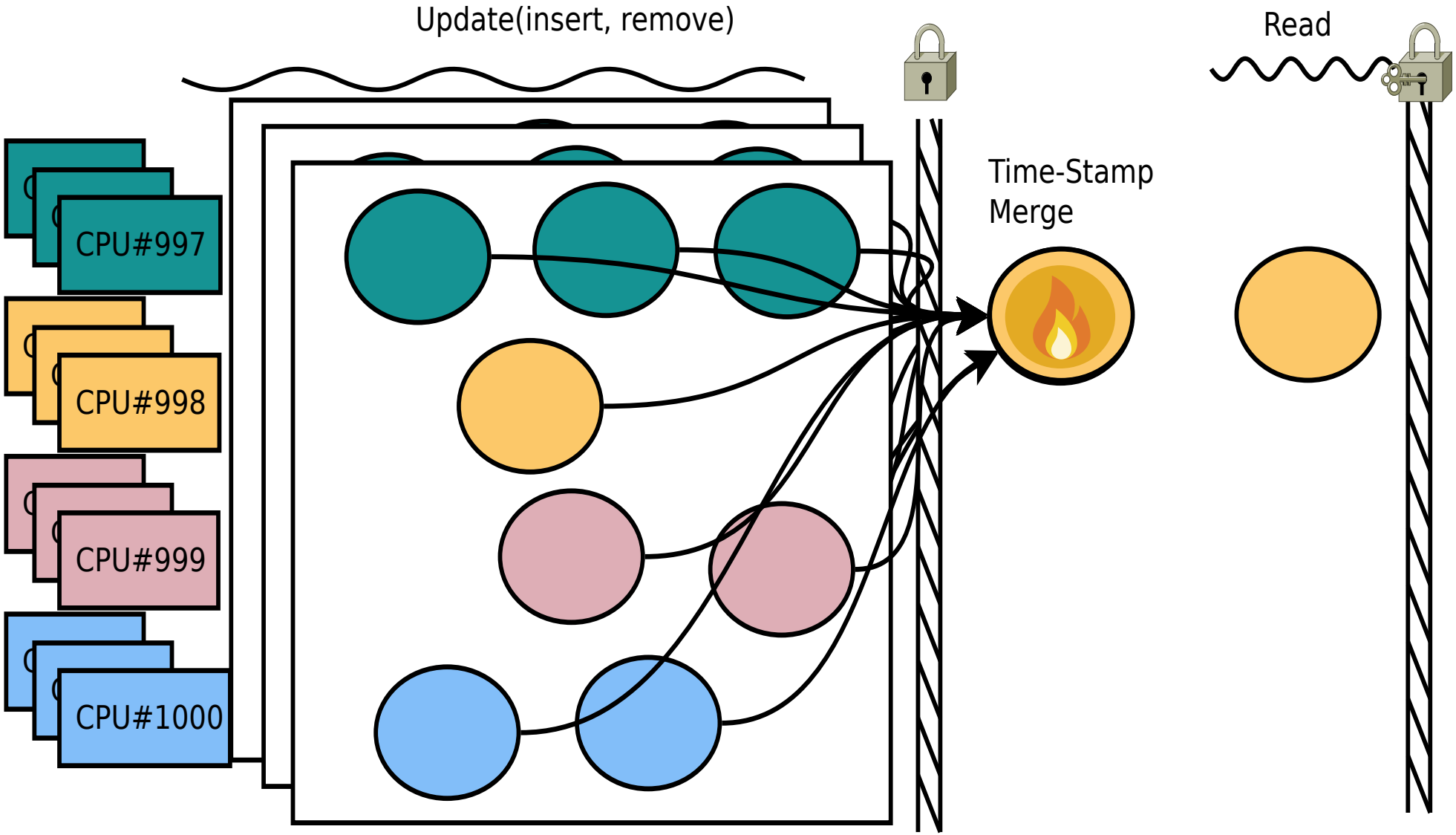
Log-based Algorithms: OpLog



Log-based Algorithms: OpLog



Time-stamp counters



Simple Solution?

Simple Solution

- A Lightweight Log-based Deferred Update(LDU).

Contributions

- Removing time-stamp counters.
- Applying the LDU to two reverse mapping(anonymous, file).
- Improved throughput and execution time from 1.5x through 2.7x on 120 core.

Contributions

- Removing time-stamp counters.
- Applying the LDU to two reverse mapping(anonymous, file).
- Improved throughput and execution time from 1.5x through 2.7x on 120 core.

Contributions

- Removing time-stamp counters.
- Applying the LDU to two reverse mapping(anonymous, file).
- Improved throughput and execution time from 1.5x through 2.7x on 120 core.

Outline

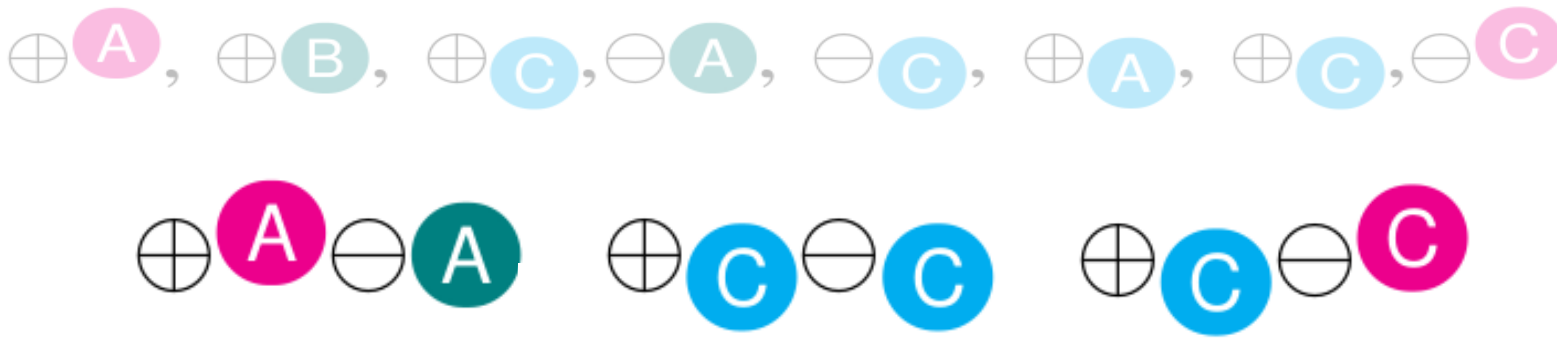
- Design
  - Approach
  - Example
- Applying the Linux kernel
- Implementation
- Evaluation

Why the OpLog needs the time-stamp counter?

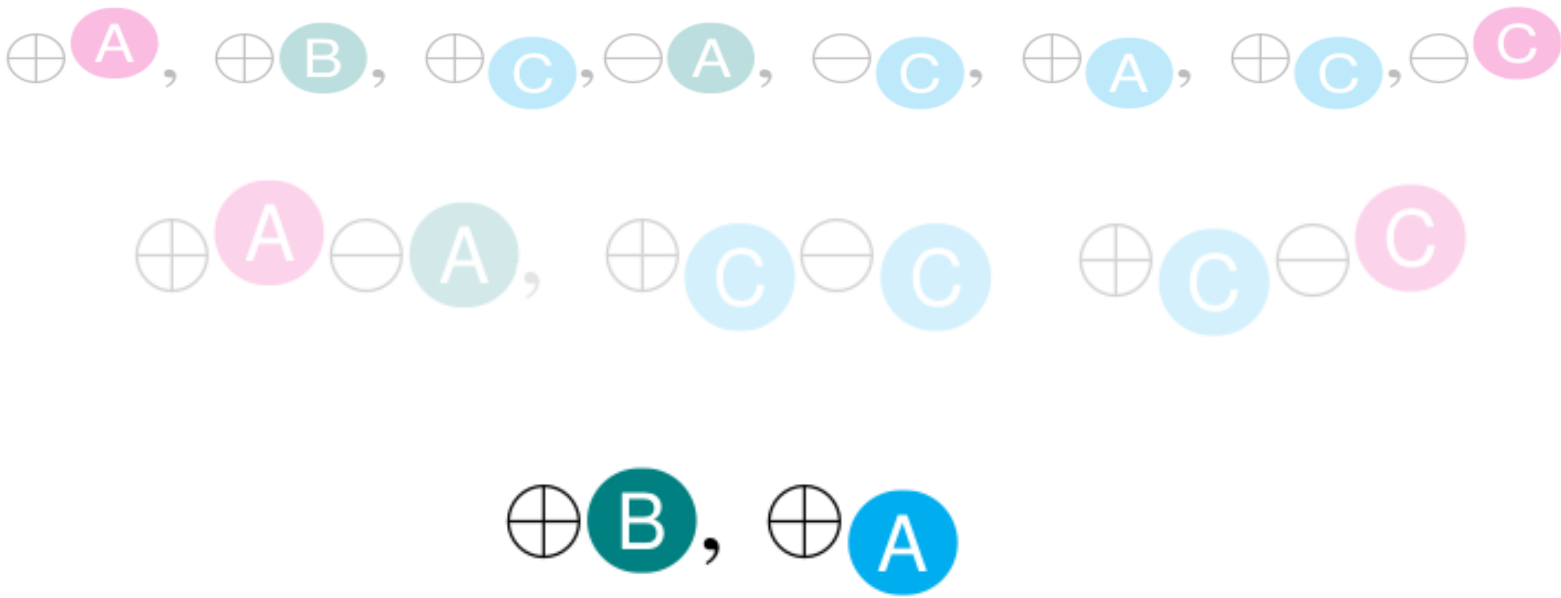
"A process logs an insert operation to the per-core memory, then it migrates to another core, and it logs a remove operation, which must eventually execute after the insert operation", OpLog.

Log Example

Cancelable – Log



Remain Log

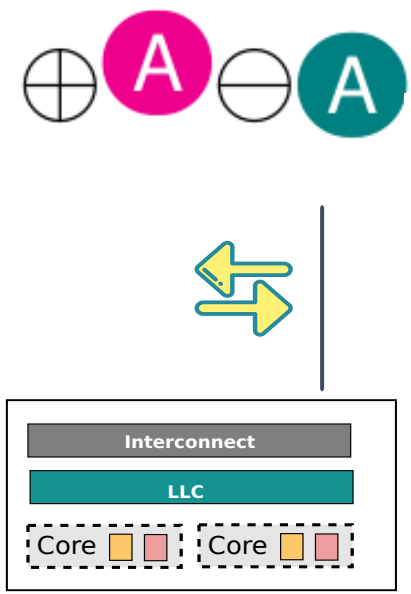


Update-side removing

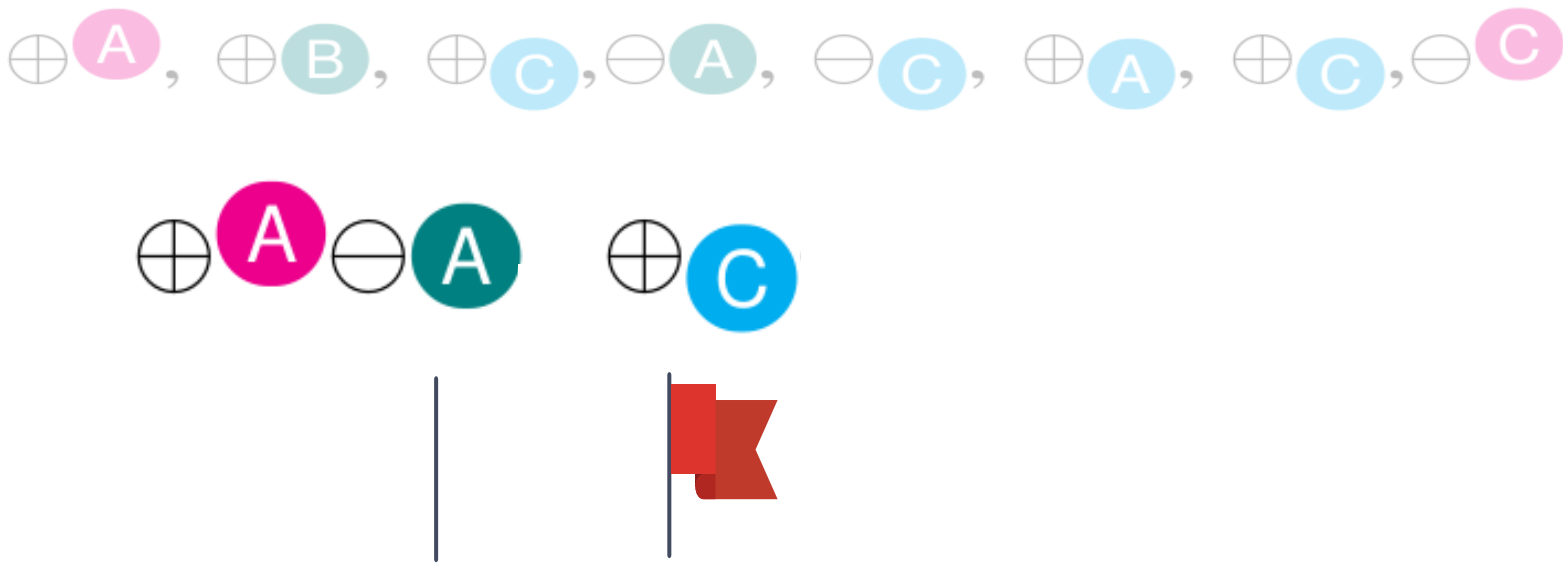
Update-side removing

Update-side removing

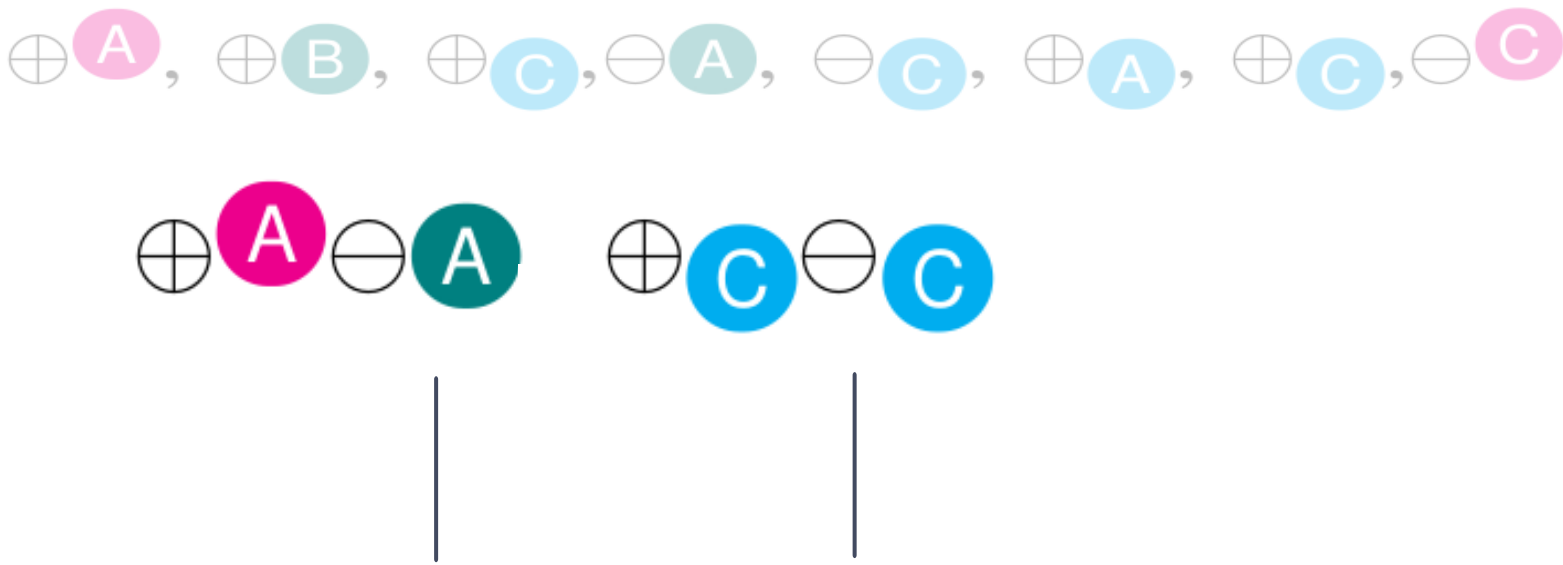
Update-side removing



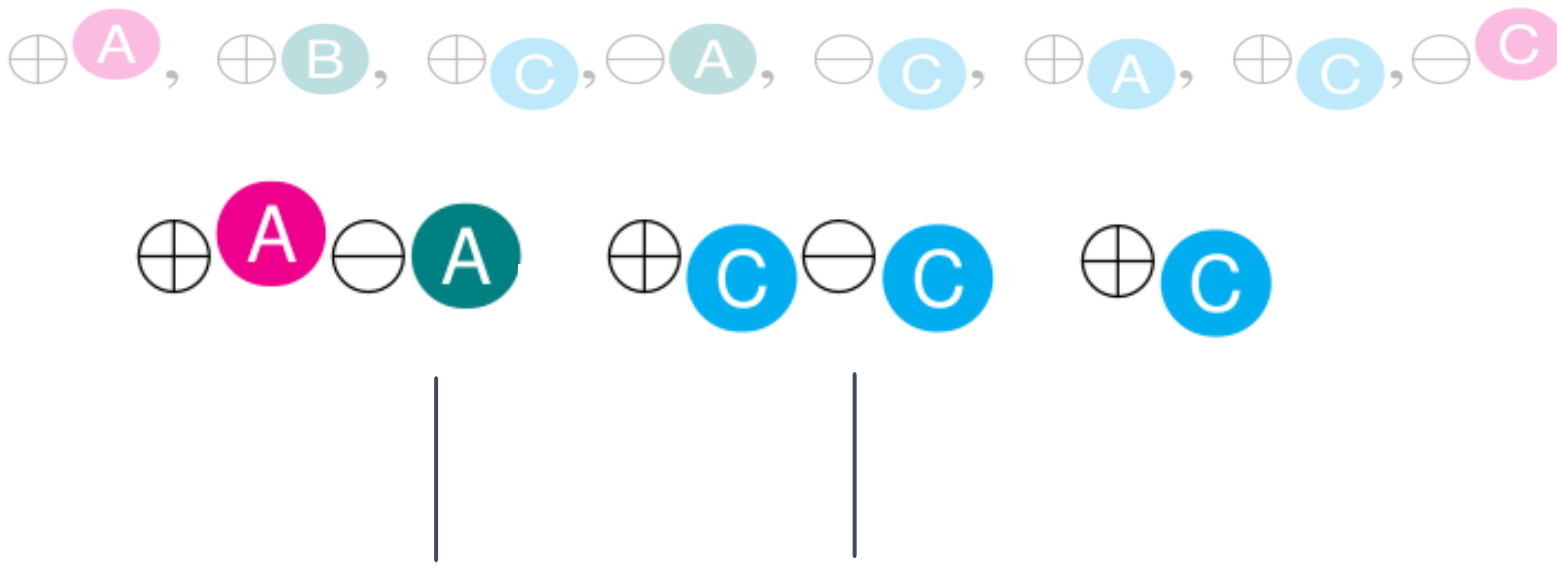
Update-side removing



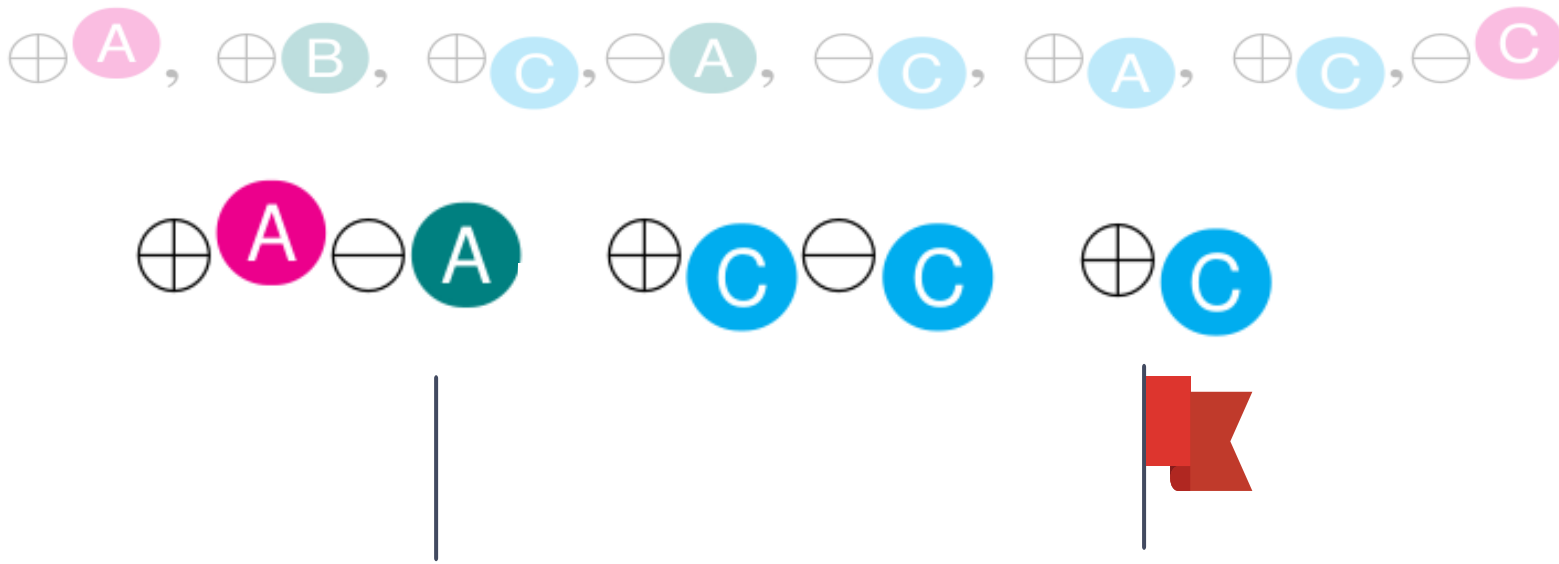
Update-side removing



Reusing garbage



Reusing garbage

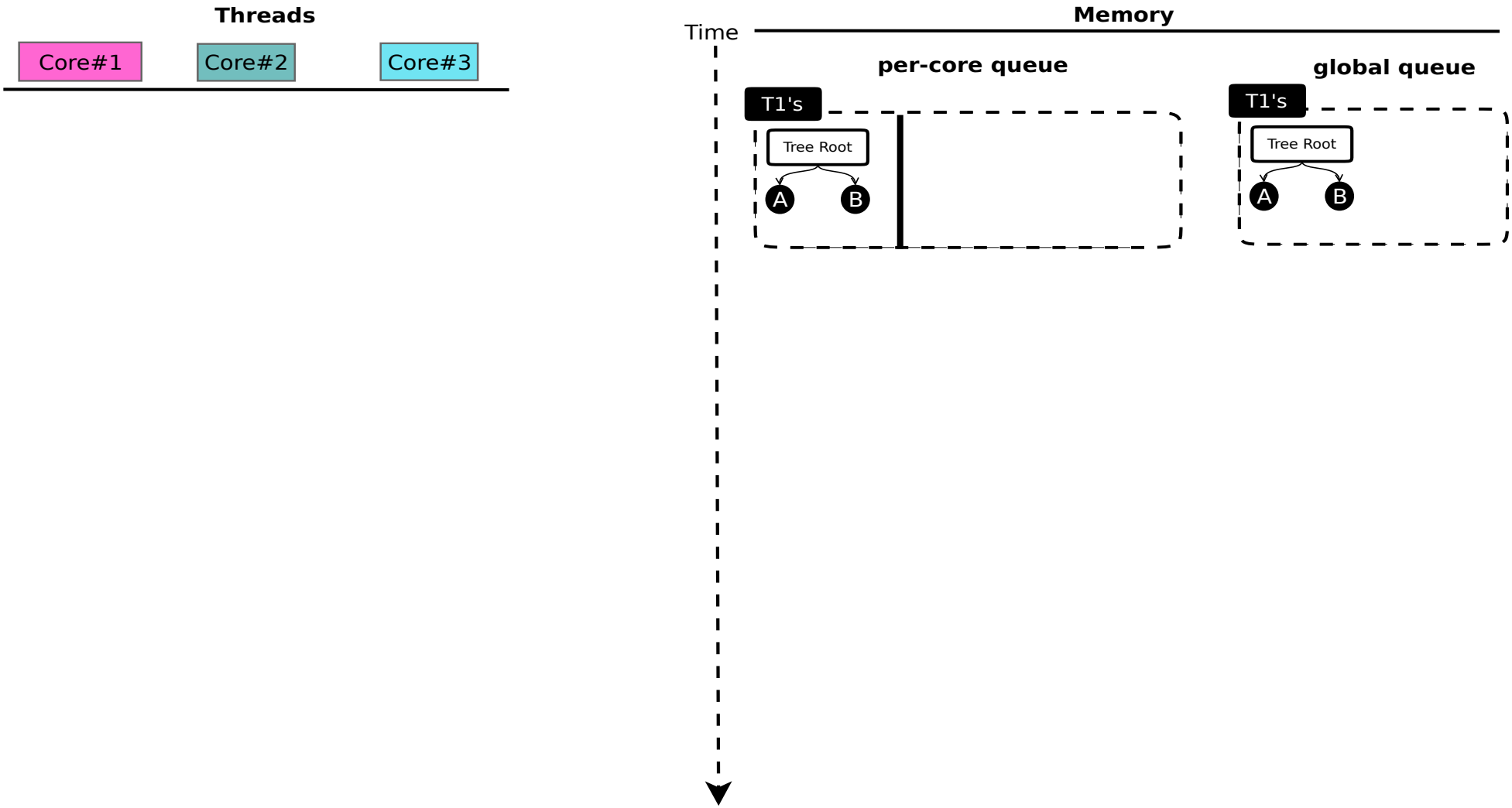


Additional approach

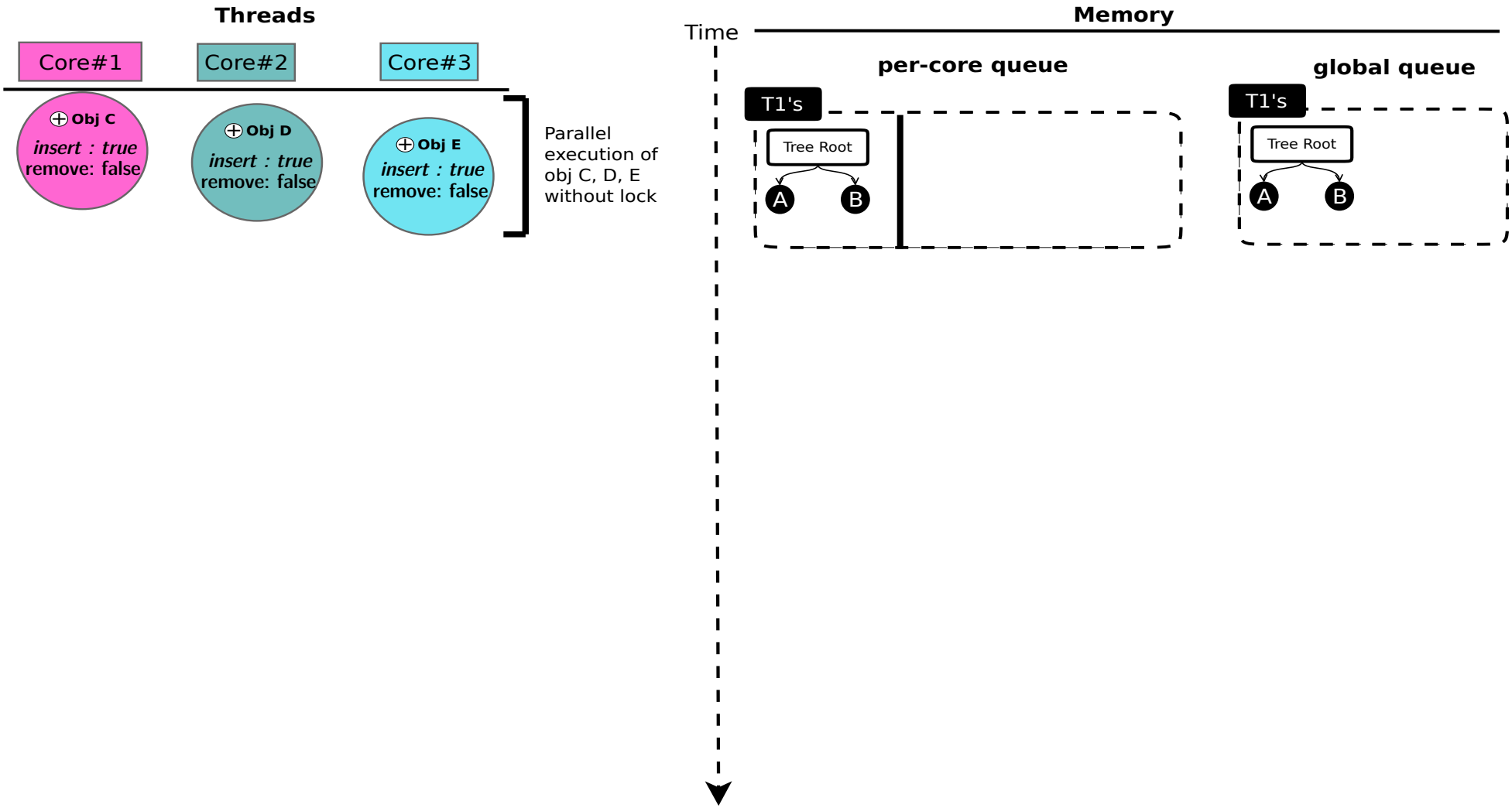
- Periodically applies the operation logs.
  - To reduce memory usage and to keep the log from growing without end.
  - Similar with the method of previous OpLog's batching updates and flat combining(FC)'s combiner thread.
- Use non-blocking queue.
  - Regardless of the per-core queue or the global queue.
  - Multiple producers and single consumer based non-blocking queue thereby reducing the CAS operations.

Example

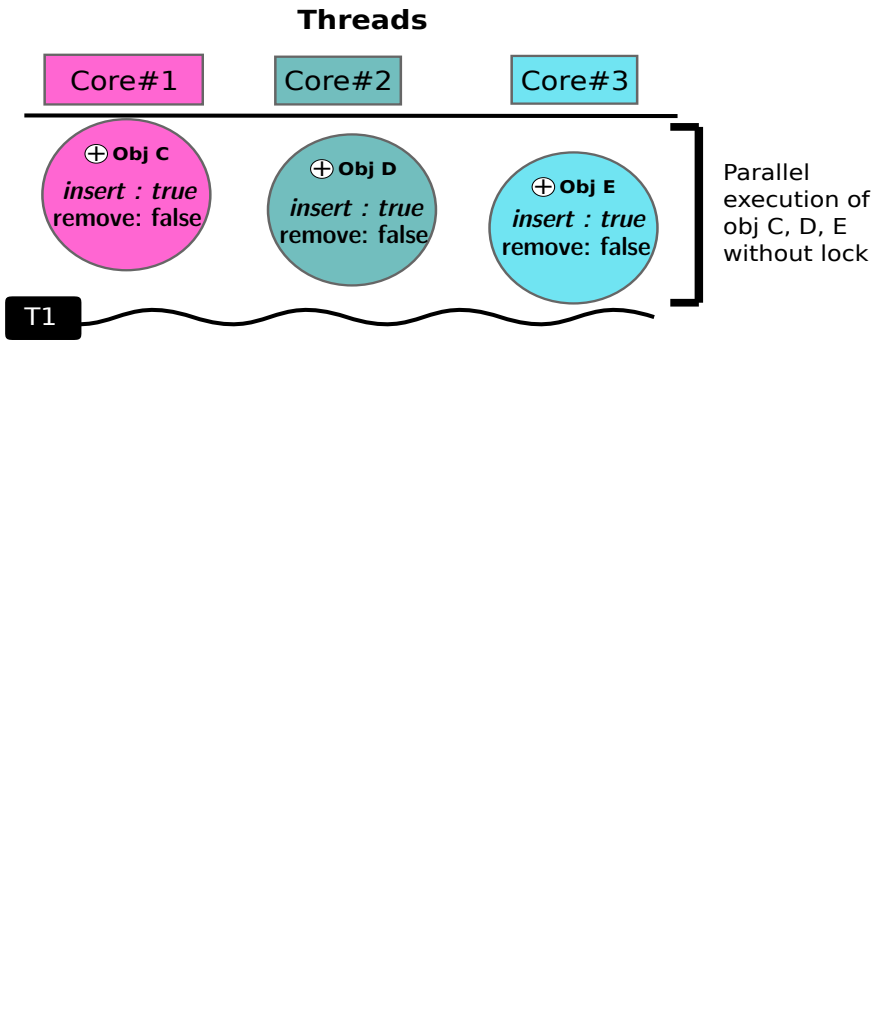
Example


Threads
Core#1
Core#2
Core#3
Time
Memory
per-core queue
global queue
T1's
Tree Root
A
B
T1's
Tree Root
A
B

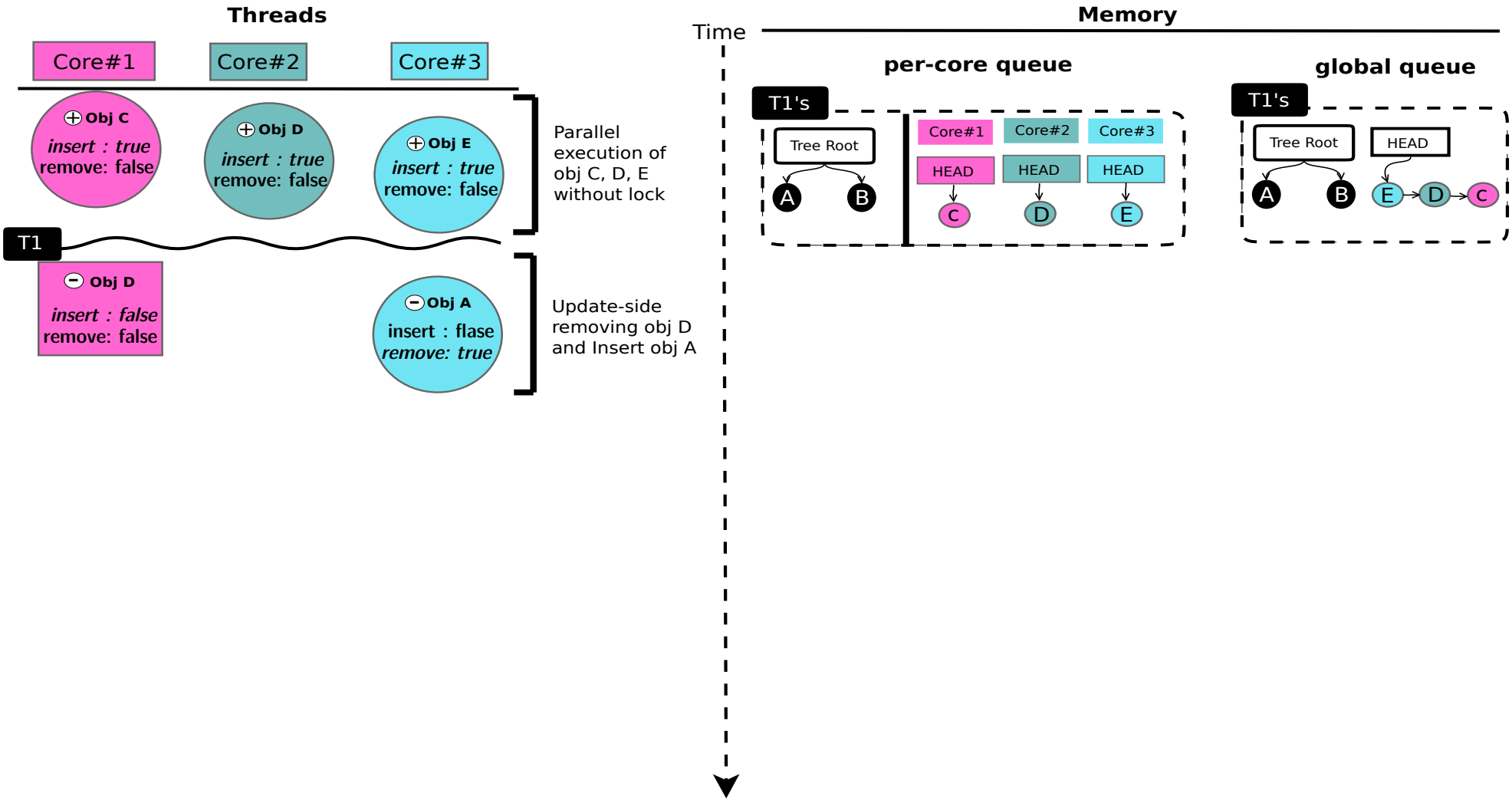
Example


Threads
Core#1
Core#2
Core#3
Obj C
insert : true
remove: false
Obj D
insert : true
remove: false
Obj E
insert : true
remove: false
Parallel execution of obj C, D, E without lock
Time
Memory
per-core queue
global queue
T1's
Tree Root
A
B
T1's
Tree Root
A
B

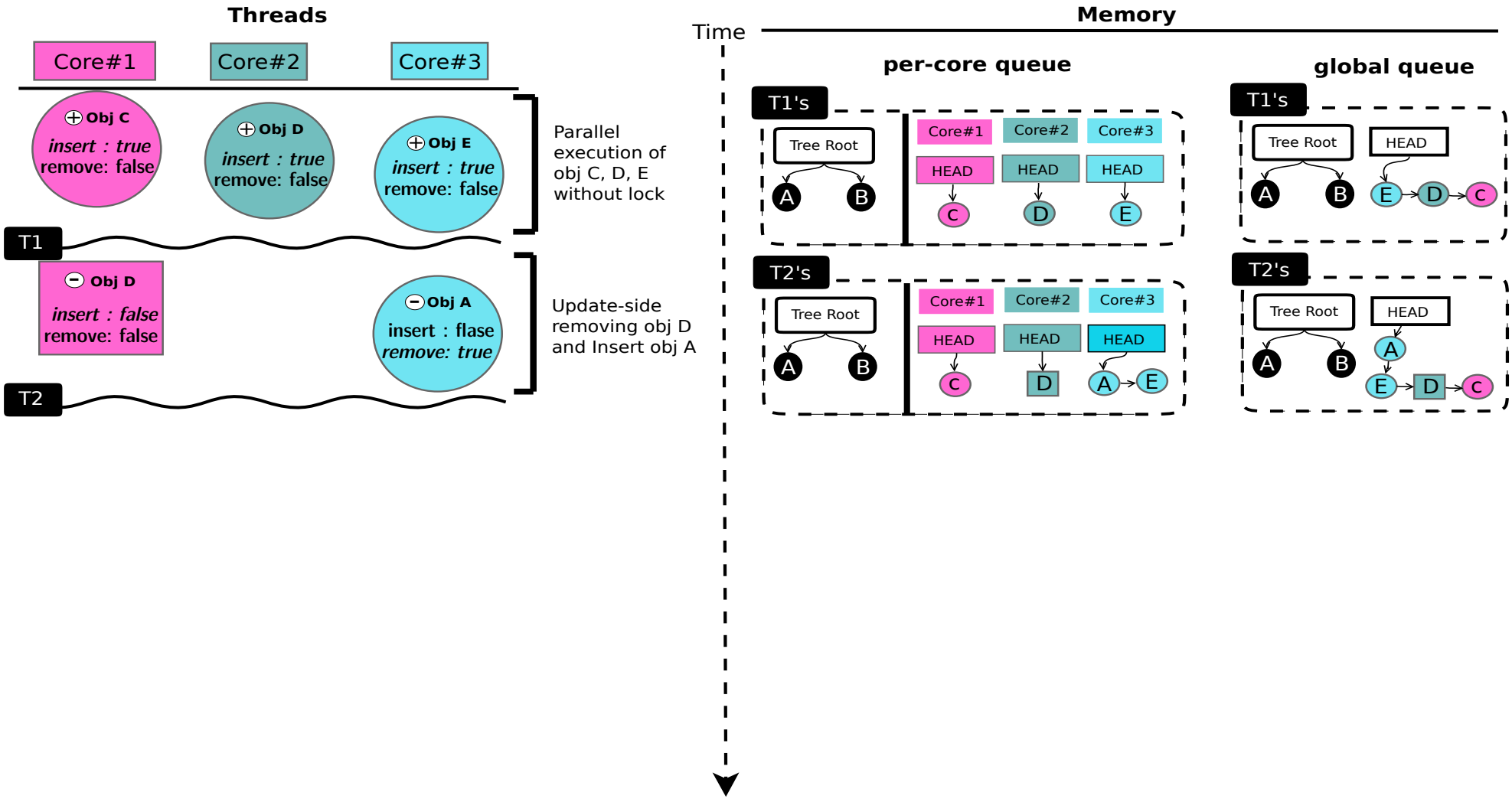
Example


Threads
Core#1
Core#2
Core#3
Obj C
insert : true
remove: false
Obj D
insert : true
remove: false
Obj E
insert : true
remove: false
Parallel execution of obj C, D, E without lock
T1
Time
Memory
per-core queue
T1's
Tree Root
A
B
Core#1
Core#2
Core#3
HEAD
HEAD
HEAD
C
D
E
global queue
T1's
Tree Root
HEAD
A
B
E
D
C

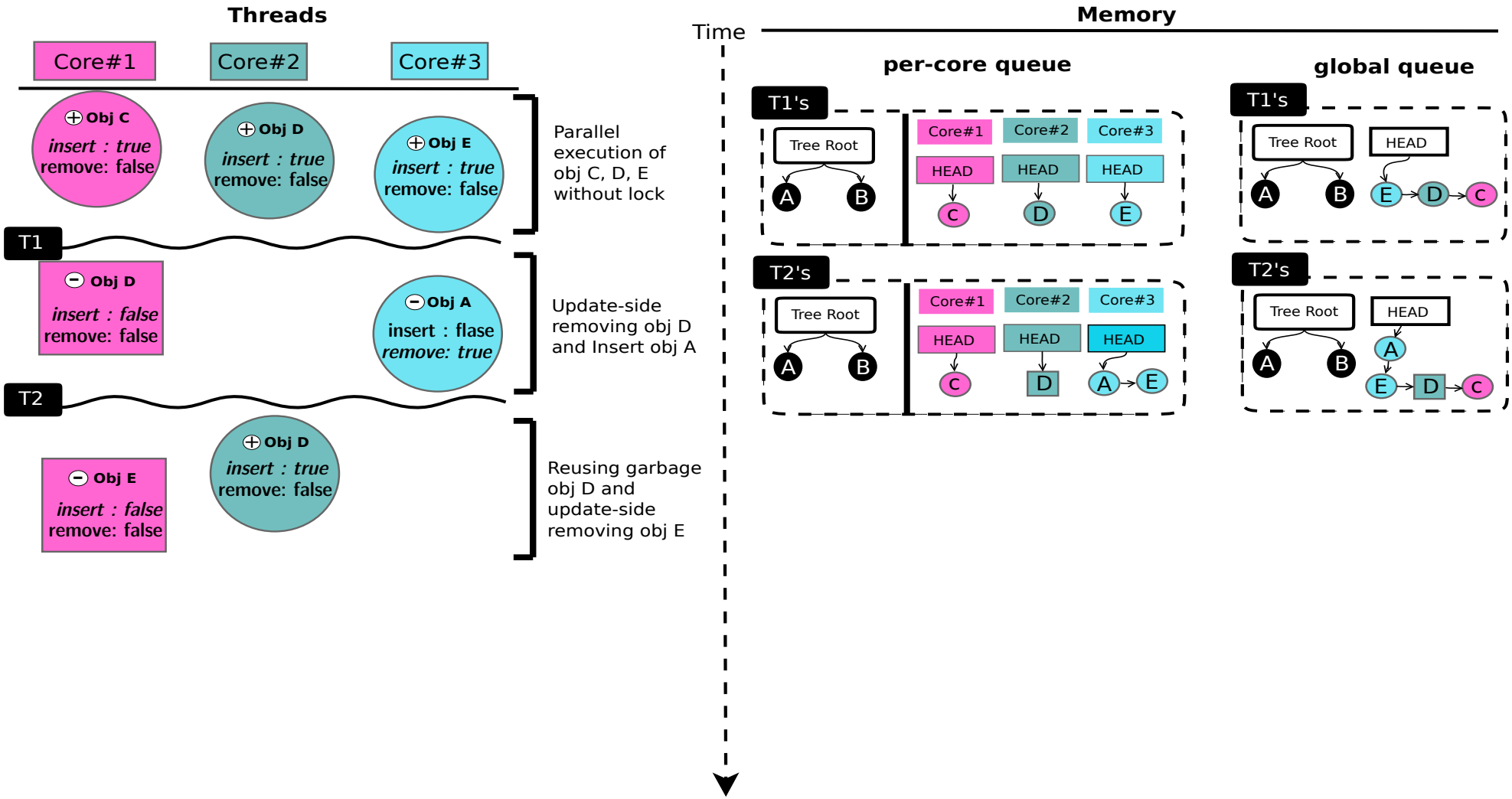
Example


Threads
Core#1
Core#2
Core#3
Obj C
insert : true
remove: false
Obj D
insert : true
remove: false
Obj E
insert : true
remove: false
Parallel execution of obj C, D, E without lock
T1
Obj D
insert : false
remove: false
Obj A
insert : flase
remove: true
Update-side removing obj D and Insert obj A
Time
Memory
per-core queue
T1's
Tree Root
A
B
Core#1
Core#2
Core#3
HEAD
HEAD
HEAD
C
D
E
global queue
T1's
Tree Root
HEAD
A
B
E
D
C

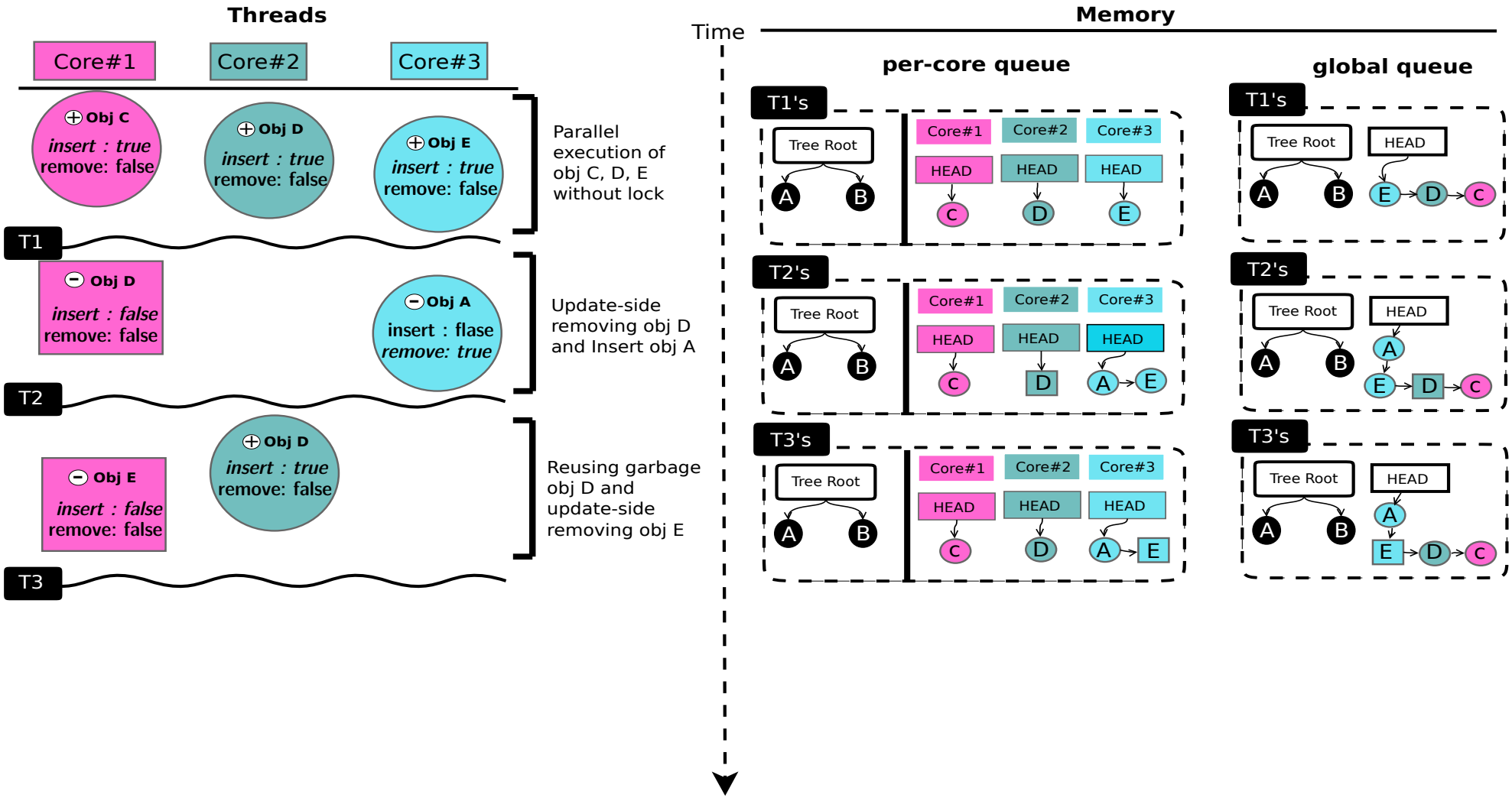
Example


Threads
Core#1
Core#2
Core#3
Obj C
insert : true
remove: false
Obj D
insert : true
remove: false
Obj E
insert : true
remove: false
Parallel execution of obj C, D, E without lock
T1
Obj D
insert : false
remove: false
Obj A
insert : flase
remove: true
Update-side removing obj D and Insert obj A
T2
Time
Memory
per-core queue
global queue
T1's
T2's
Tree Root
HEAD
A
B
C
D
E

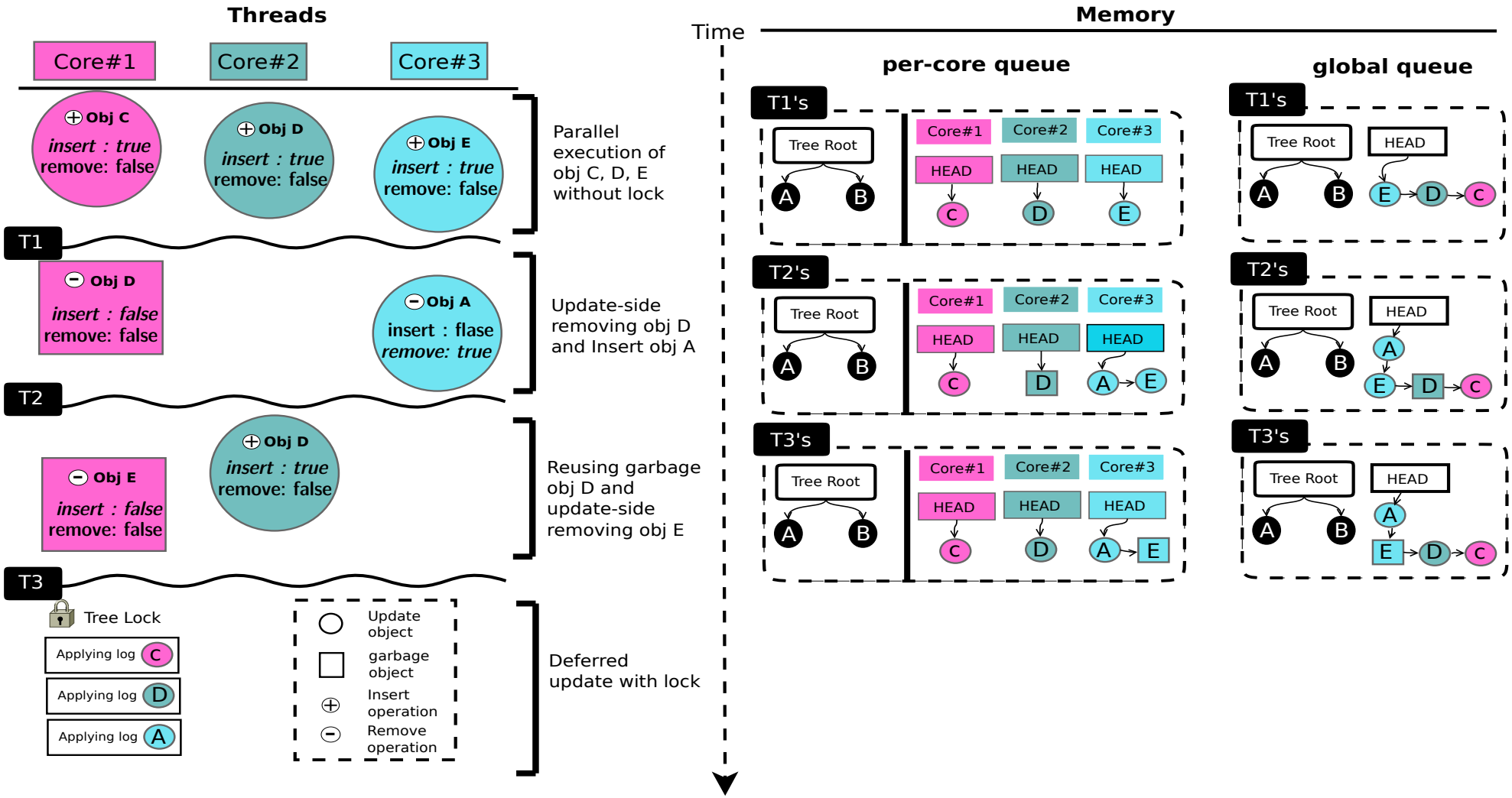
Example


Threads
Core#1
Core#2
Core#3
Time
Memory
per-core queue
global queue
Obj C
insert : true
remove: false
Obj D
insert : true
remove: false
Obj E
insert : true
remove: false
Parallel execution of obj C, D, E without lock
T1
Obj D
insert : false
remove: false
Obj A
insert : flase
remove: true
Update-side removing obj D and Insert obj A
T2
Obj D
insert : true
remove: false
Obj E
insert : false
remove: false
Reusing garbage obj D and update-side removing obj E
T1's
T2's
Tree Root
HEAD
A
B
C
D
E

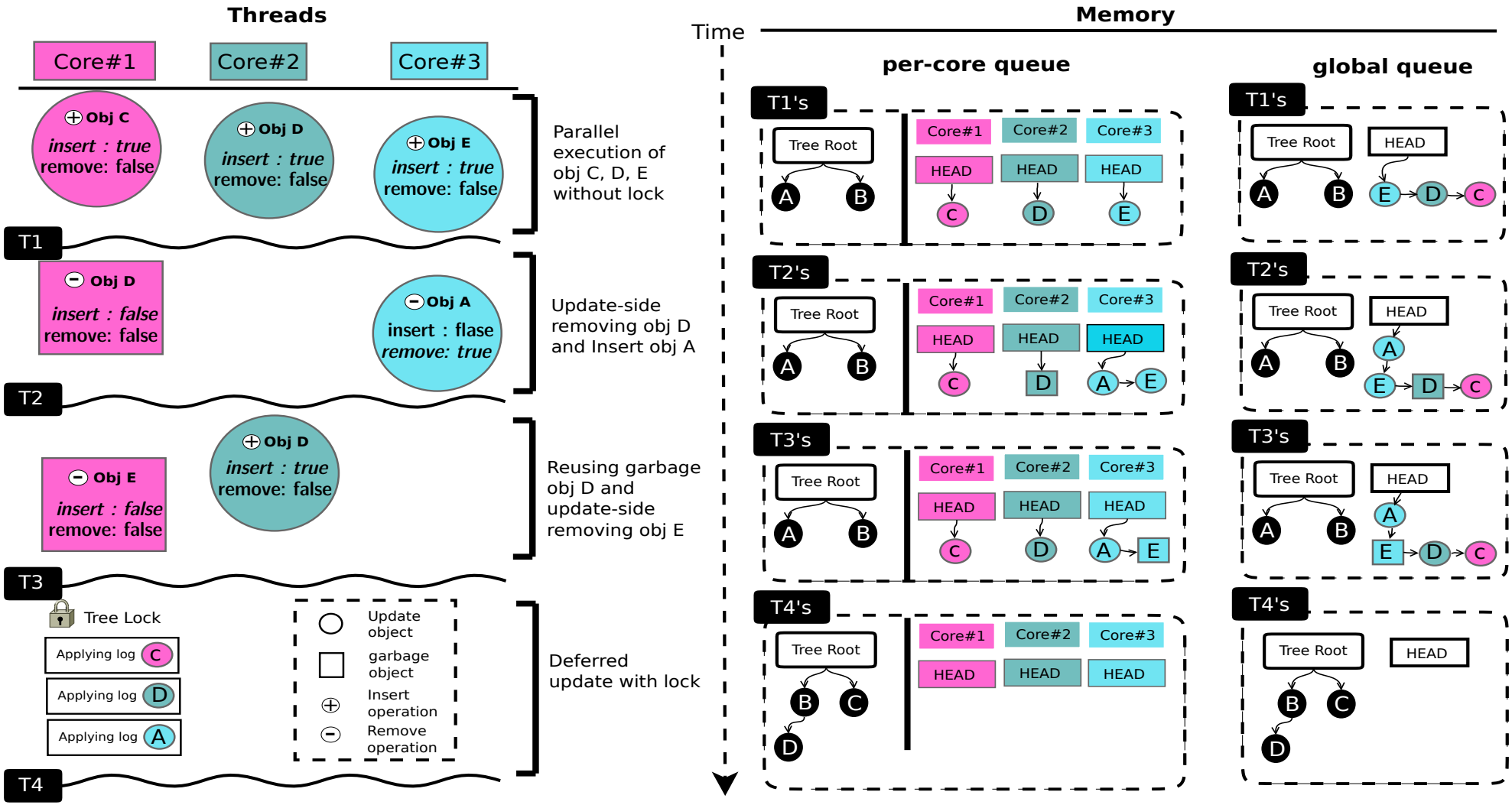
Example


Threads
Core#1
Core#2
Core#3
Obj C
insert : true
remove: false
Obj D
insert : true
remove: false
Obj E
insert : true
remove: false
Parallel execution of obj C, D, E without lock
T1
Obj D
insert : false
remove: false
Obj A
insert : flase
remove: true
Update-side removing obj D and Insert obj A
T2
Obj D
insert : true
remove: false
Obj E
insert : false
remove: false
Reusing garbage obj D and update-side removing obj E
T3
Time
Memory
per-core queue
global queue
T1's
T2's
T3's
Tree Root
HEAD
A
B
C
D
E

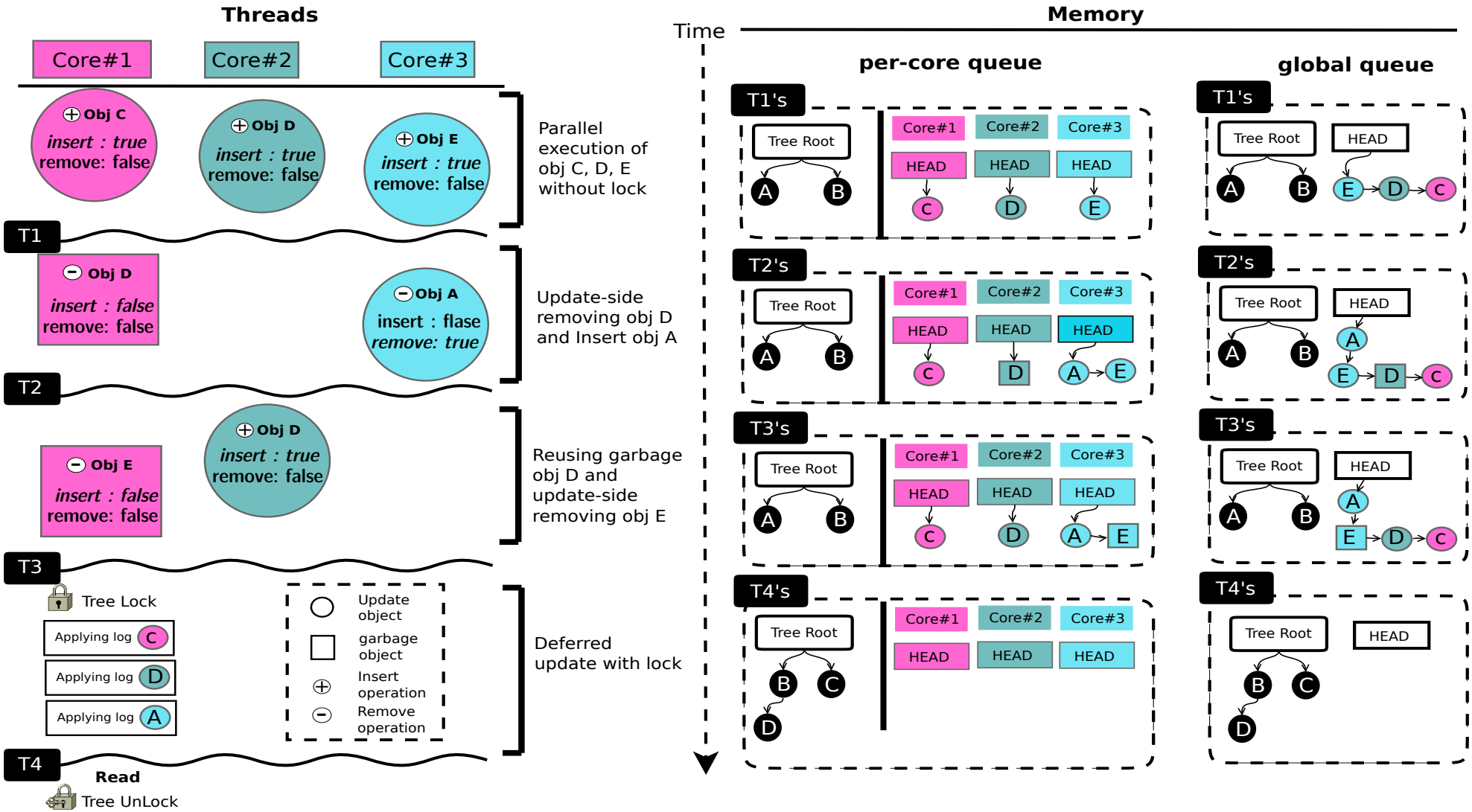
Example


Threads
Core#1
Core#2
Core#3
Time
Memory
per-core queue
global queue
Obj C
insert : true
remove: false
Obj D
insert : true
remove: false
Obj E
insert : true
remove: false
Parallel execution of obj C, D, E without lock
T1
Obj D
insert : false
remove: false
Obj A
insert : flase
remove: true
Update-side removing obj D and Insert obj A
T2
Obj D
insert : true
remove: false
Obj E
insert : false
remove: false
Reusing garbage obj D and update-side removing obj E
T3
Tree Lock
Applying log C
Applying log D
Applying log A
Update object
garbage object
Insert operation
Remove operation
Deferred update with lock
T1's
T2's
T3's
Tree Root
HEAD
A
B
C
D
E

Example


Threads
Core#1
Core#2
Core#3
Obj C
insert : true
remove: false
Obj D
insert : true
remove: false
Obj E
insert : true
remove: false
Parallel execution of obj C, D, E without lock
T1
Obj D
insert : false
remove: false
Obj A
insert : flase
remove: true
Update-side removing obj D and Insert obj A
T2
Obj D
insert : true
remove: false
Obj E
insert : false
remove: false
Reusing garbage obj D and update-side removing obj E
T3
Tree Lock
Applying log C
Applying log D
Applying log A
Update object
garbage object
Insert operation
Remove operation
Deferred update with lock
T4
Time
Memory
per-core queue
global queue
T1's
T2's
T3's
T4's
Tree Root
HEAD
A
B
C
D
E

Example


Threads
Core#1
Core#2
Core#3
Obj C
insert : true
remove: false
Obj D
insert : true
remove: false
Obj E
insert : true
remove: false
Parallel execution of obj C, D, E without lock
T1
Obj D
insert : false
remove: false
Obj A
insert : flase
remove: true
Update-side removing obj D and Insert obj A
T2
Obj D
insert : true
remove: false
Obj E
insert : false
remove: false
Reusing garbage obj D and update-side removing obj E
T3
Tree Lock
Applying log C
Applying log D
Applying log A
Update object
garbage object
Insert operation
Remove operation
Deferred update with lock
T4
Read
Tree UnLock
Time
Memory
per-core queue
global queue
T1's
T2's
T3's
T4's
Tree Root
HEAD
A
B
C
D
E

Applying the Linux kernel

- The Linux reverse page mapping(rmap).
- Kernel memory management mechanism.
- Consists of anonymous rmap and file rmap.
- Update-heavy data structure.

Applying the Linux kernel

- The Linux reverse page mapping(rmap).
- Kernel memory management mechanism.
- Consists of anonymous rmap and file rmap.
- Update-heavy data structure.

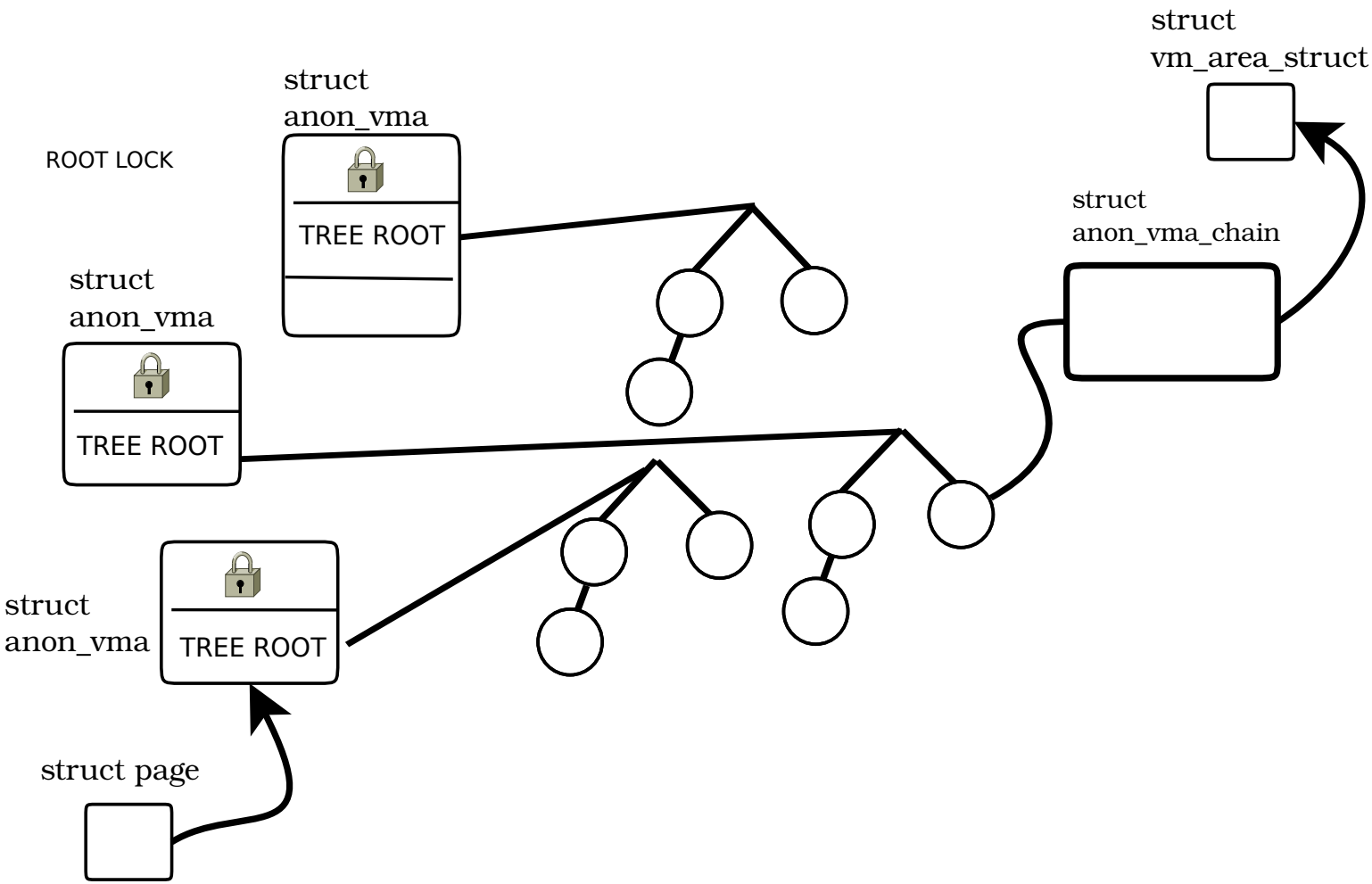
Applying the Linux kernel

- The Linux reverse page mapping(rmap).
- Kernel memory management mechanism.
- Consists of anonymous rmap and file rmap.
- Update-heavy data structure.

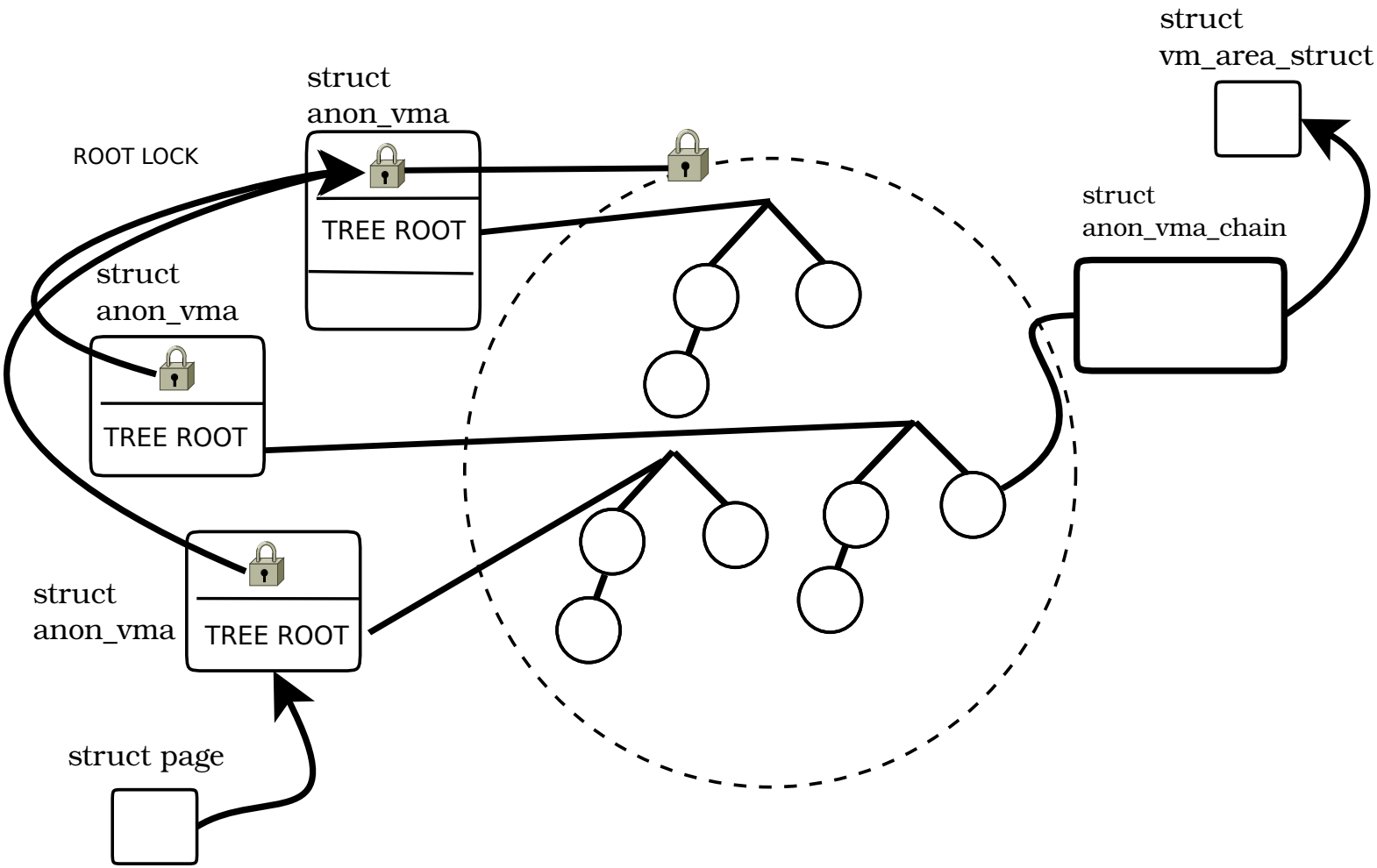
Applying the Linux kernel

- The Linux reverse page mapping(rmap).
- Kernel memory management mechanism.
- Consists of anonymous rmap and file rmap.
- Update-heavy data structure.

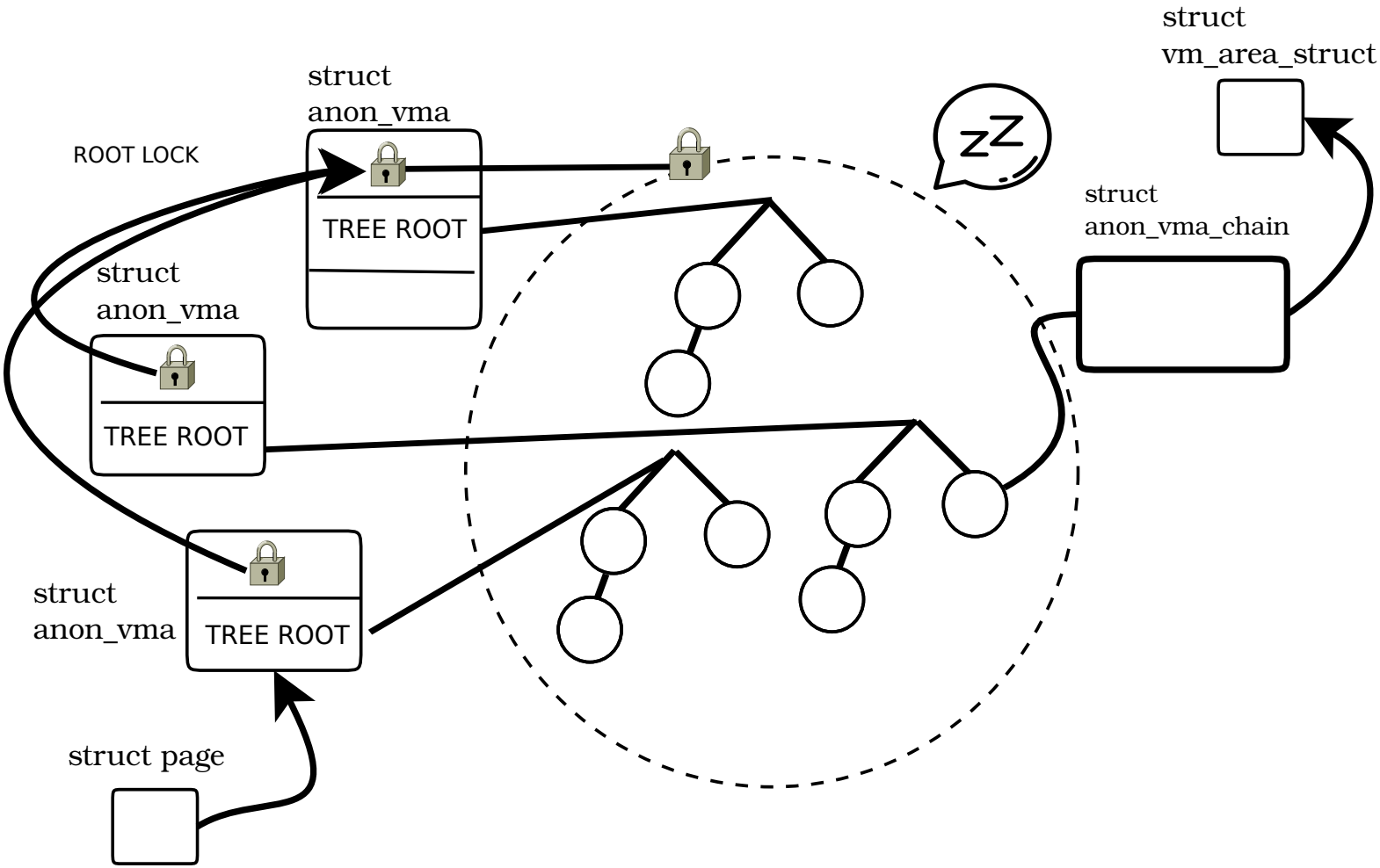
Anonymous reverse mapping



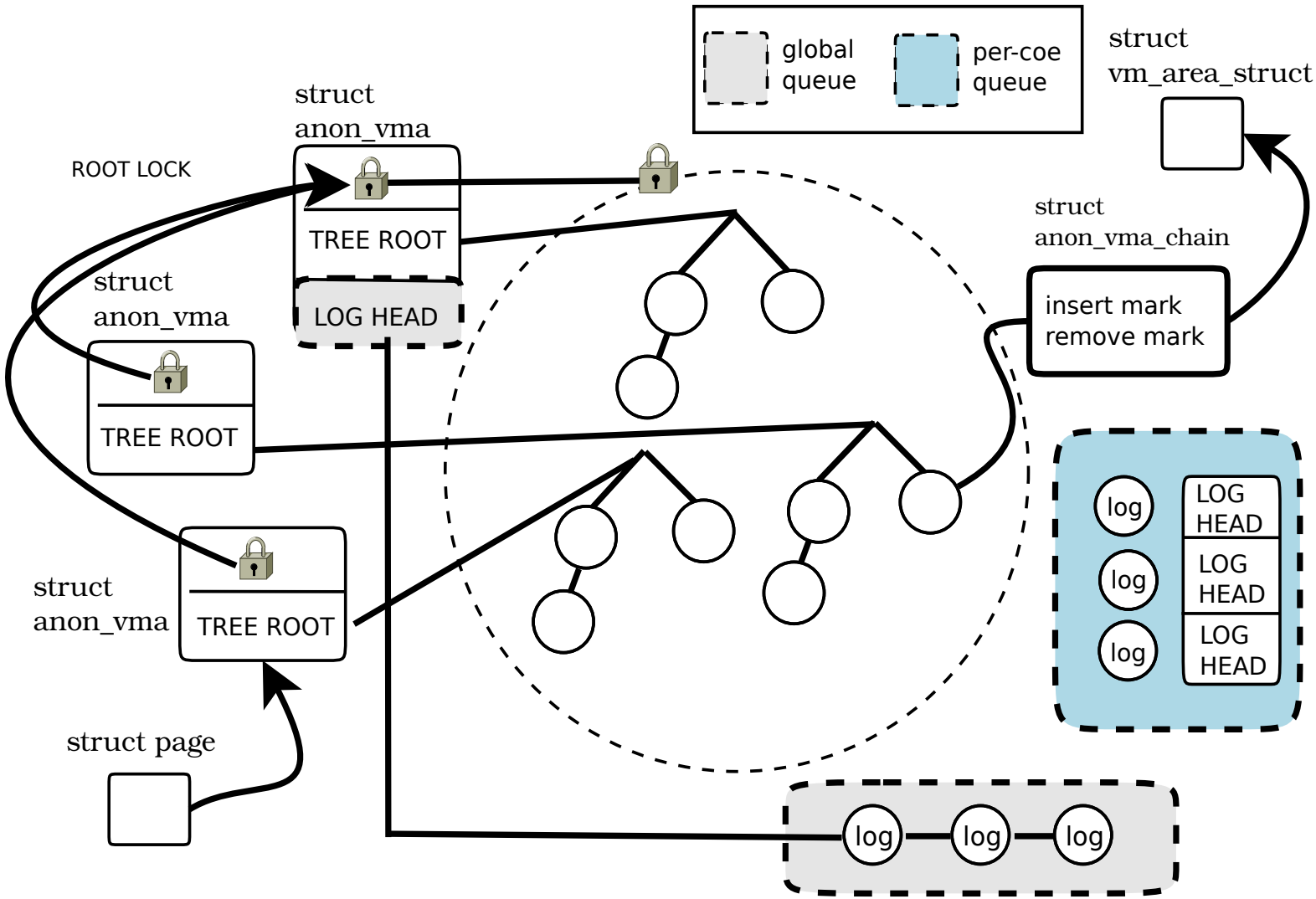
Anonymous reverse mapping



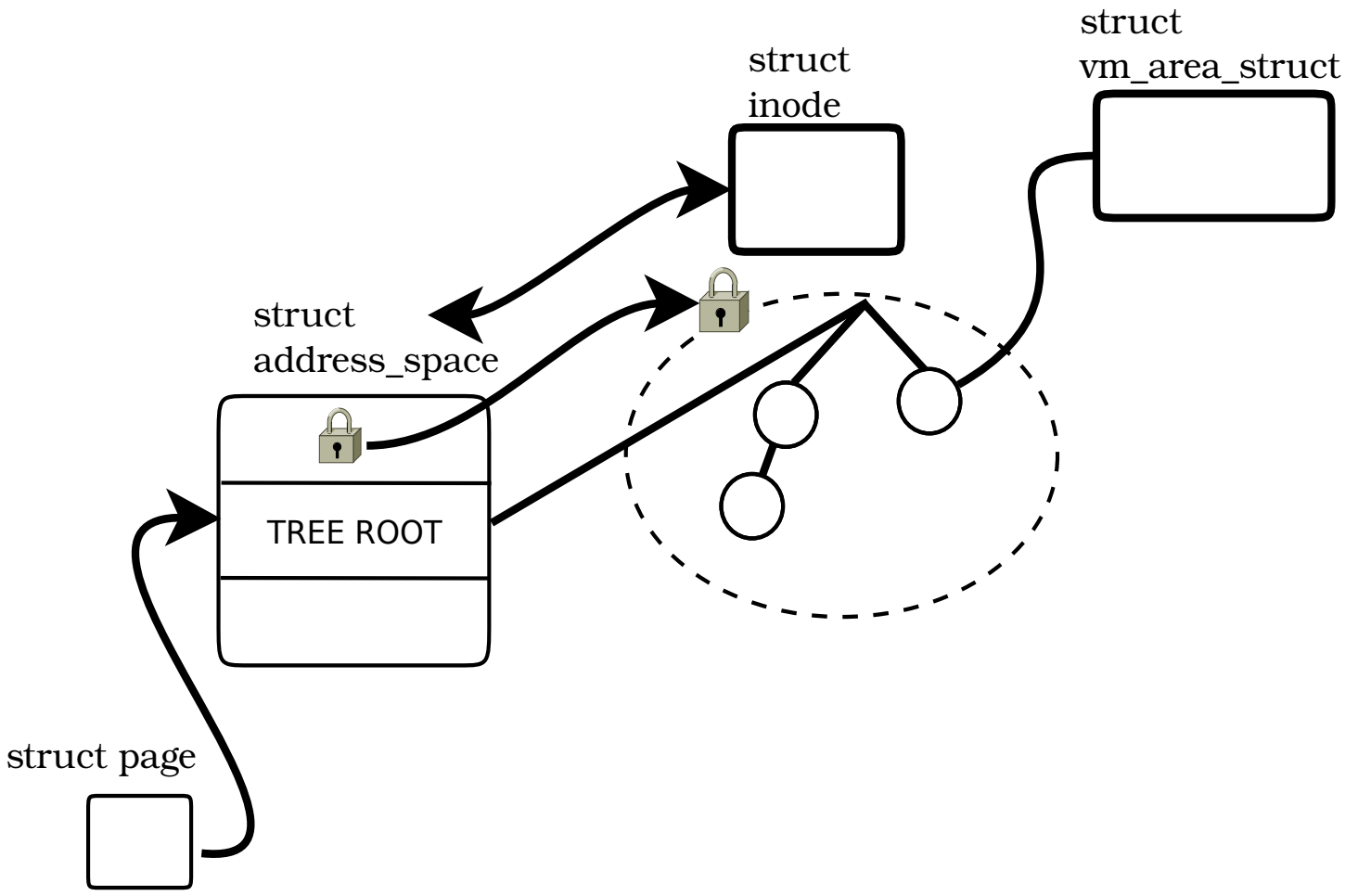
Anonymous reverse mapping



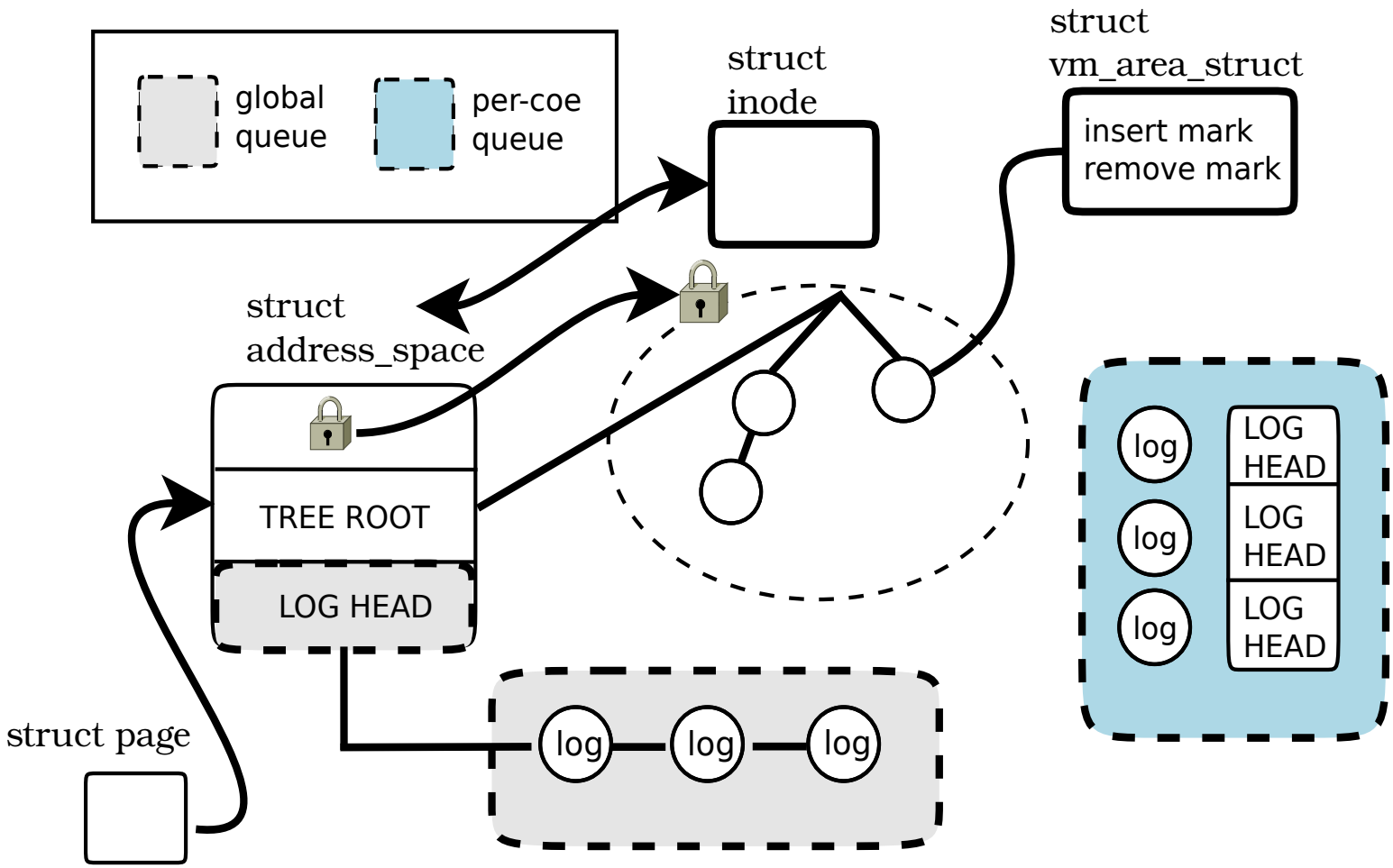
Anonymous reverse mapping



File mapping



File mapping



Evaluation

- We used four different experiment settings.
- 1. stock Linux
- 2. LDU + global queue
- 3. LDU + per-core queue
- 4. Harris linked list

Evaluation

- We used four different experiment settings.
- 1. stock Linux
- 2. LDU + global queue
- 3. LDU + per-core queue
- 4. Harris linked list

Evaluation

- We used four different experiment settings.
- 1. stock Linux
- 2. LDU + global queue
- 3. LDU + per-core queue
- 4. Harris linked list

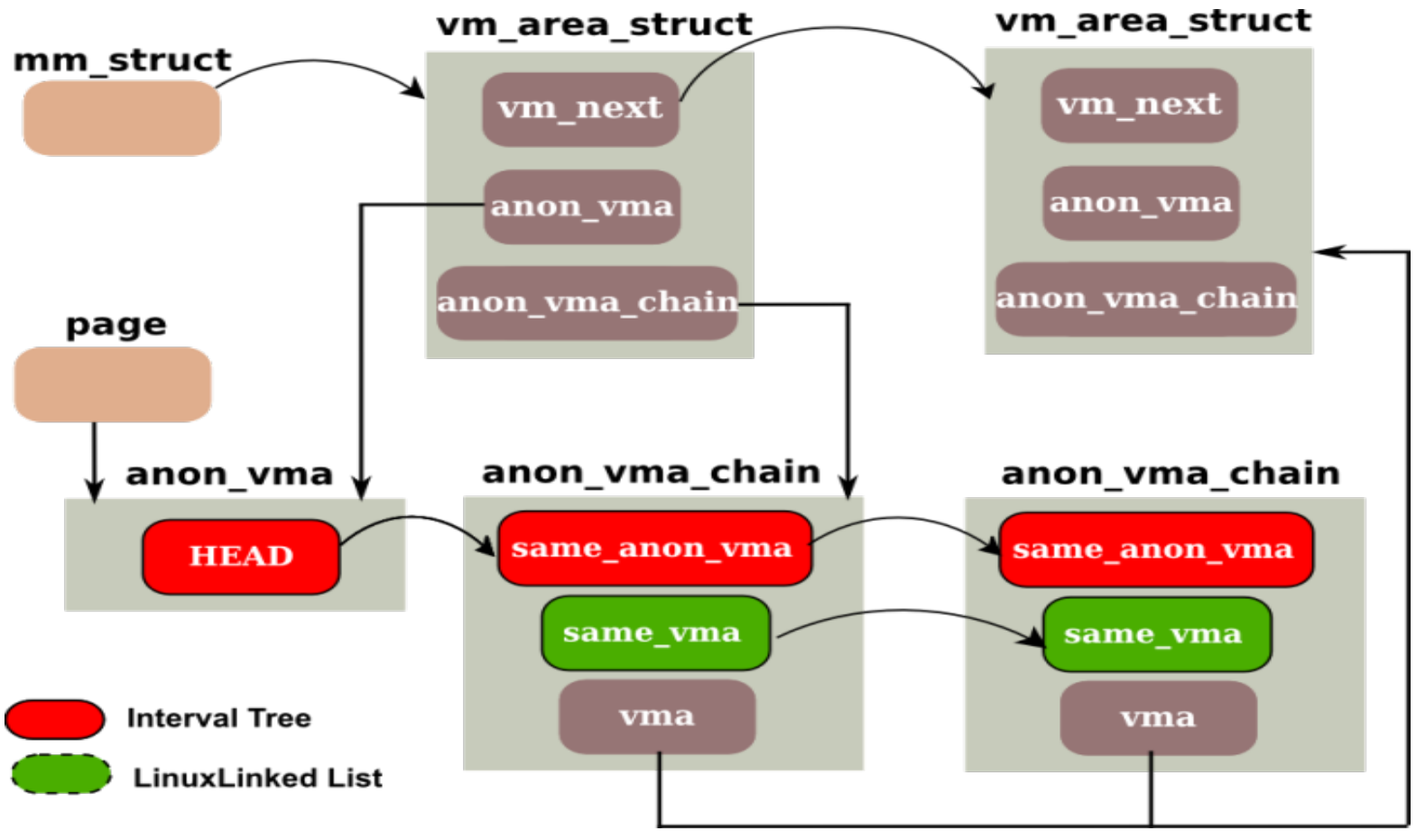
Evaluation

- We used four different experiment settings.
- 1. stock Linux
- 2. LDU + global queue
- 3. LDU + per-core queue
- 4. Harris linked list

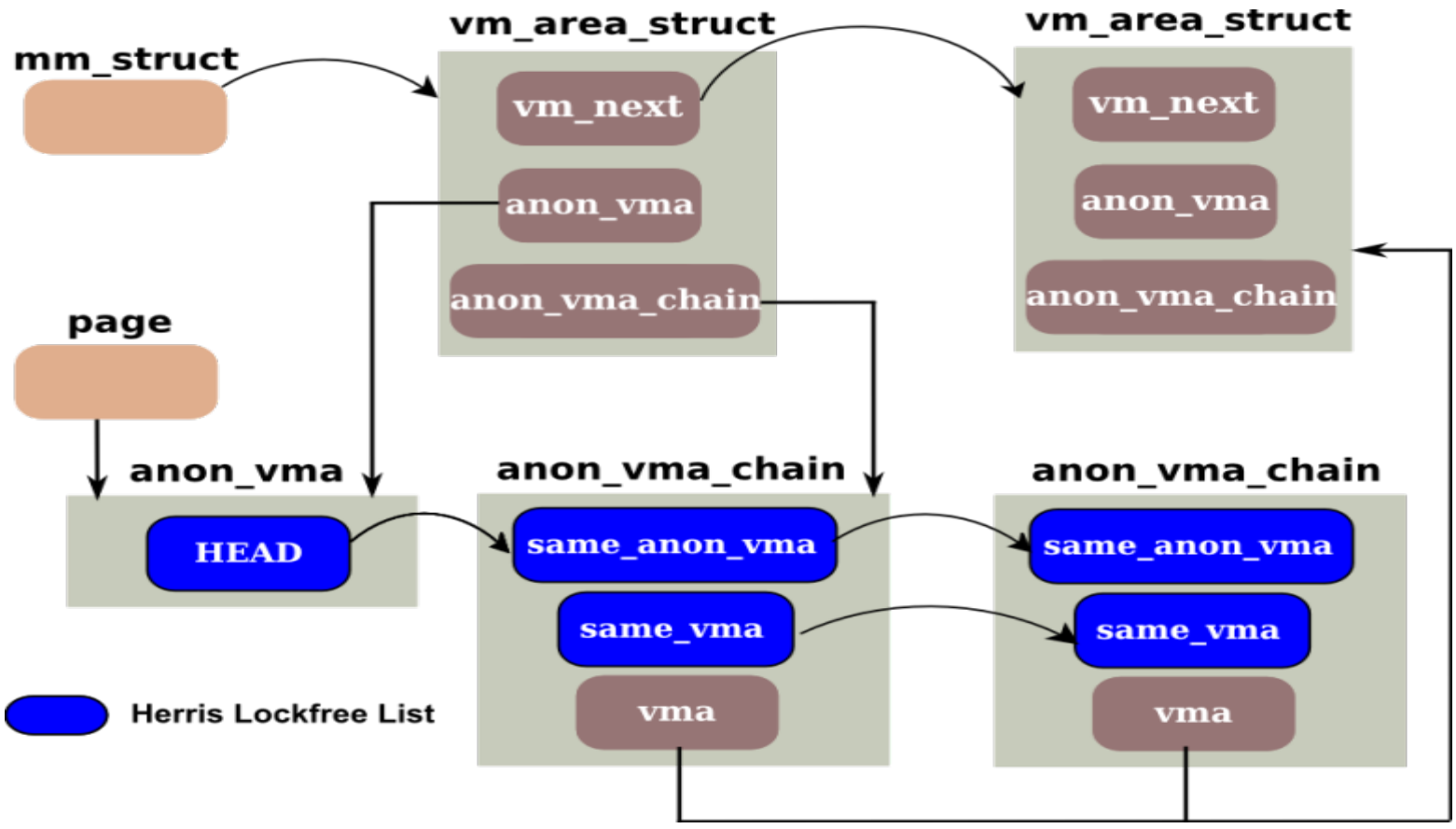
Evaluation

- We used four different experiment settings.
- 1. stock Linux
- 2. LDU + global queue
- 3. LDU + per-core queue
- 4. Harris linked list

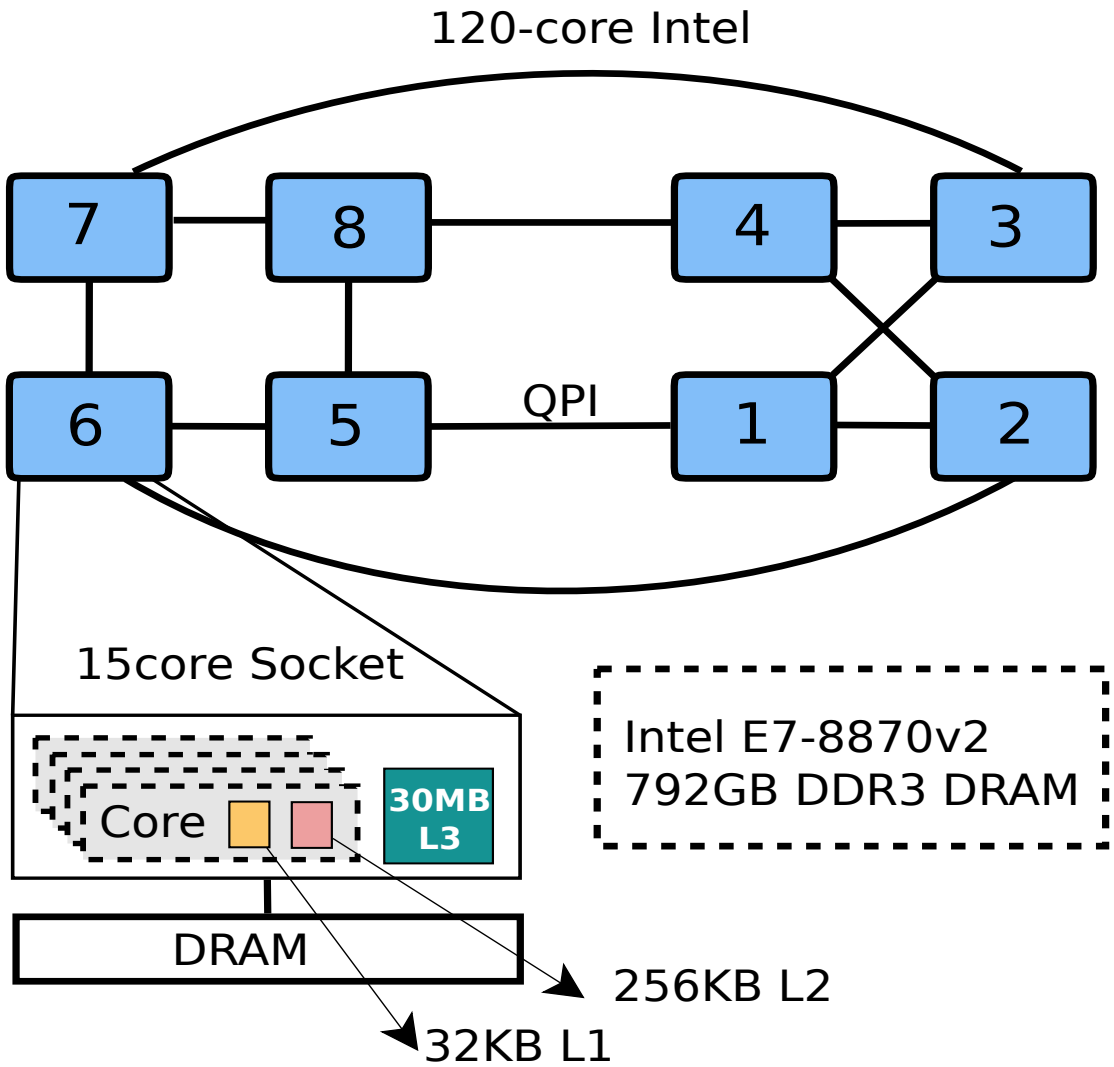
Non-blocking algorithm – Harris linked list



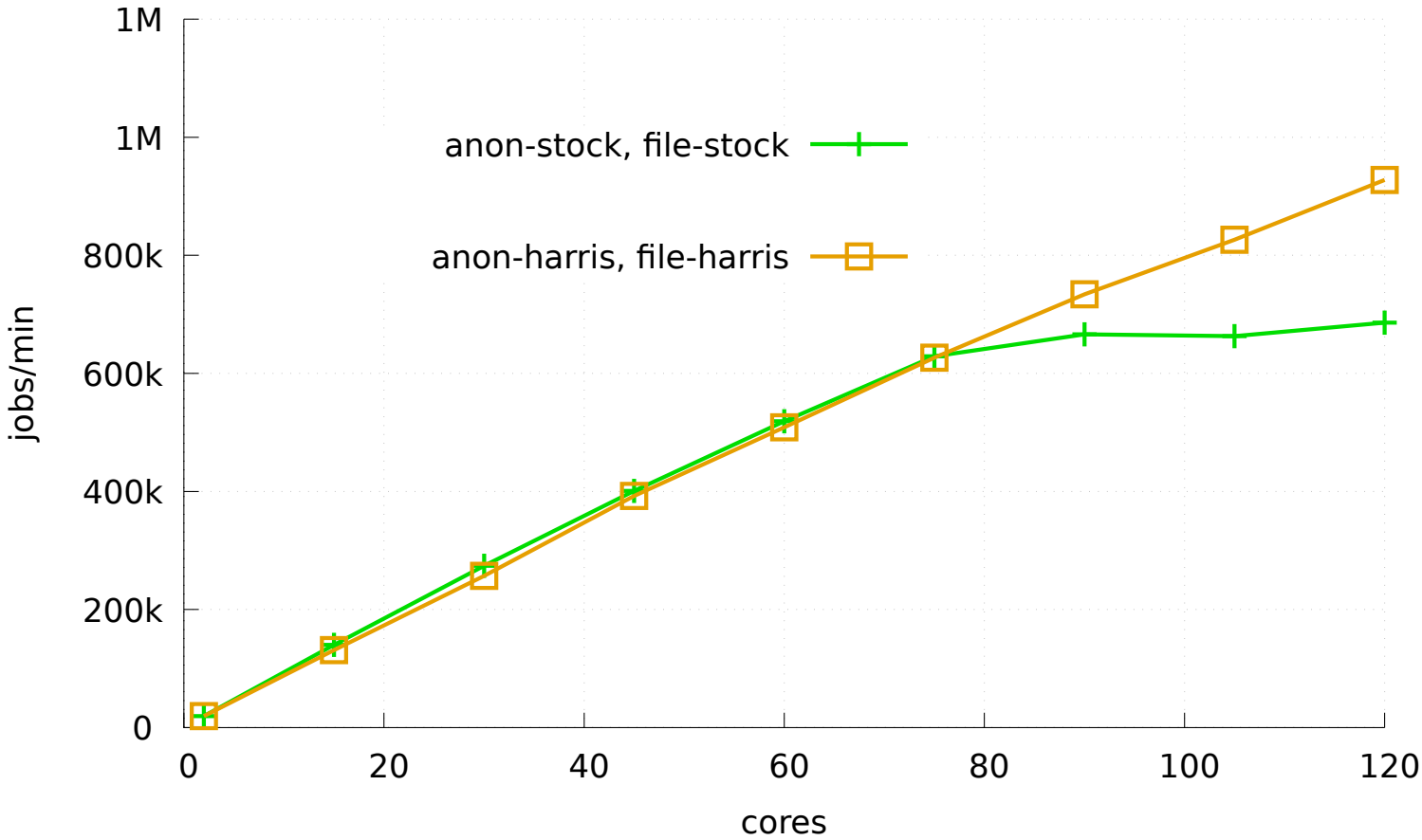
Non-blocking algorithm – Harris linked list



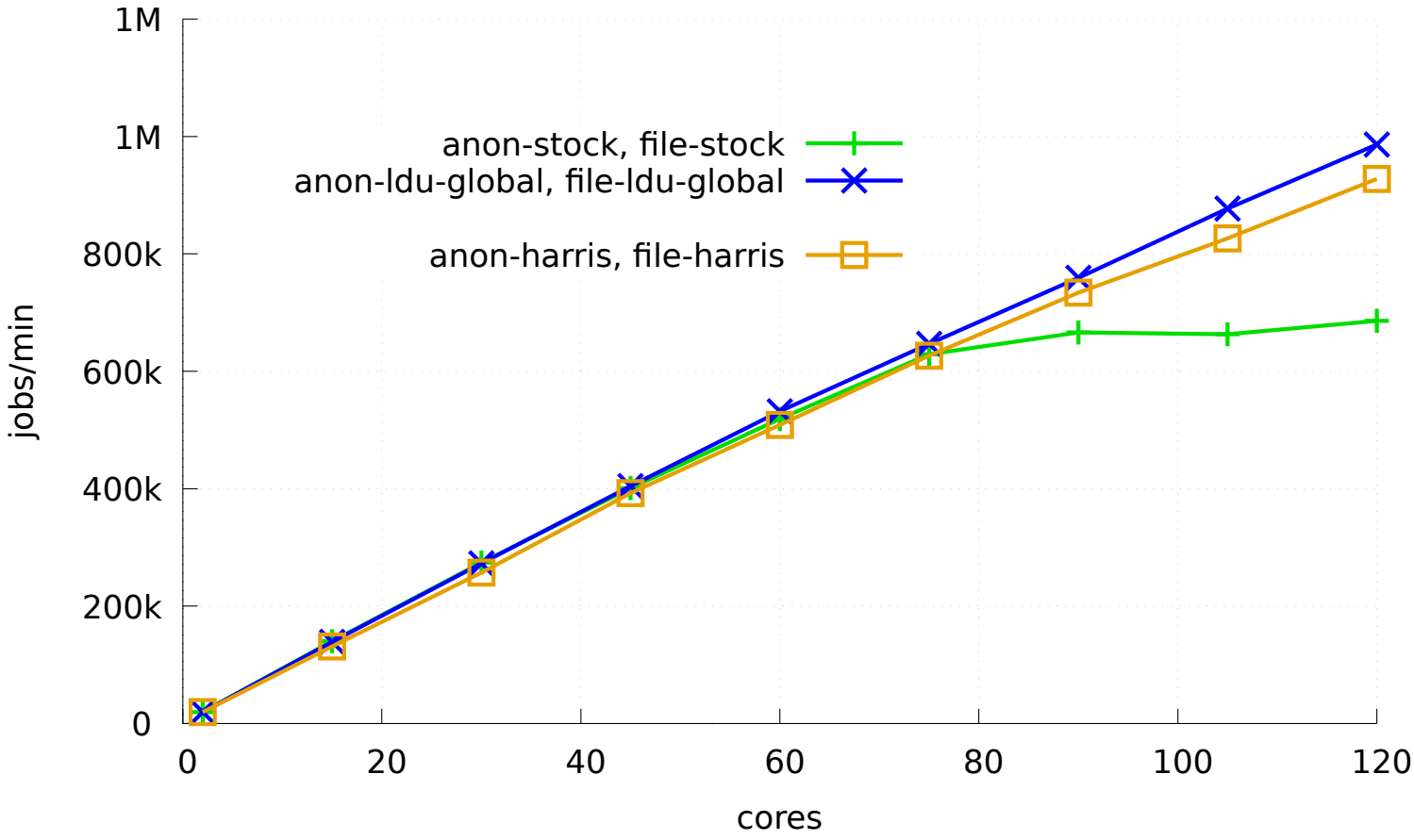
Evaluation : Hardware



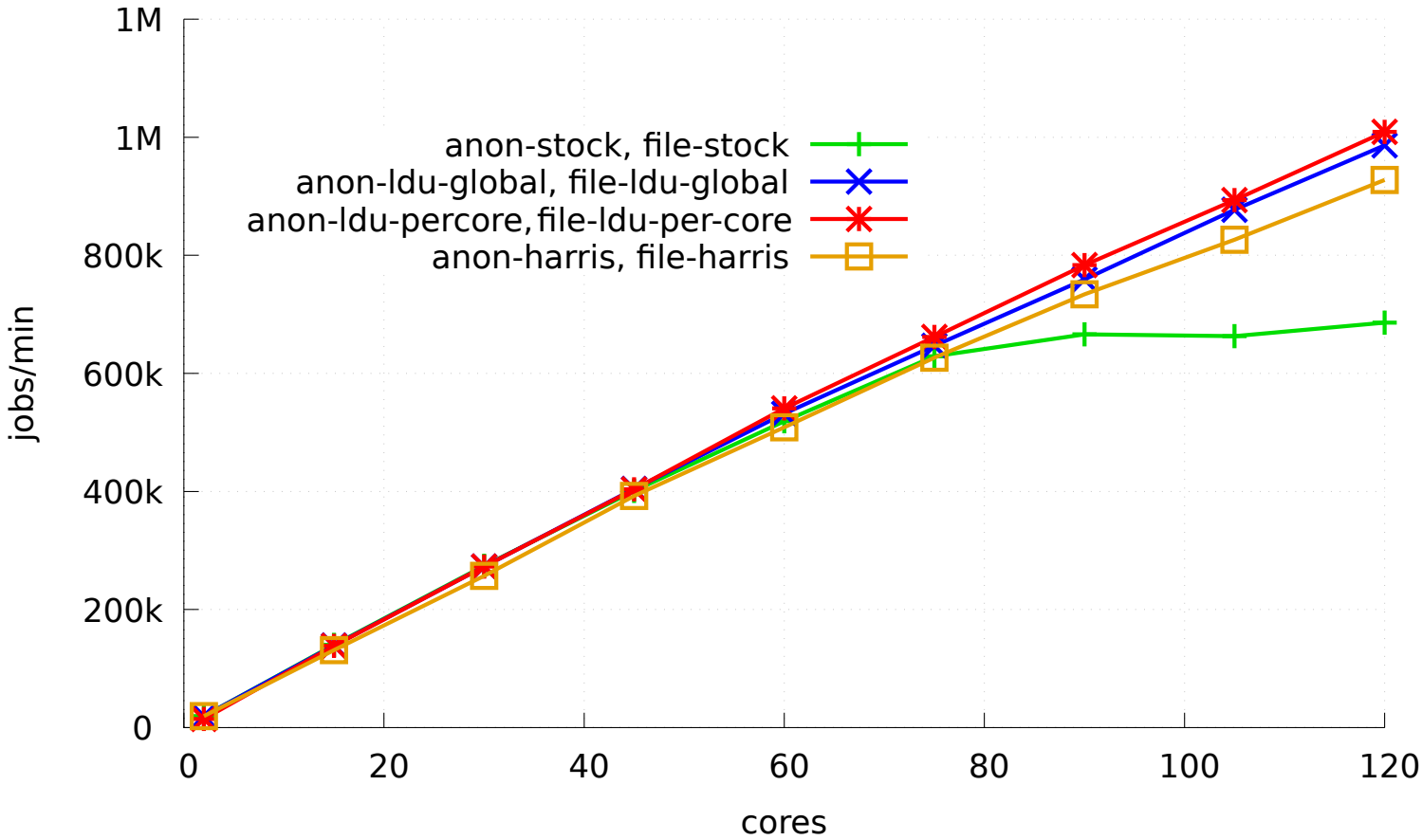
AIM7


1M
1M
800k
600k
400k
200k
0
jobs/min
0
20
40
60
80
100
120
cores
anon-stock, file-stock
anon-harris, file-harris

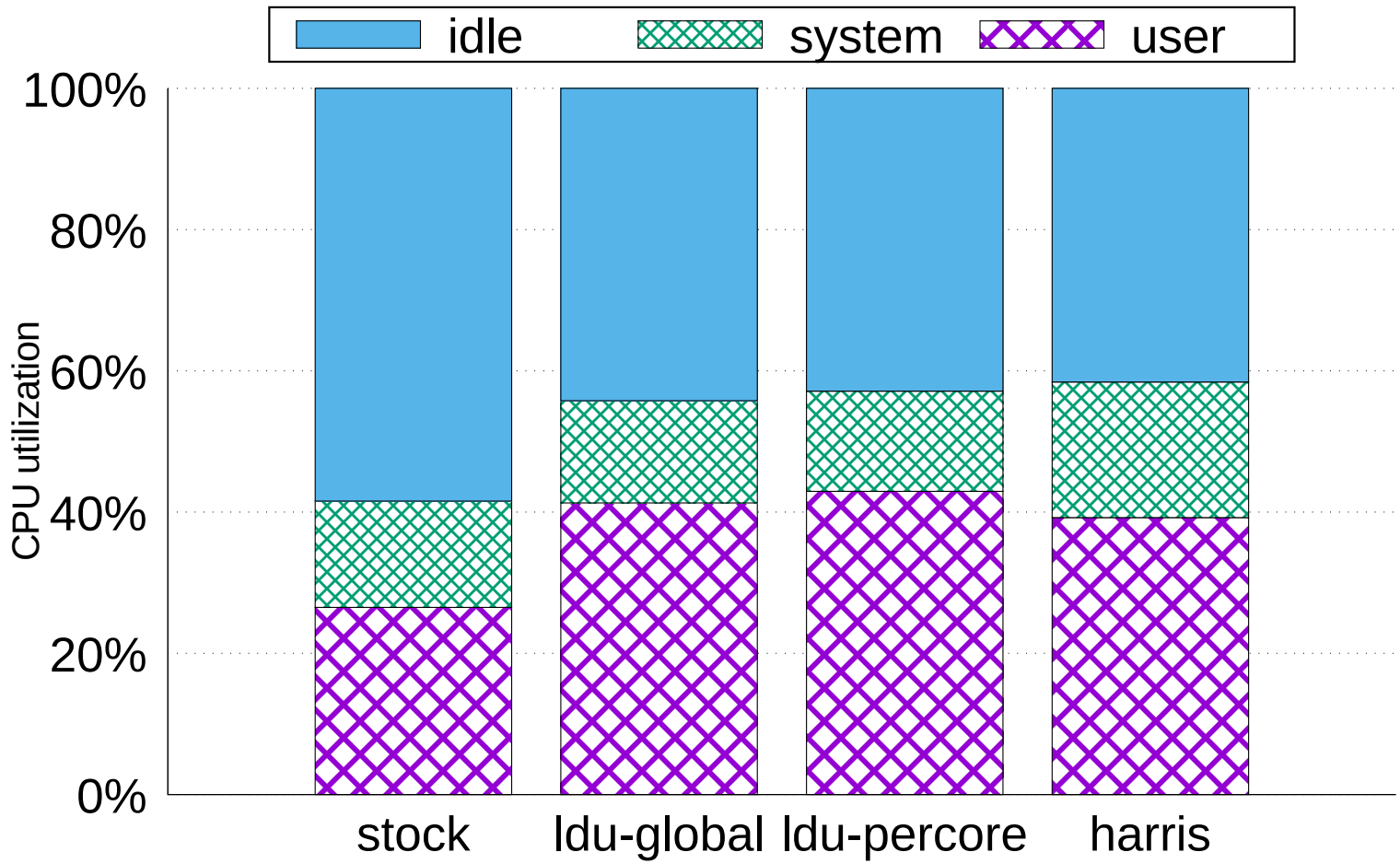
AIM7


1M
1M
800k
600k
400k
200k
0
jobs/min
0
20
40
60
80
100
120
cores
anon-stock, file-stock
anon-ldu-global, file-ldu-global
anon-harris, file-harris

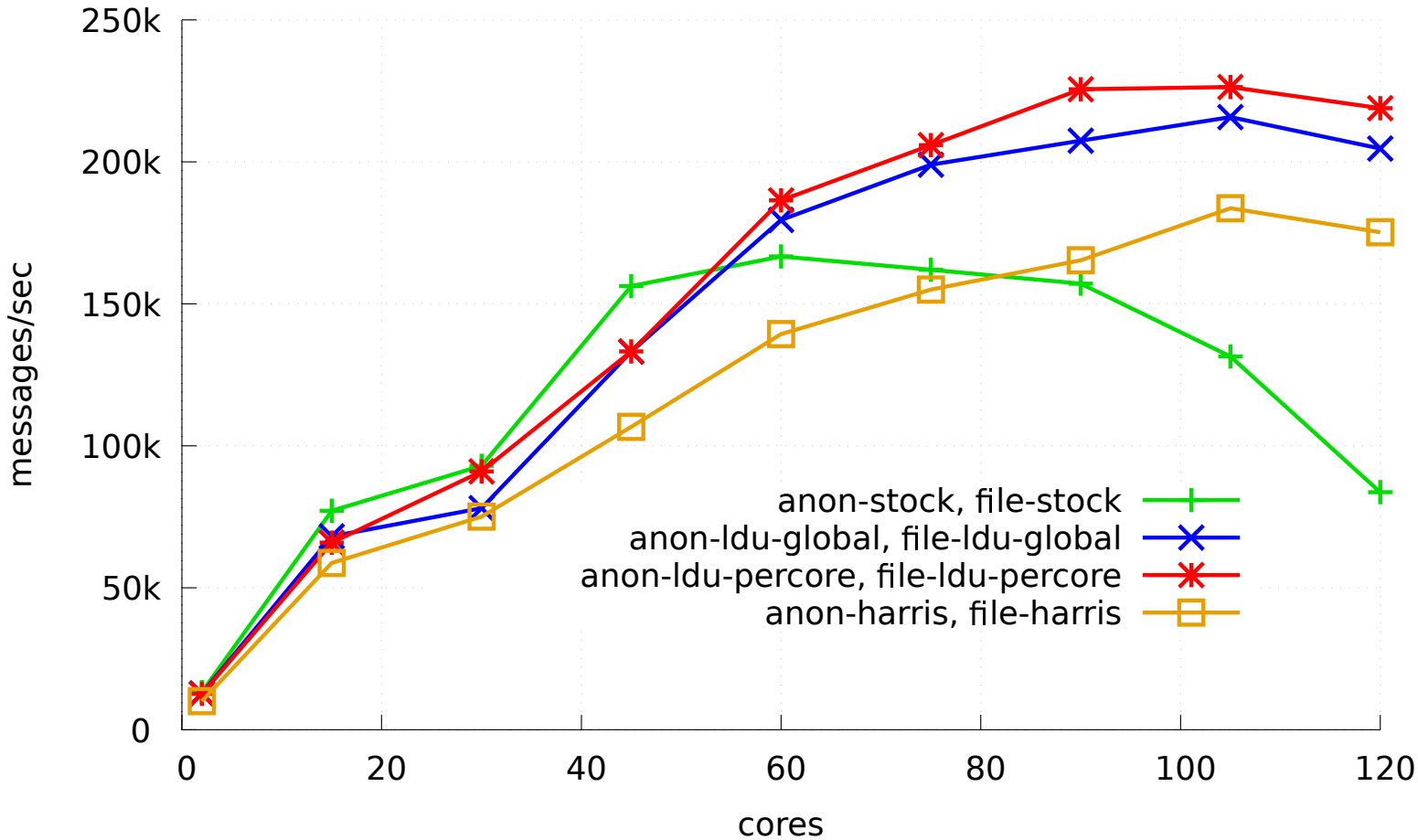
AIM7


1M
1M
800k
600k
400k
200k
0
jobs/min
0
20
40
60
80
100
120
cores
anon-stock, file-stock
anon-ldu-global, file-ldu-global
anon-ldu-percore, file-ldu-per-core
anon-harris, file-harris

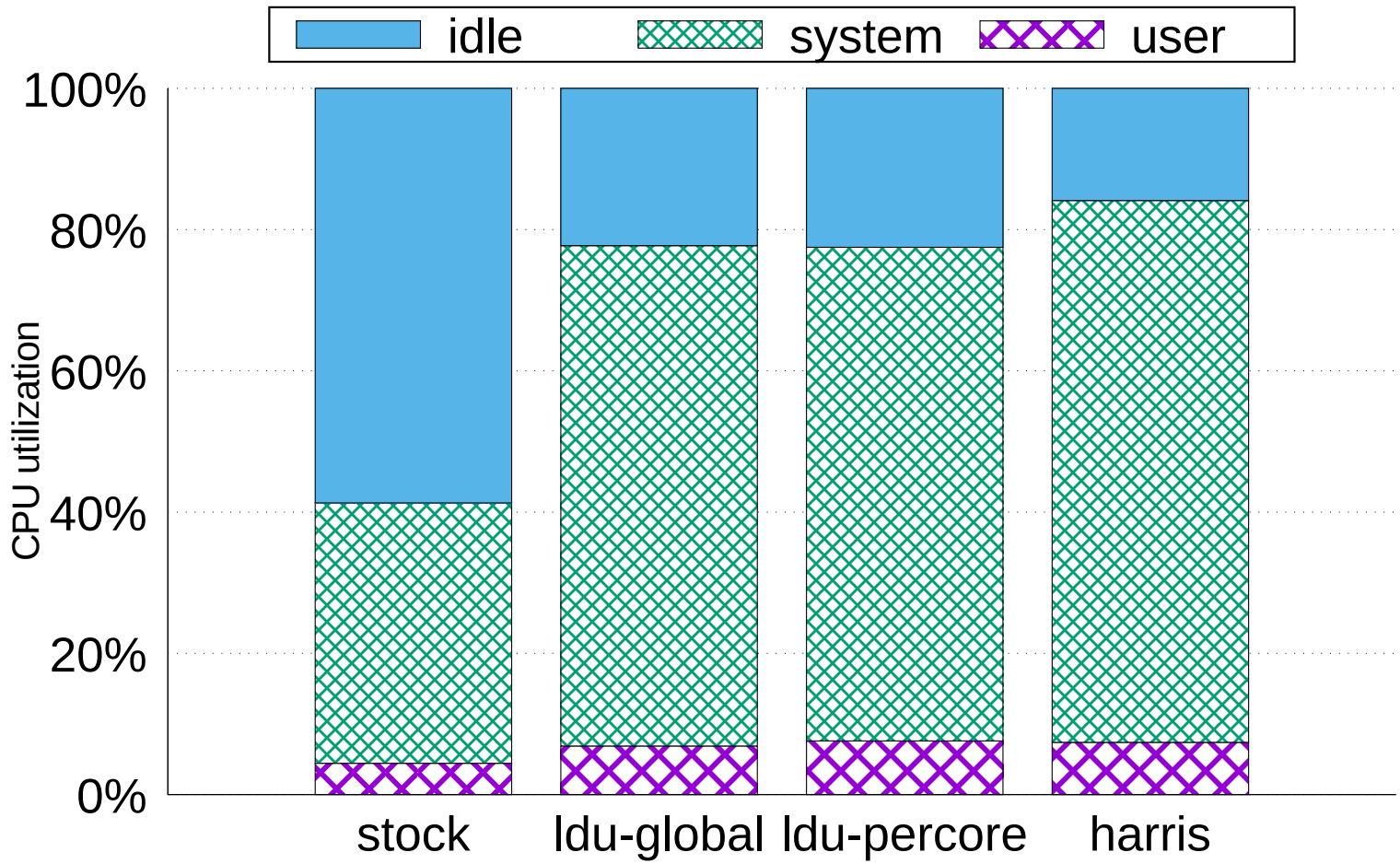
AIM7 – CPU utilization



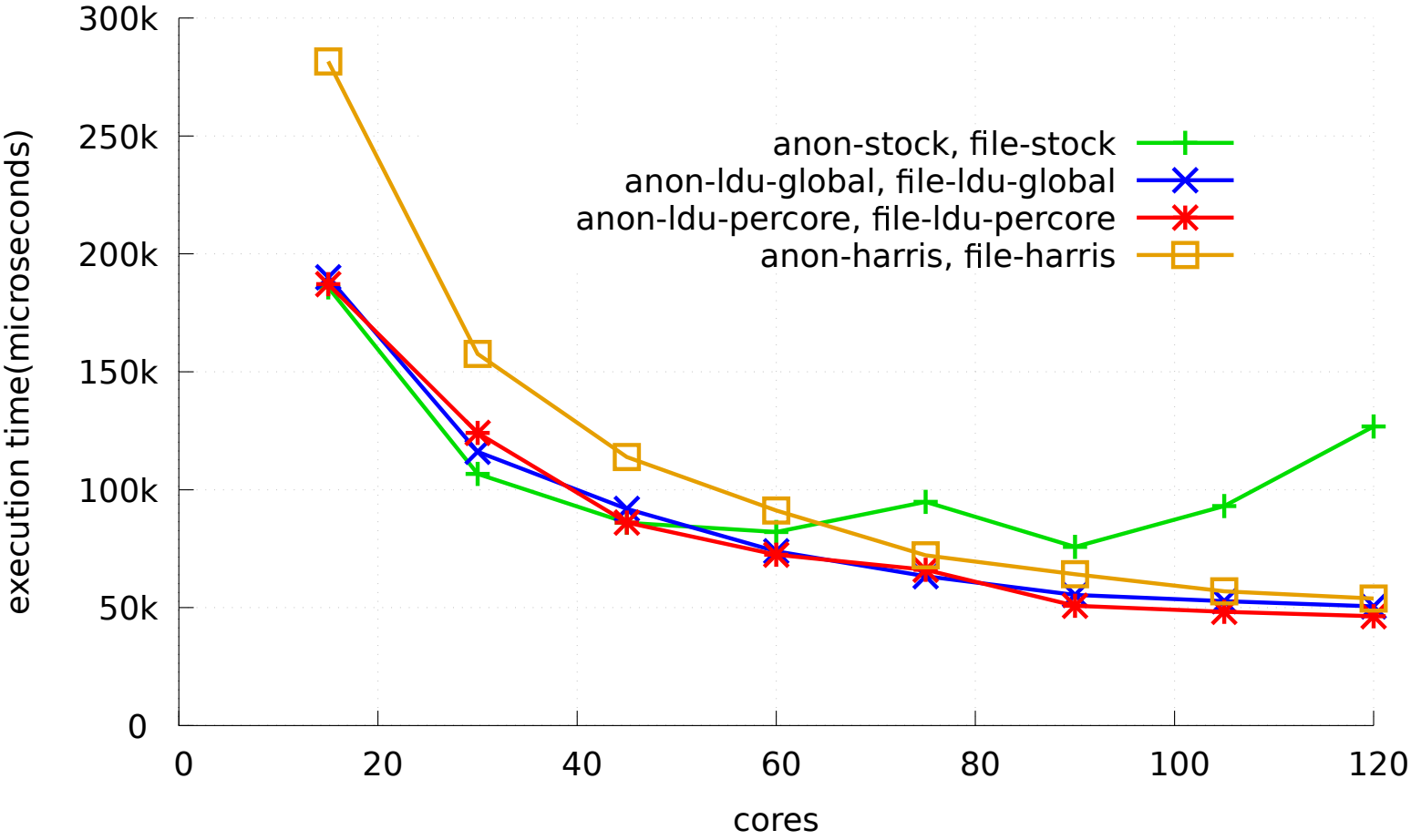
EXIM


250k
200k
150k
100k
50k
0
messages/sec
0
20
40
60
80
100
120
cores
anon-stock, file-stock
anon-ldu-global, file-ldu-global
anon-ldu-percore, file-ldu-percore
anon-harris, file-harris

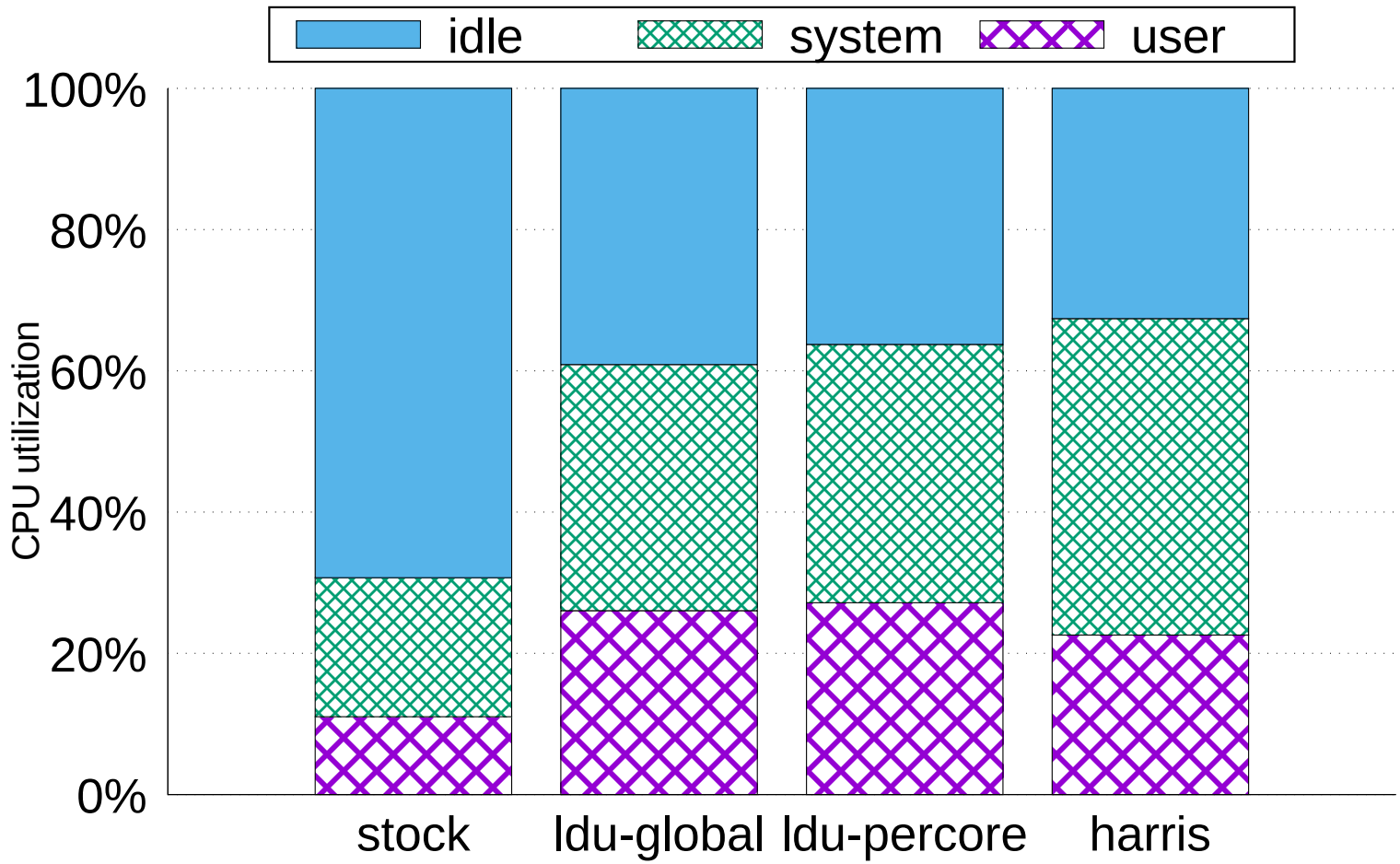
EXIM – CPU utilization



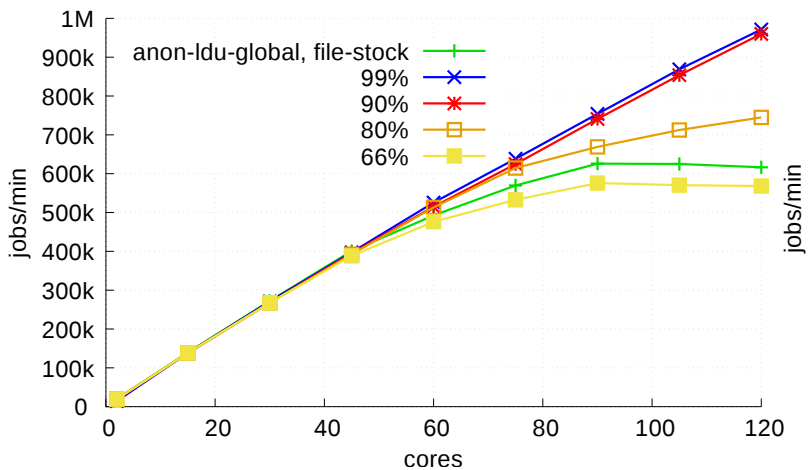
Lmbench

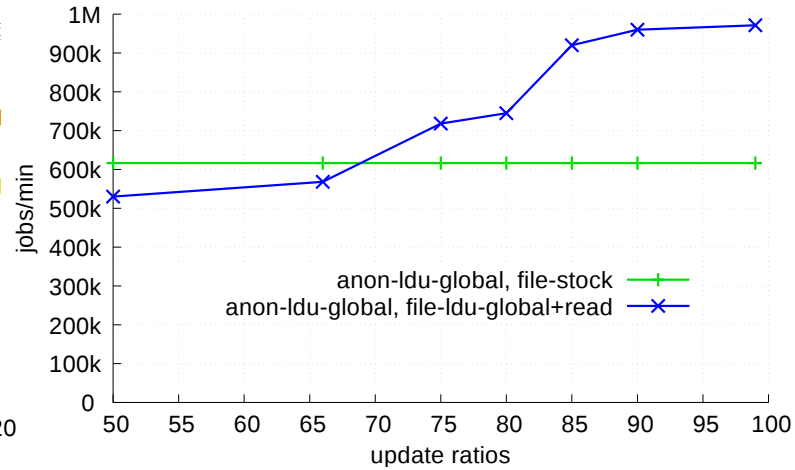

300k
250k
200k
150k
100k
50k
0
execution time(microseconds)
0
20
40
60
80
100
120
cores
anon-stock, file-stock
anon-ldu-global, file-ldu-global
anon-ldu-percore, file-ldu-percore
anon-harris, file-harris

Lmbench – CPU utilization

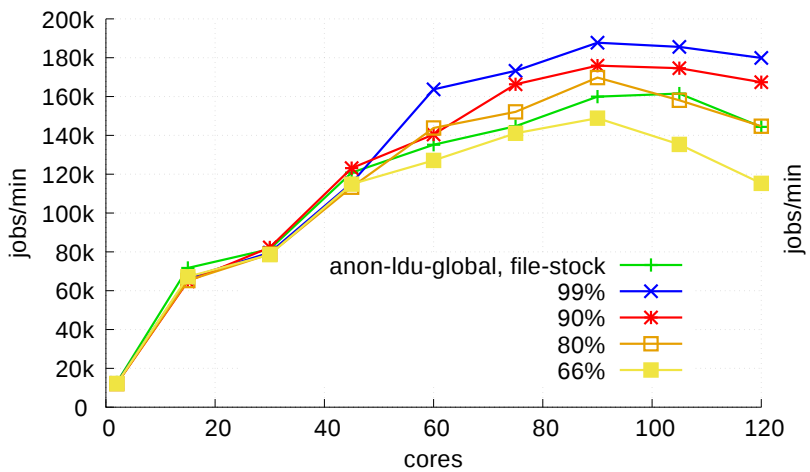


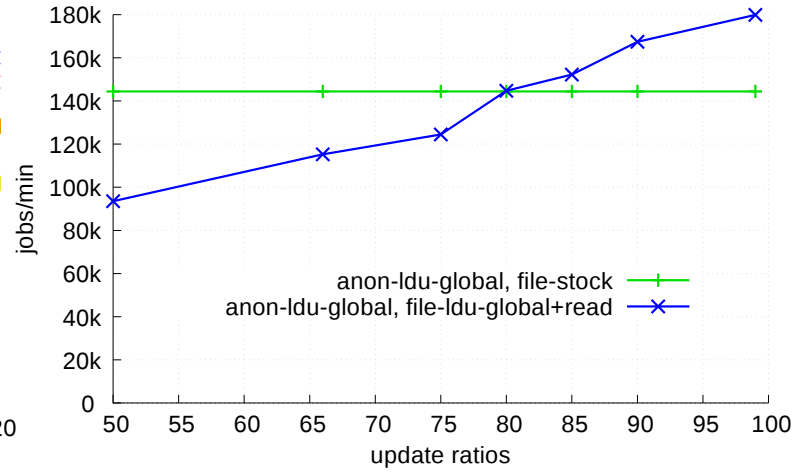
Update ratios – AIM7



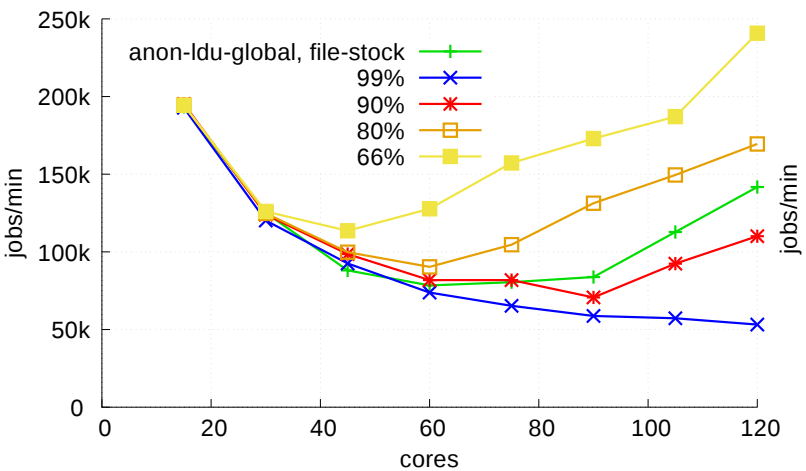


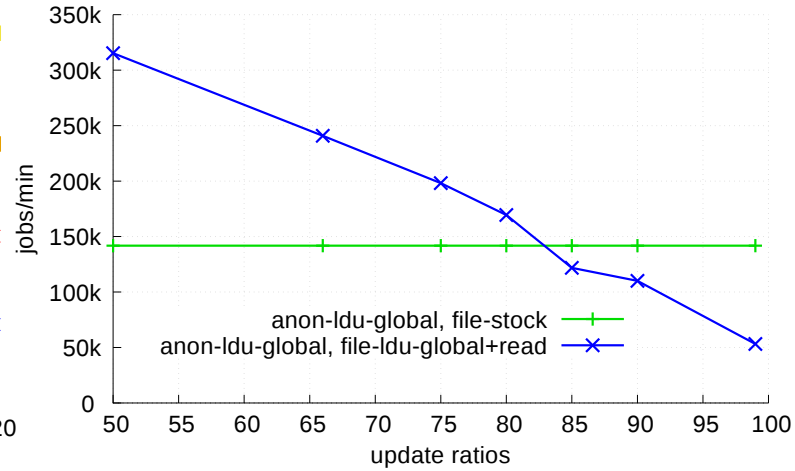
Update ratios – Exim





Update ratios – Lmbench





Related work

- Operating systems scalability.
  - Create new operating systems.
  - Optimize existing operating systems.
- Scalable lock.
  - Queue-based locks.
  - Hierarchical locks.
  - Delegation techniques.
- Scalable data structures.
  - Different performances depending on their update ratios.

Futuer Directions

- Combine LDU with time-stamp based approach for stack, queue.
- Combine RCU readers with log-based updates.

Combine RCU readers with log-based updates

News from the source

Content

Weekly Edition
Archives
Search
Kernel
Security
Distributions
Events calendar
Unread comments

LWN FAQ
Write for us

within the STM community. It is therefore doubly good to see his prescient work acknowledged.

I was surprised to see this paper assert that RCU only permits single writers. After all, multi-writer RCU-protected data structures have been used in production for decades in the guise of hash tables, and, as mentioned earlier, a 2004 USENIX paper notes that RCU has been observed satisfying more than 1,000 updates with a single grace period. This of course means that a single reader could see more than 1,000 updates. Interestingly enough, Nir Shavit's own split-ordered-list resizable hash table is itself an example of multiple concurrent updates: Mathieu Desnoyers converted it to RCU-protected form as part of the user-space RCU library.

On the other hand, perhaps they mean something else by the single-writer limitation:

1. They might be differentiating between TM-based updates and other types of updates. However, in 2011, Phil Howard and Jonathan Walpole published a paper [PDF] describing a red-black tree that featured RCU readers and STM writers. Prior to that, Ramadan et al. published another paper [PDF] adding TM-based updates to the Linux kernel (where they interacted with RCU readers), and prior to that, Keir Fraser's Ph.D dissertation [PDF] applied RCU-like techniques to STM implementations. (Note that Matveev et al. go beyond all this work by providing facilities for easy, or at least easier, application to multiple algorithms and data structures.)

2. Perhaps they are arguing that RCU updates always use a single pointer assignment to expose new data to readers. My solution to the Issaquah Challenge [PDF] (which updates this C++ Standards Committee paper [PDF]) would be an exception, but perhaps they are unaware of this work, which to be fair has not yet been formally published in any academic venue.

3. They might mean the coupling of RCU readers with log-based updates, but Silas Boyd-Wickizer's 2014 Ph.D. dissertation [PDF] seems to cover this possibility. Some might argue that Linux log-based filesystems such as Btrfs also cover this possibility.

4. They might mean that a given RCU reader can only be exposed to single update. This certainly can be the case if a given reader is only permitted to traverse data that is updated under a single lock and if updaters hold that lock across a grace period following each update. However, in my experi
(Here [PDF] is an example.)

December 16, 2015
This article was contributed by Paul McKenney

Conclusion

- https://github.com/manycore-ldu/ldu