경량 로그 기반 지연 업데이트 기법을 활용한 리눅스 커널 확장성 향상

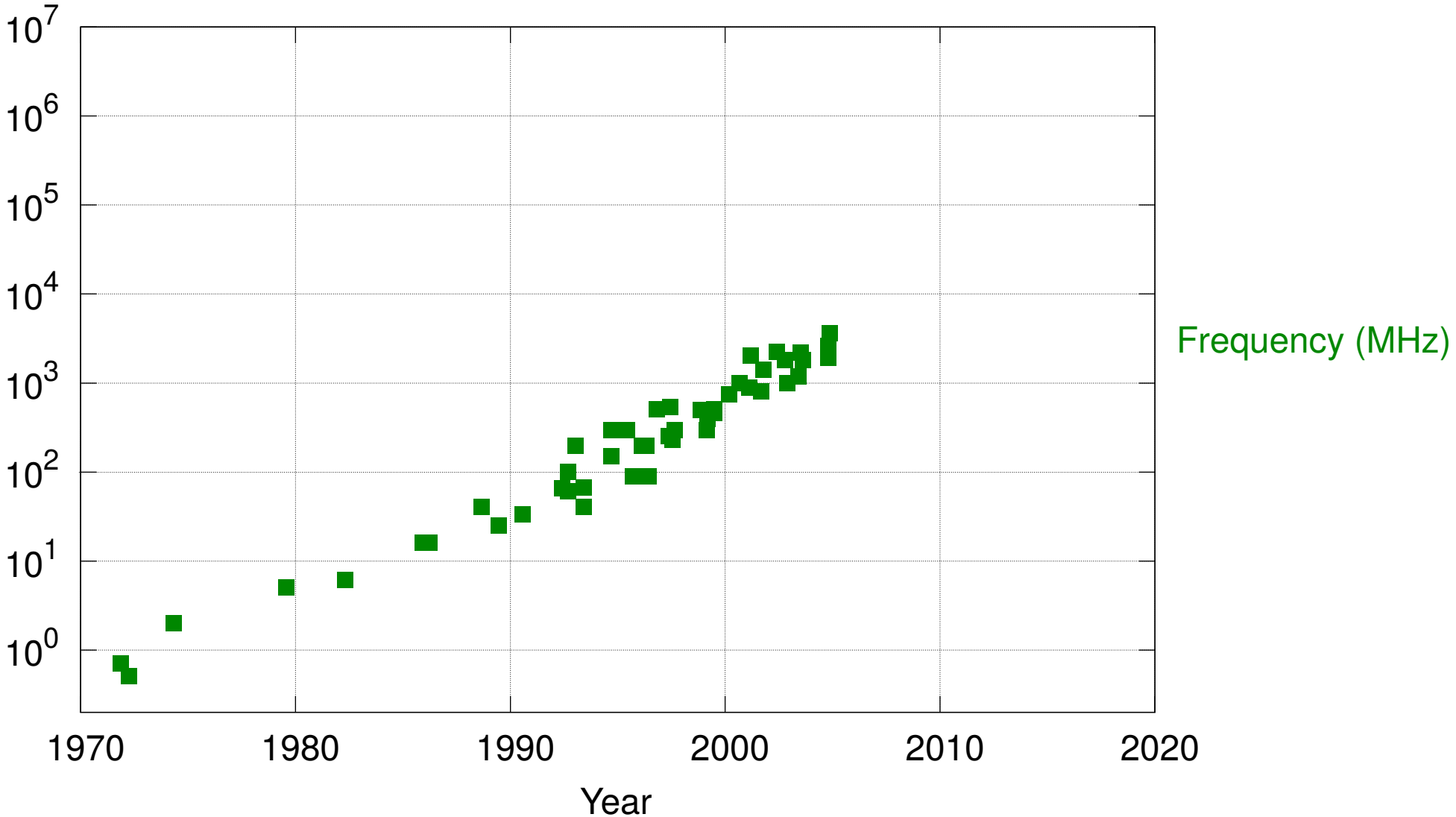
A Lightweight Log-based Deferred Update for Linux Kernel Scalability

Joohyun Kyong

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

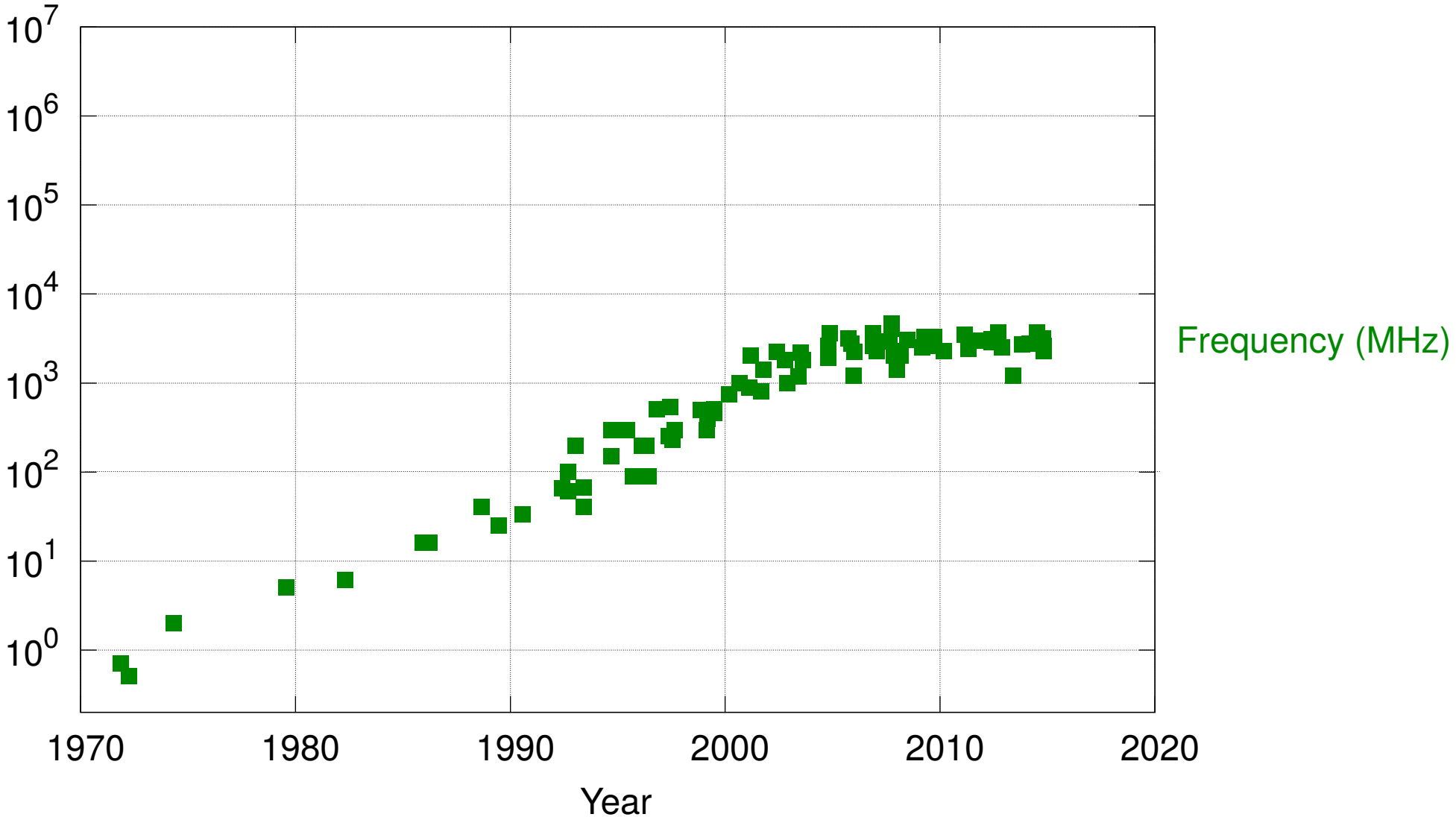
January 9, 2017

40 Years of Microprocessor Trend Data



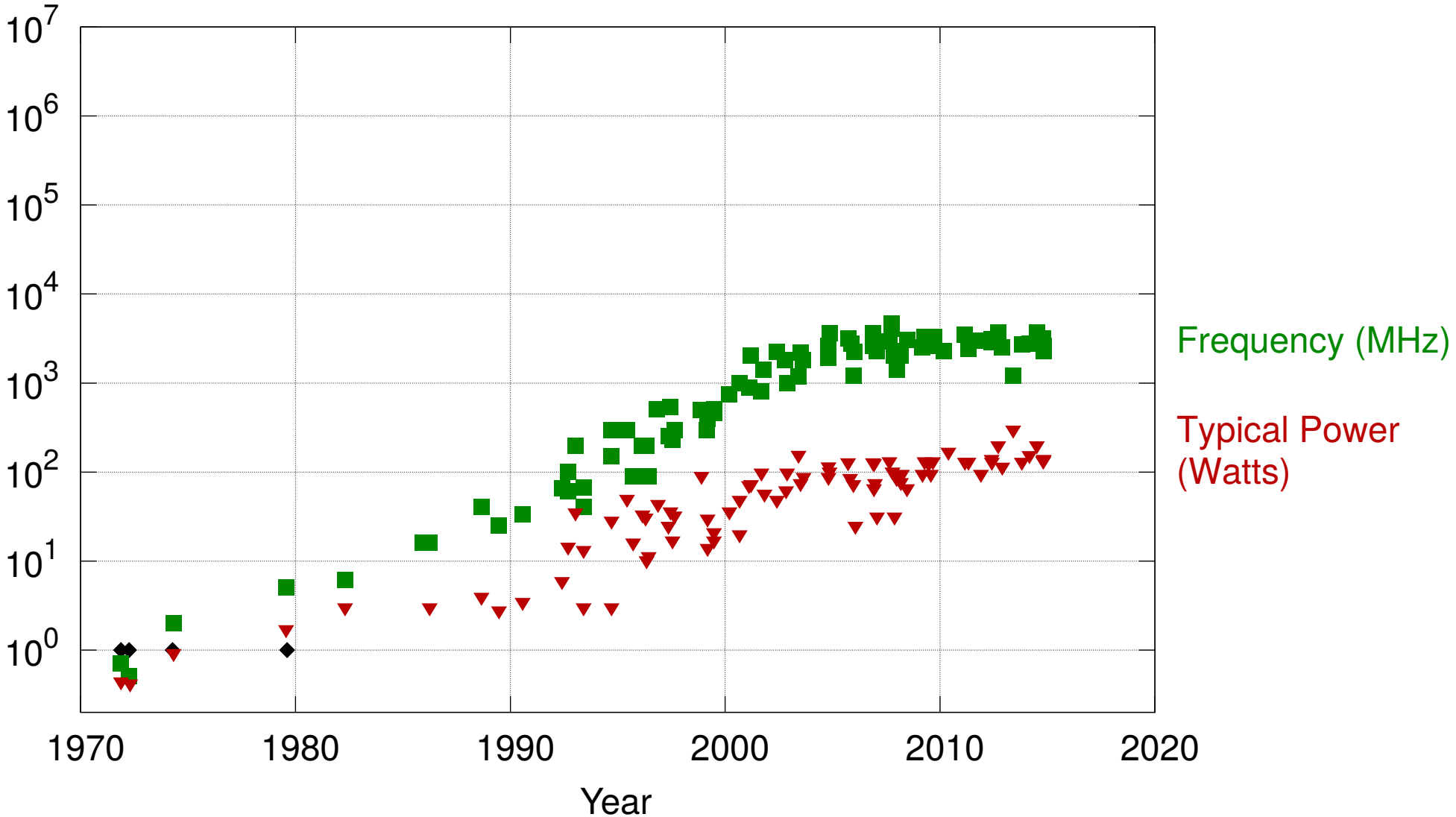
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

40 Years of Microprocessor Trend Data



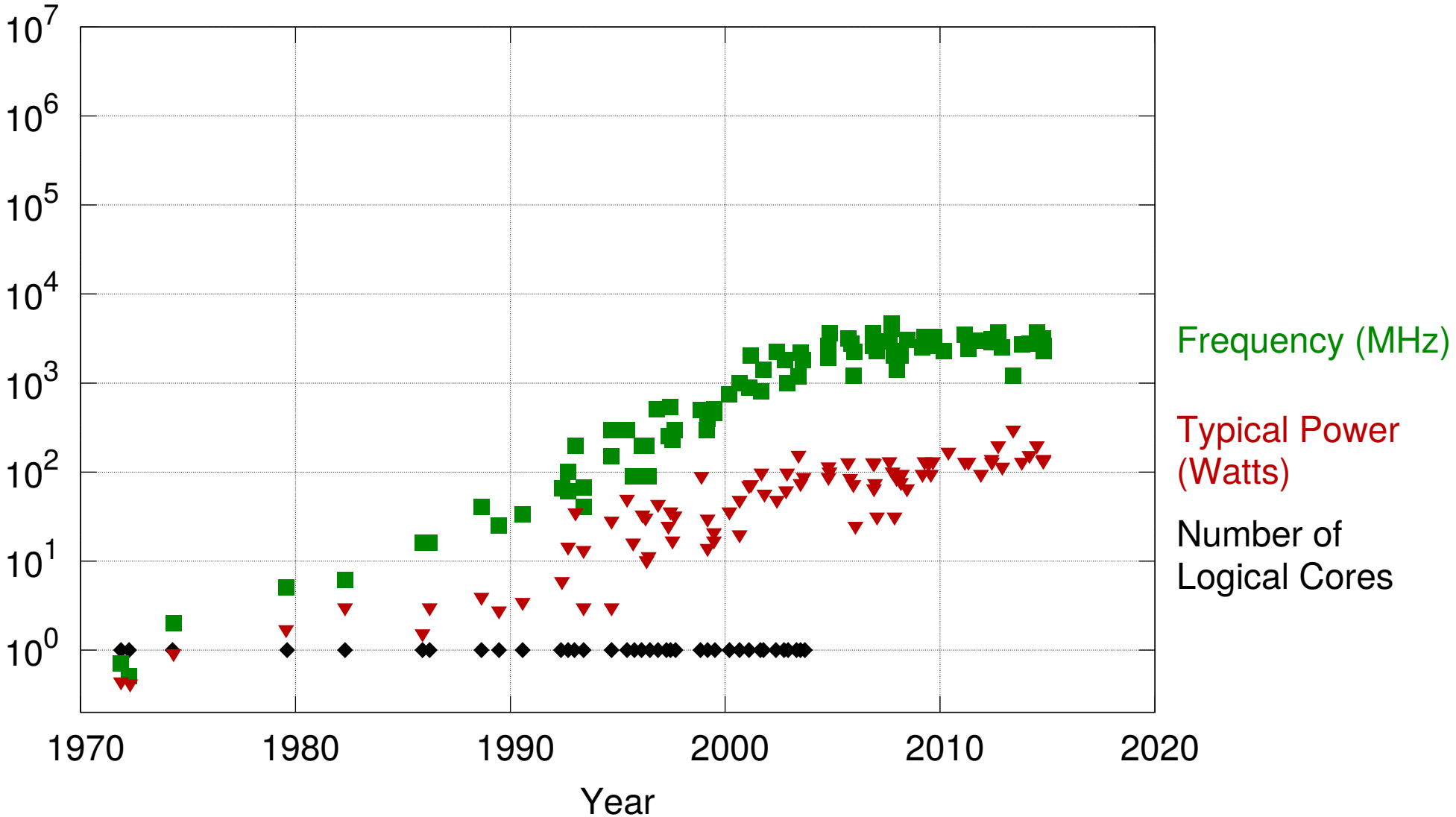
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

40 Years of Microprocessor Trend Data



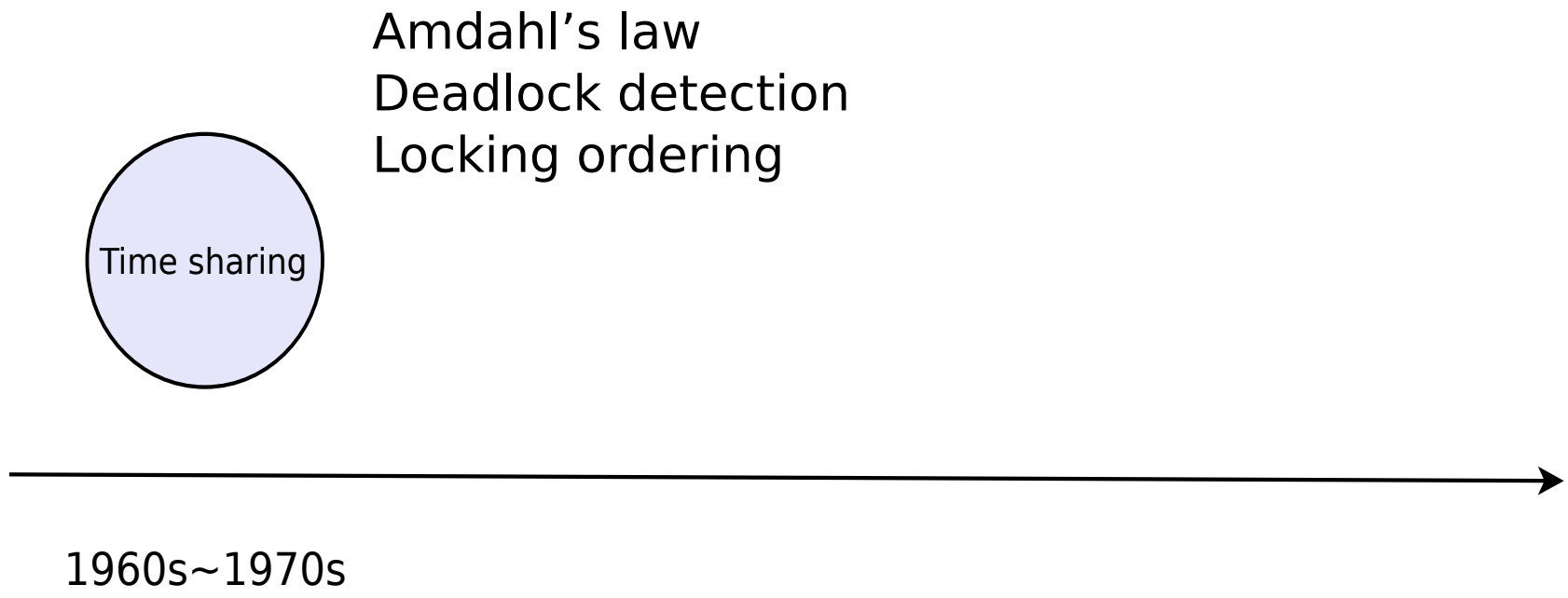
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

40 Years of Microprocessor Trend Data



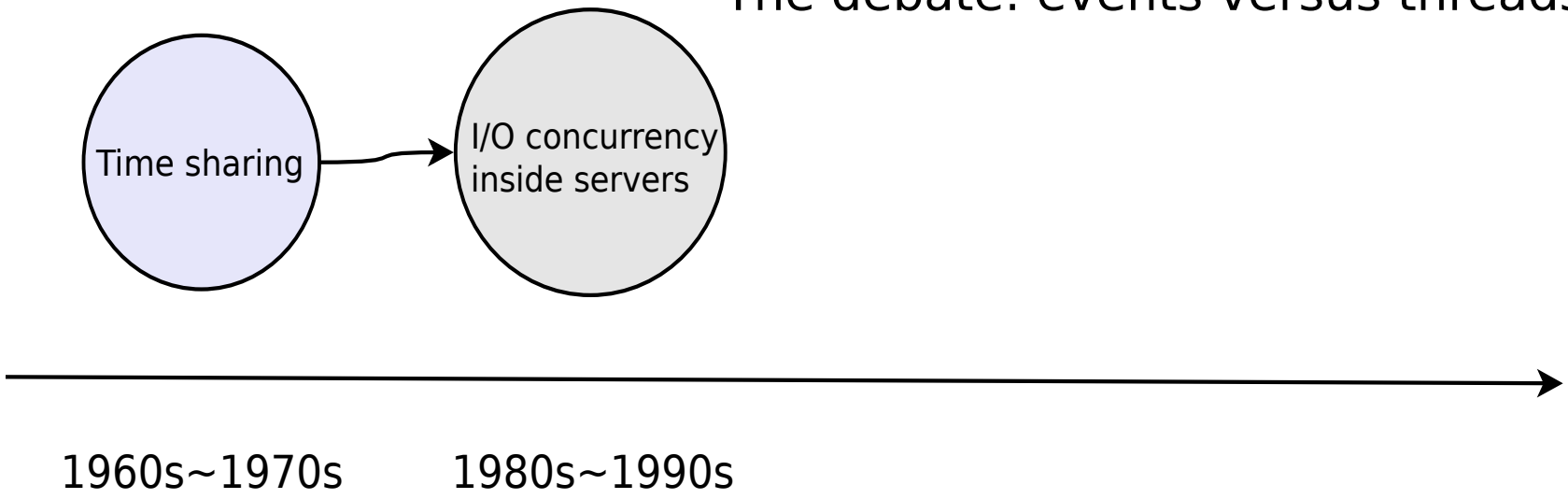
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

OS Kernel Scalability History



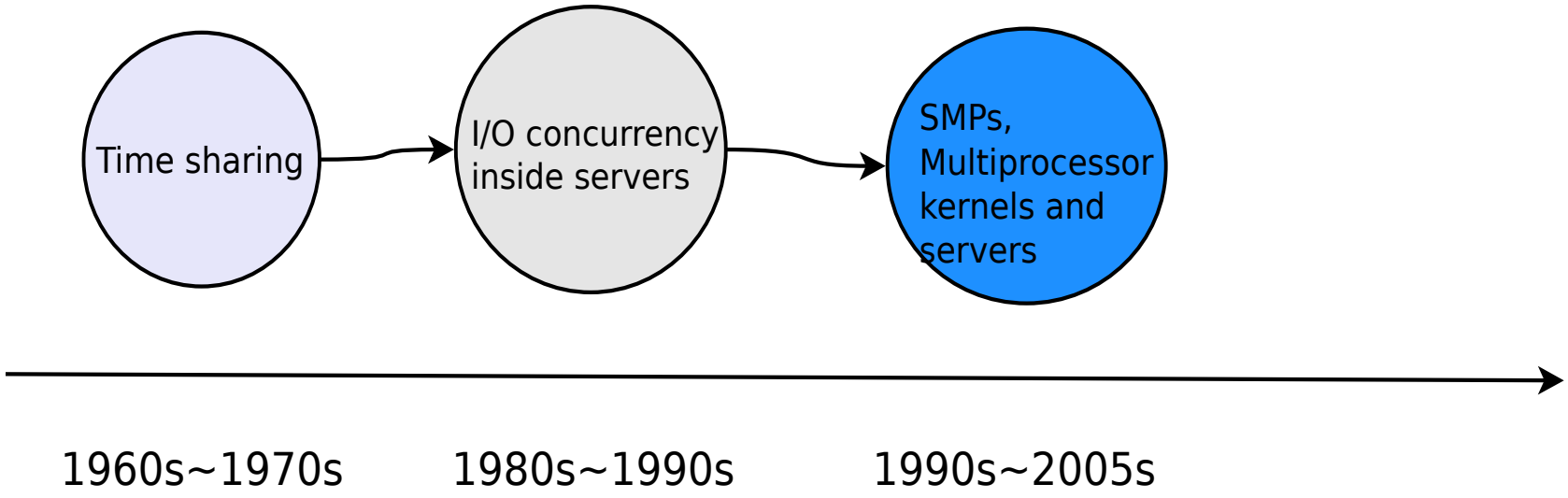
OS Kernel Scalability History

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

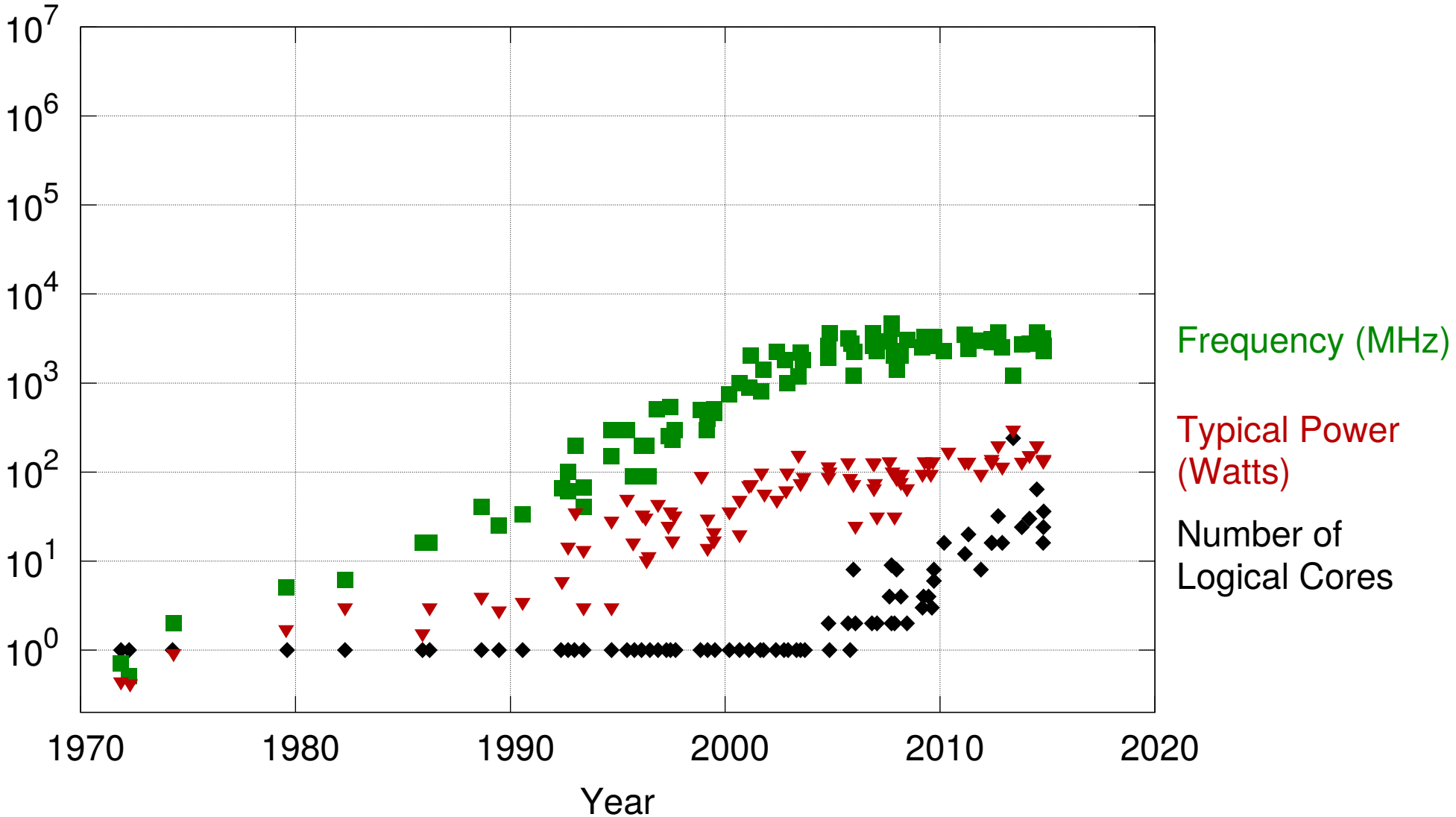


OS Kernel Scalability History

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

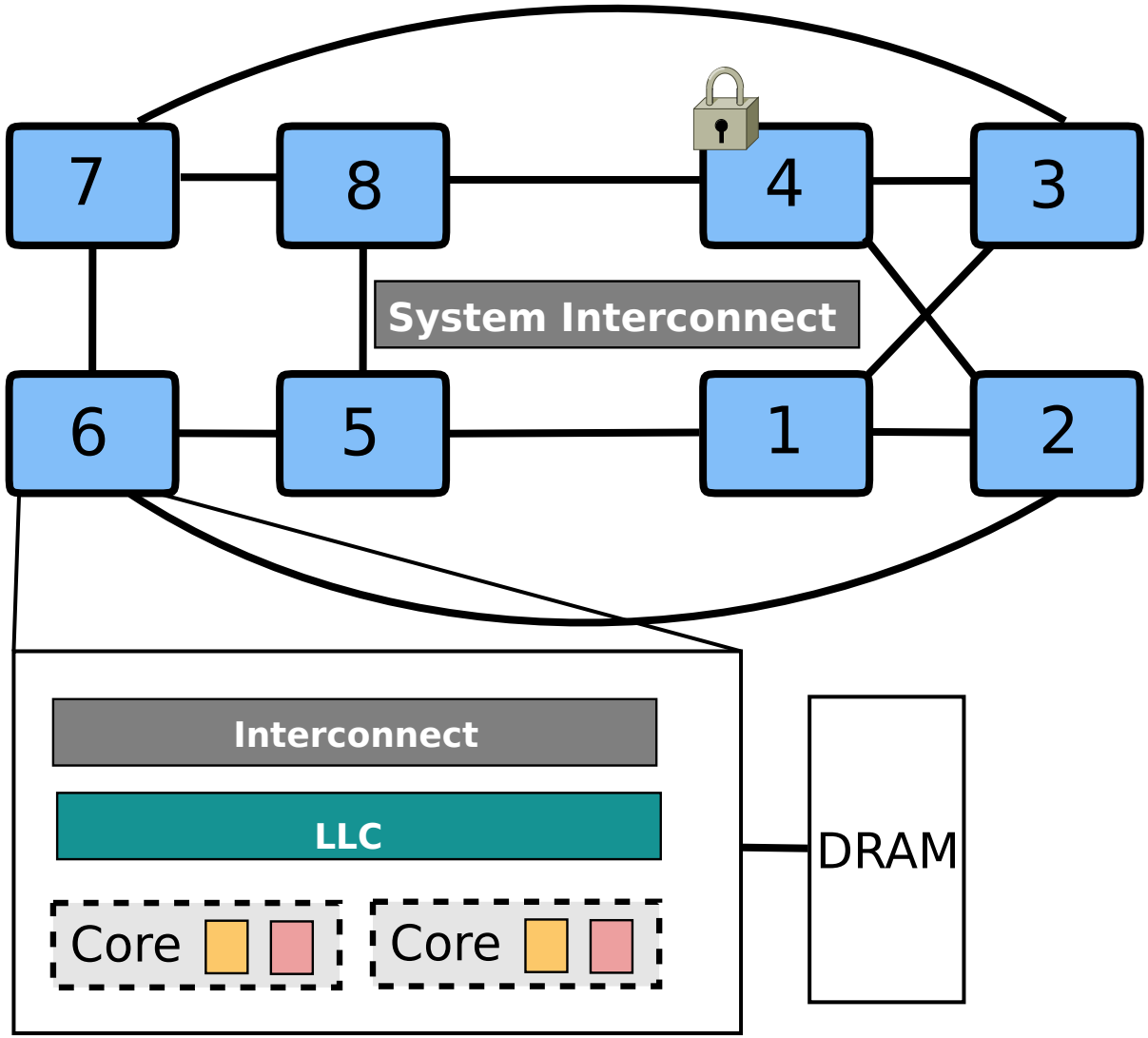


40 Years of Microprocessor Trend Data

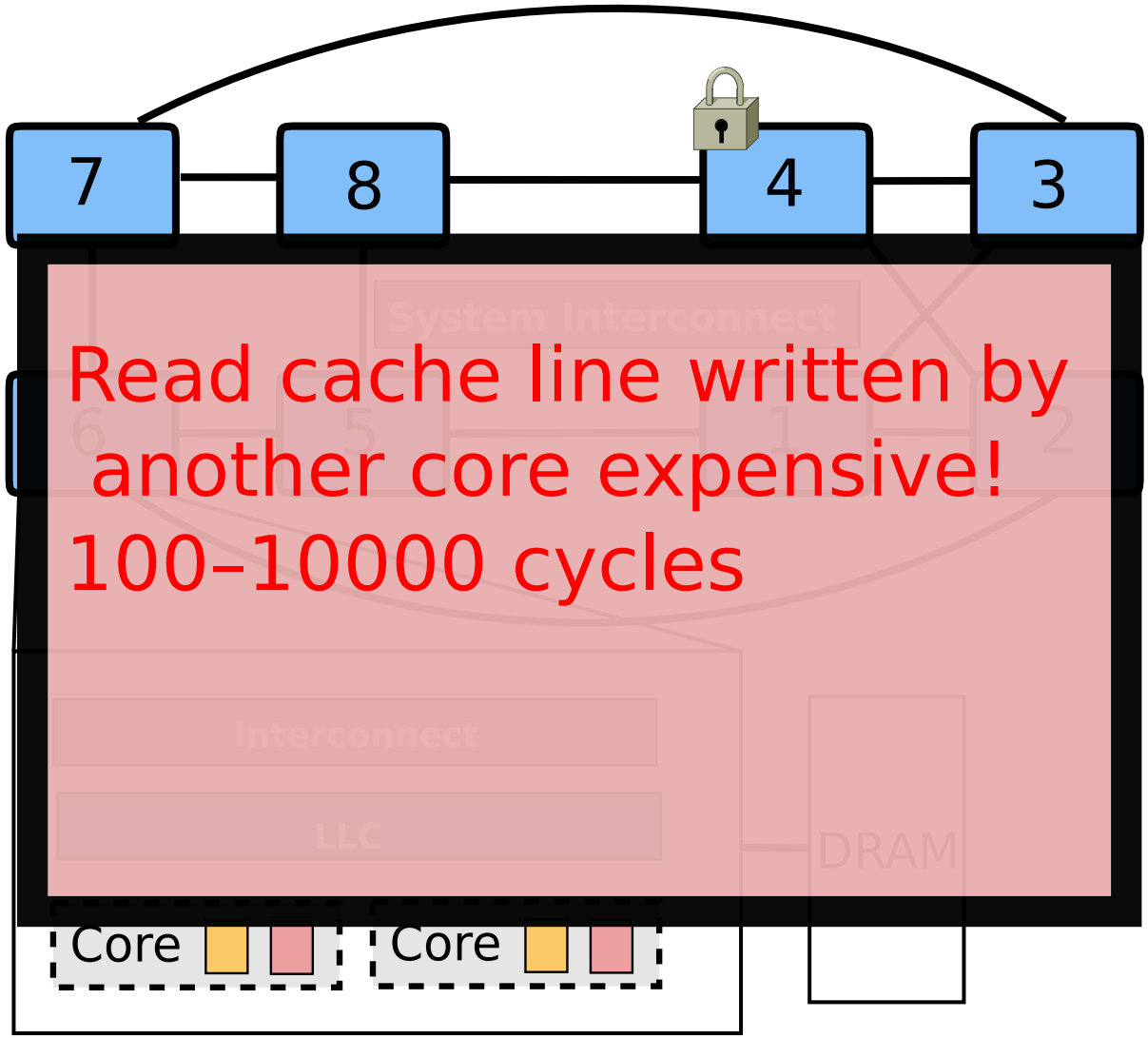


Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

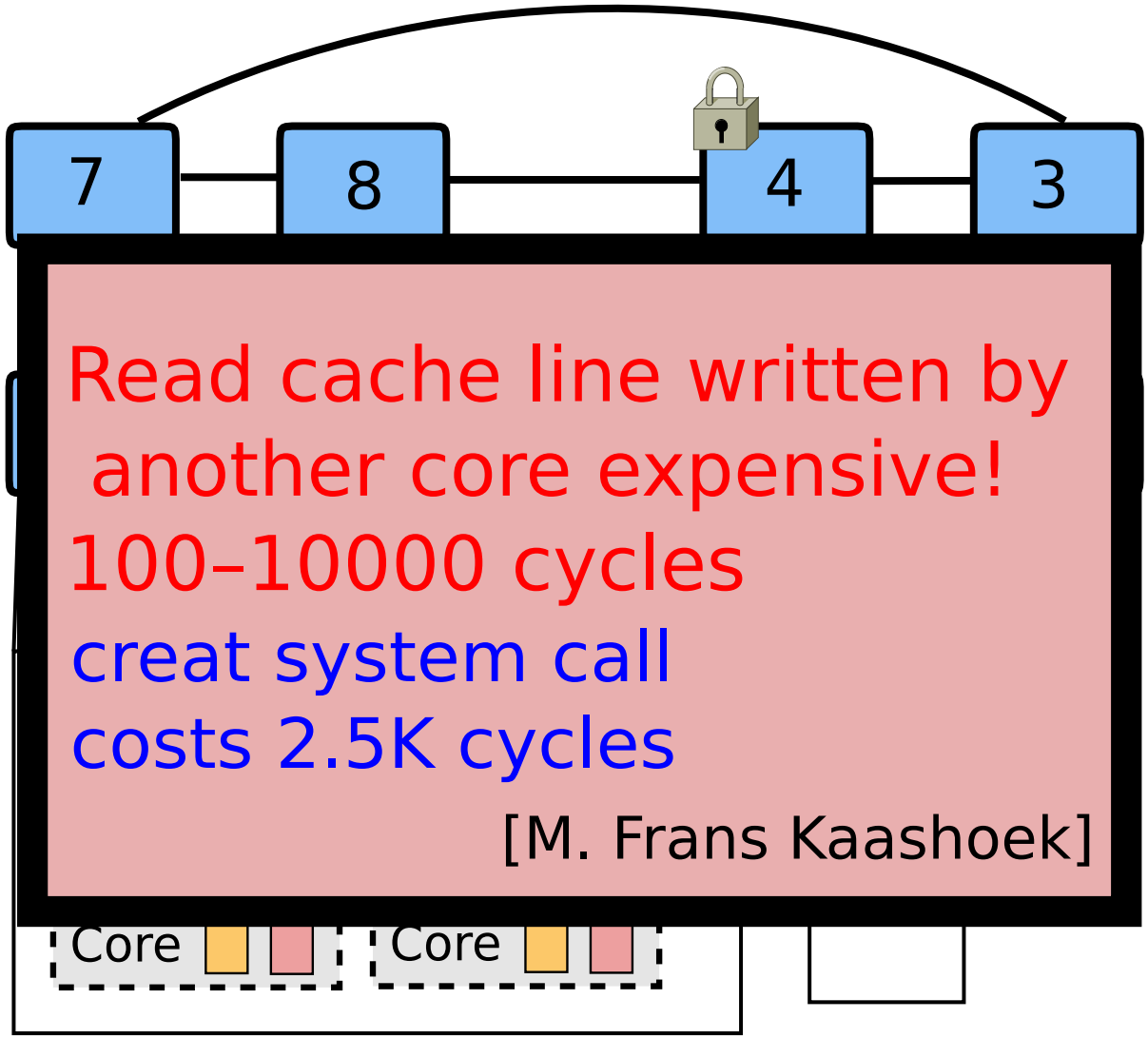
Cache-Coherence System



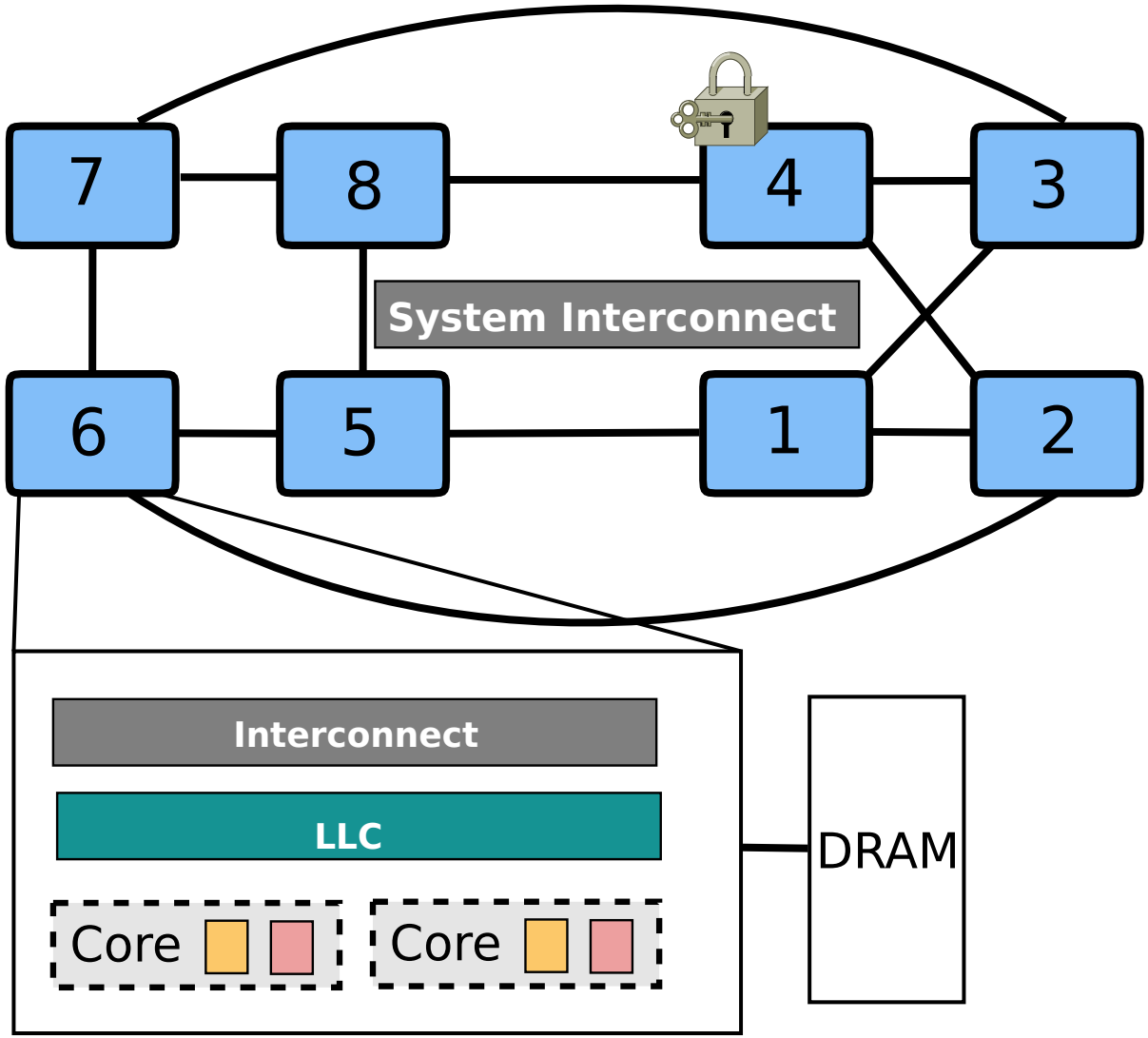
Cache-Coherence System



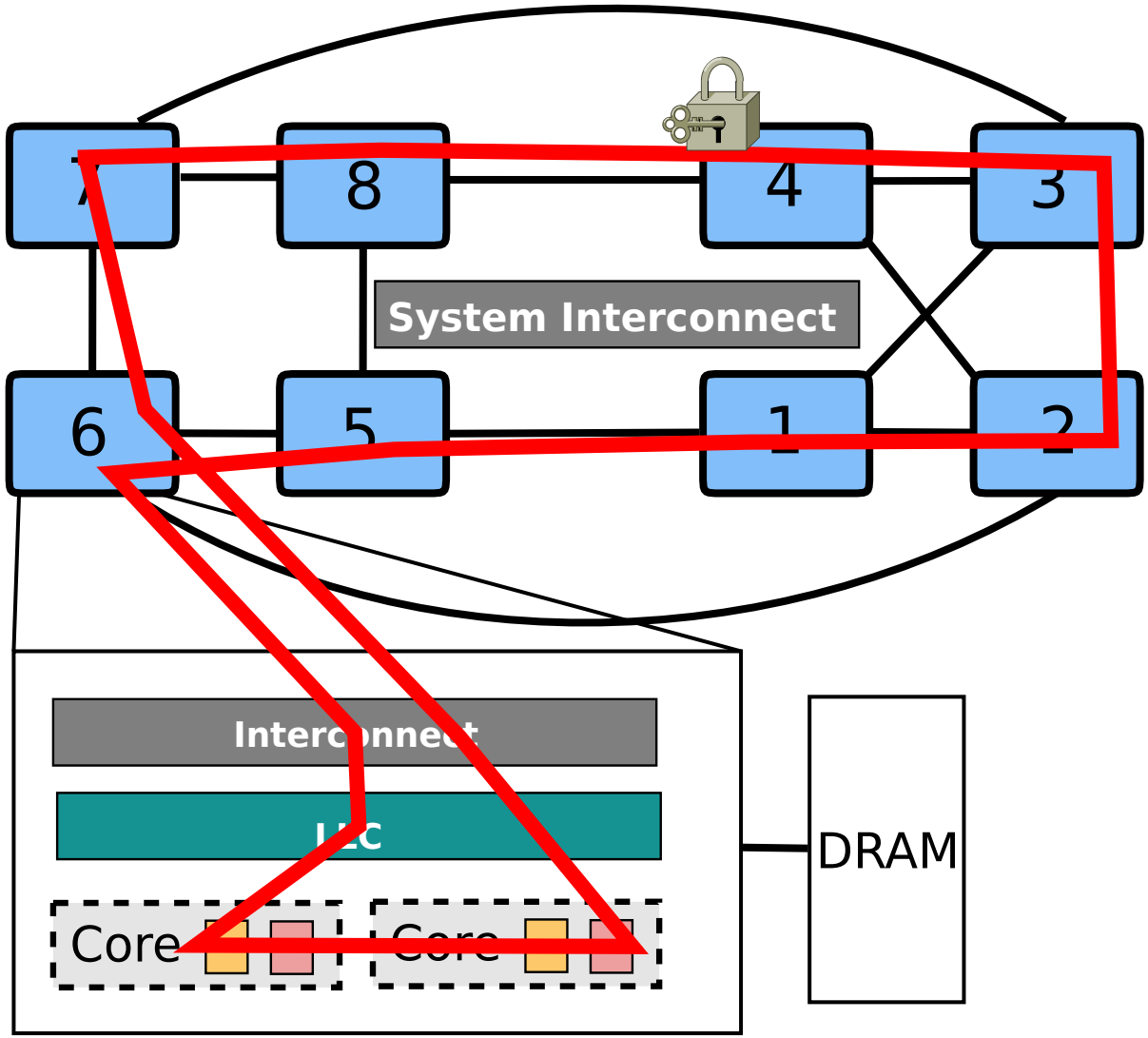
Cache-Coherence System



Cache-Coherence System

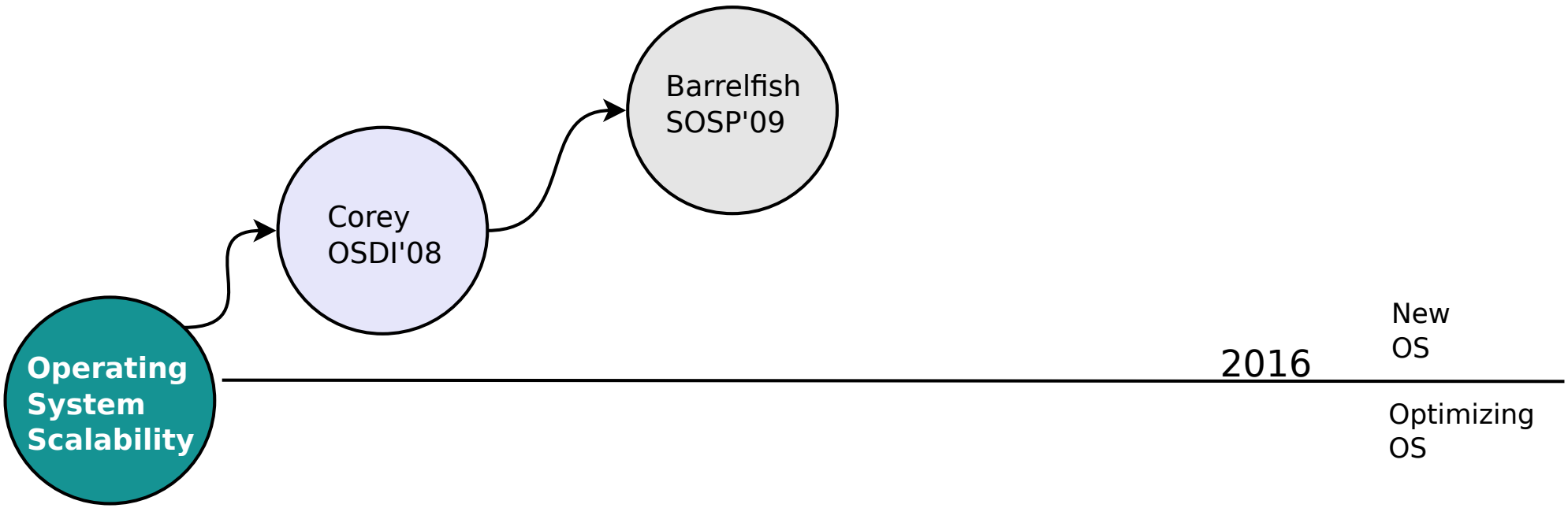


Cache-Coherence System

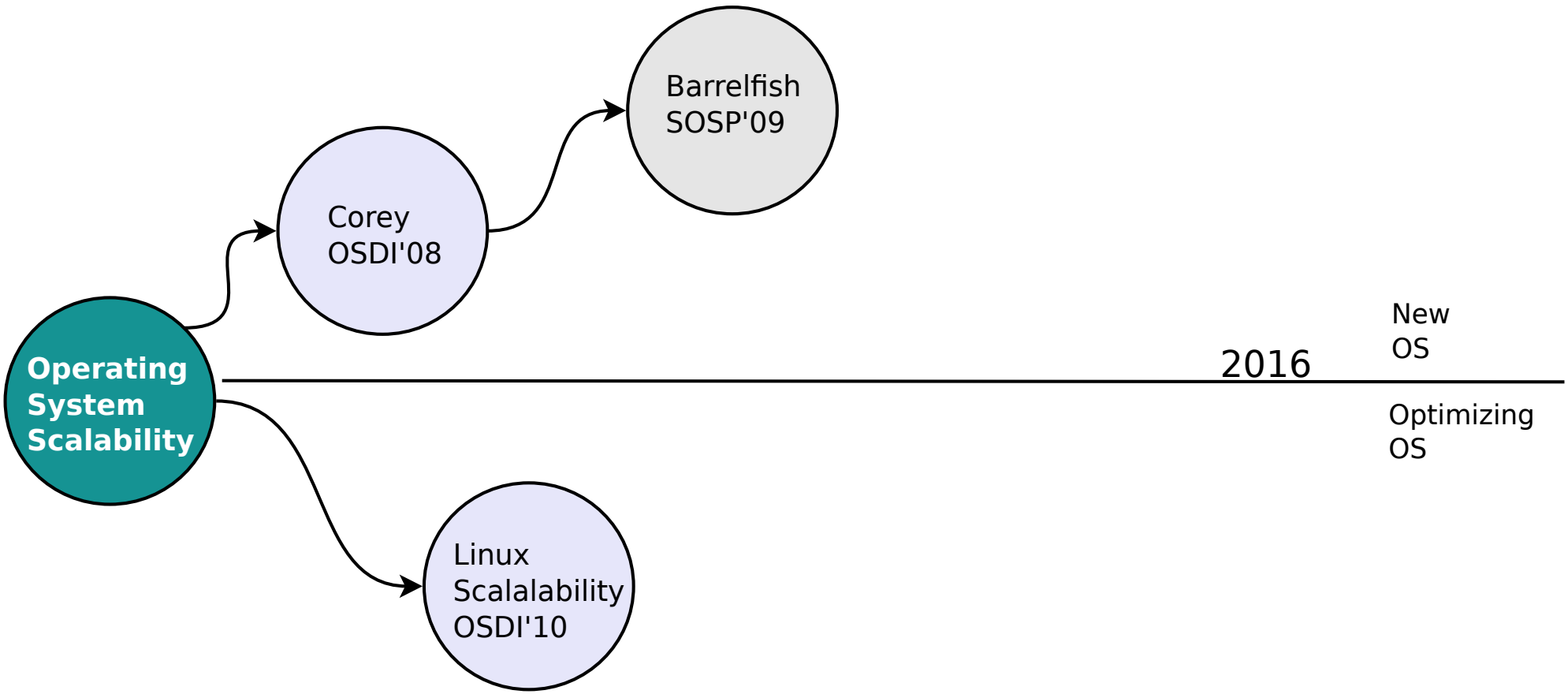


OS Kernel Scalability History

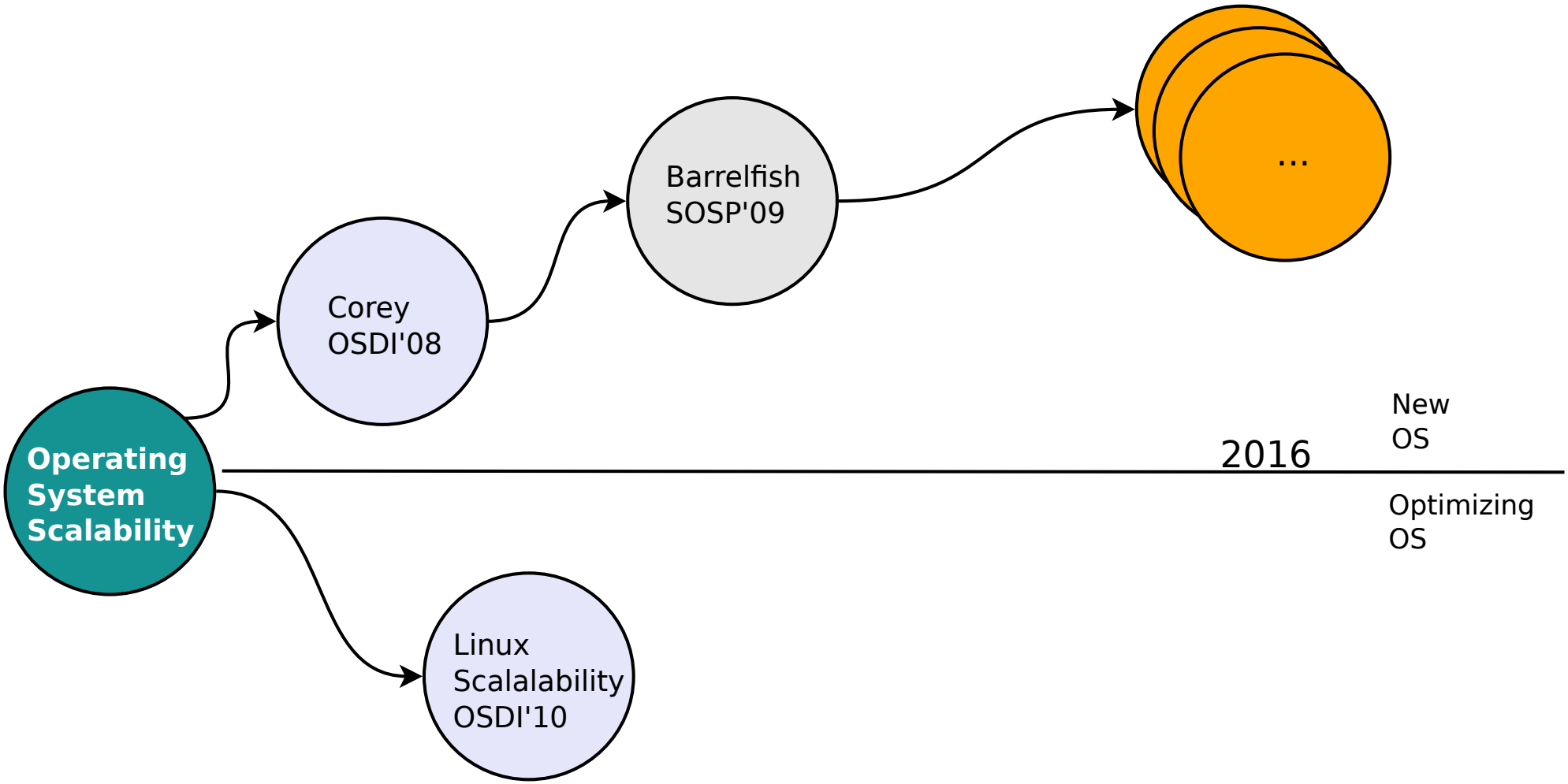
OS Kernel Scalability History



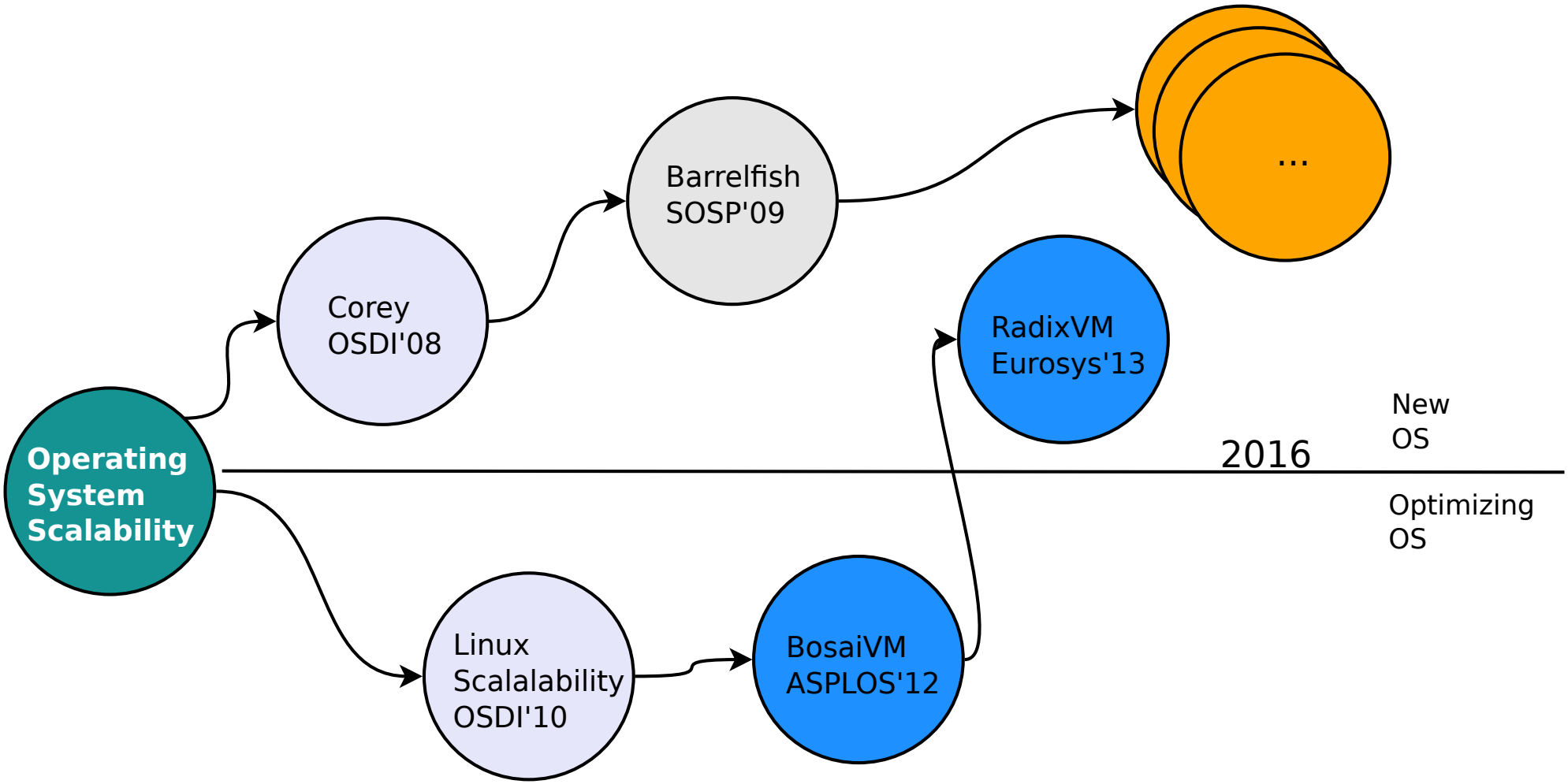
OS Kernel Scalability History



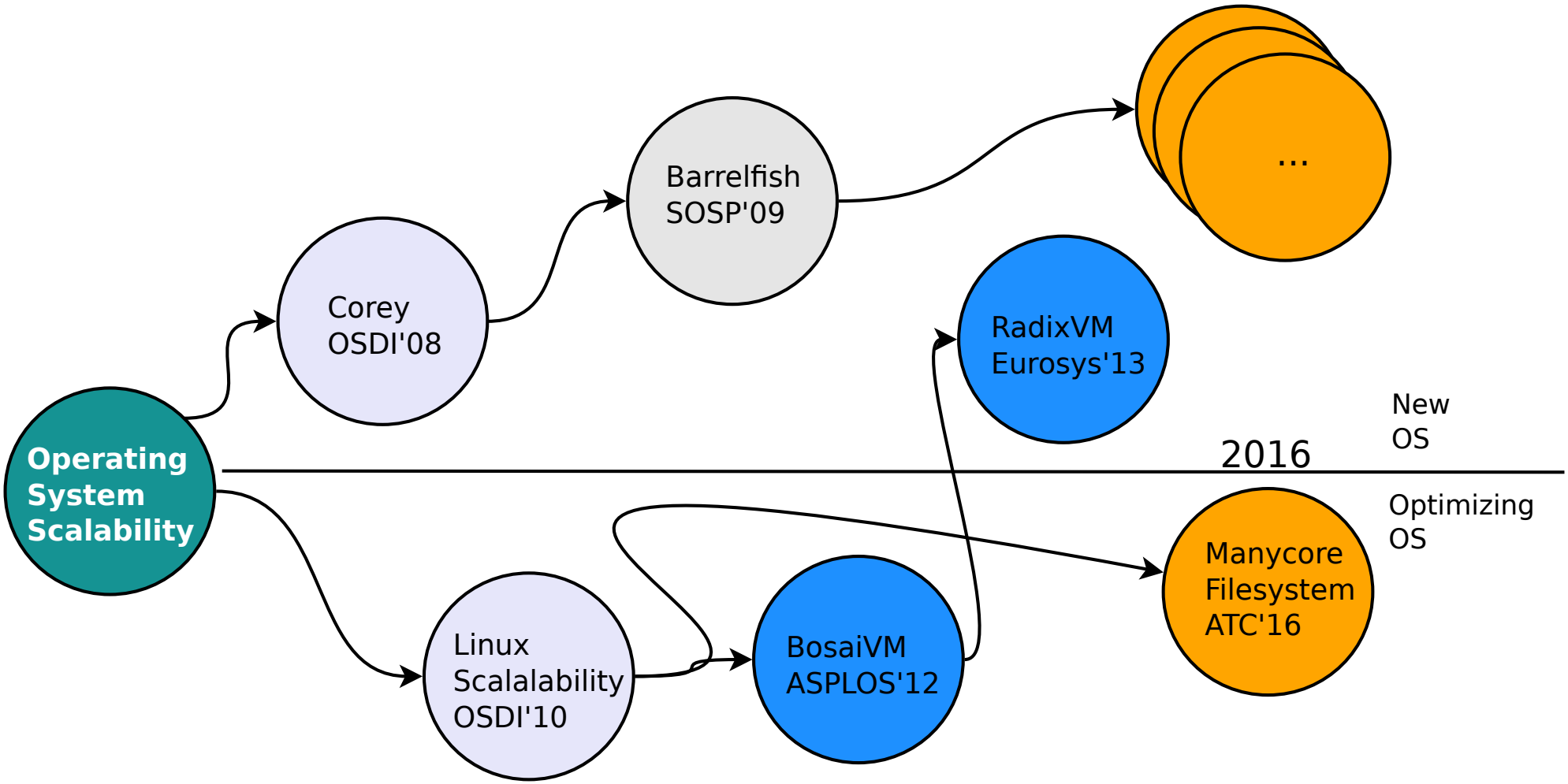
OS Kernel Scalability History



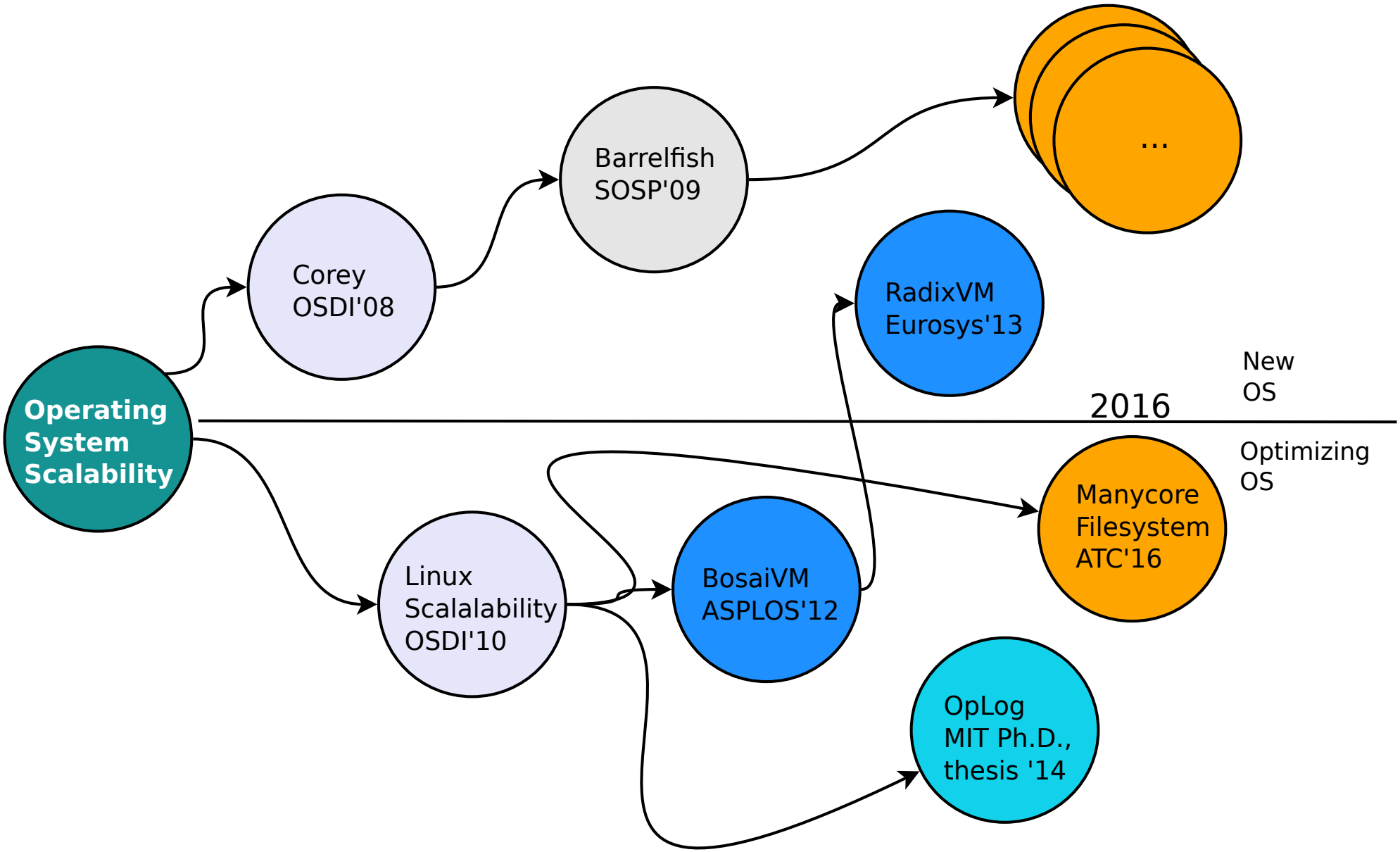
OS Kernel Scalability History



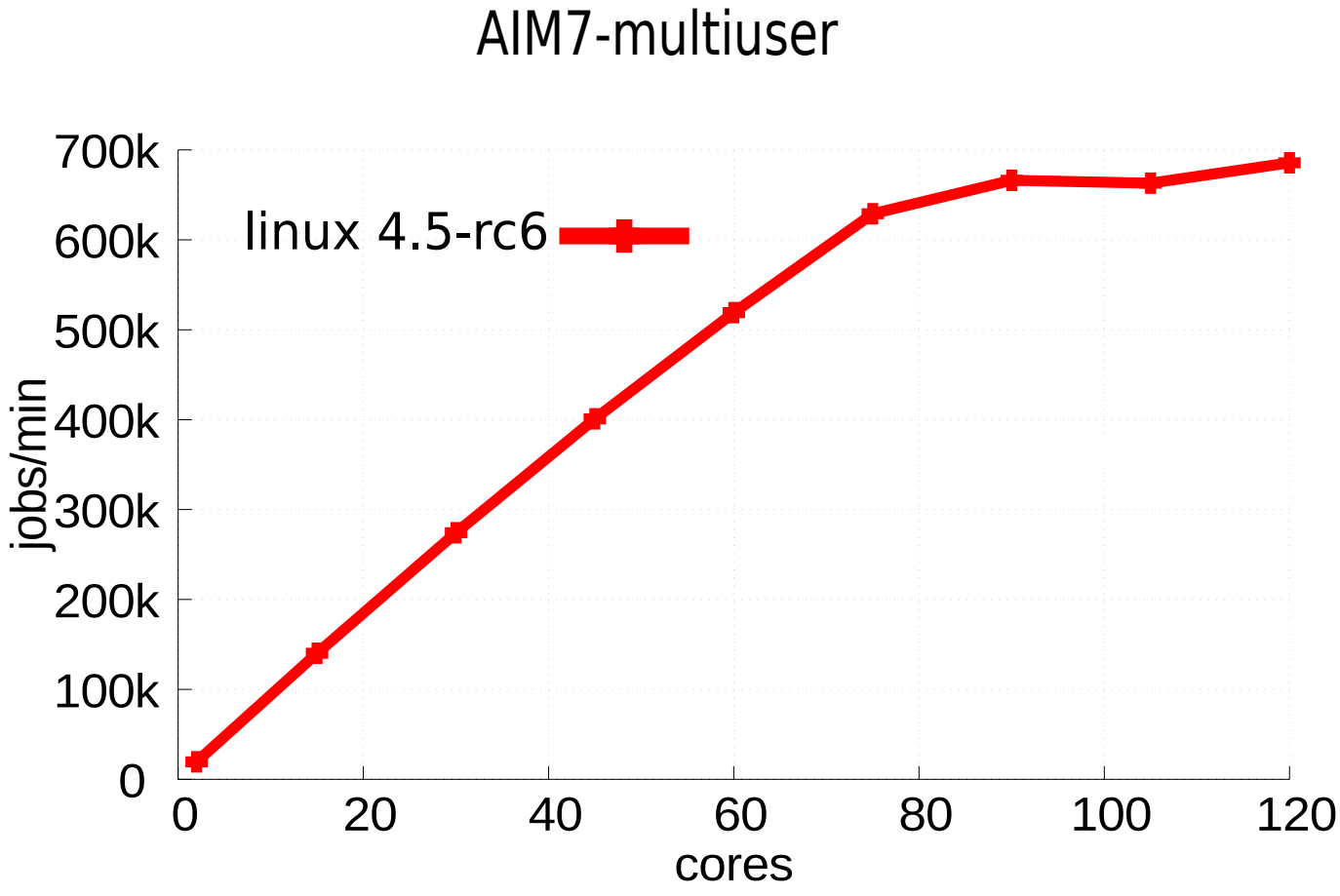
OS Kernel Scalability History



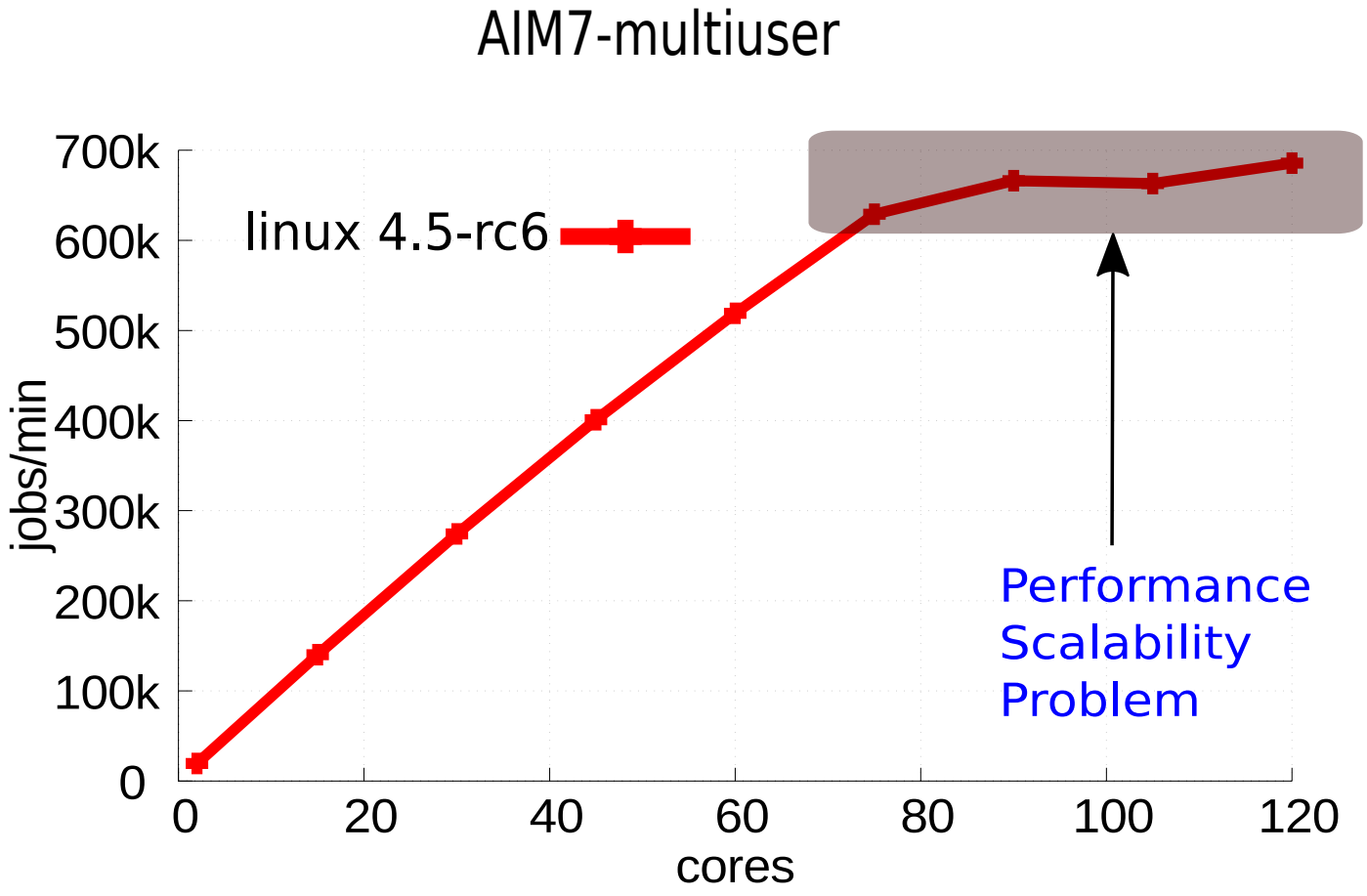
OS Kernel Scalability History



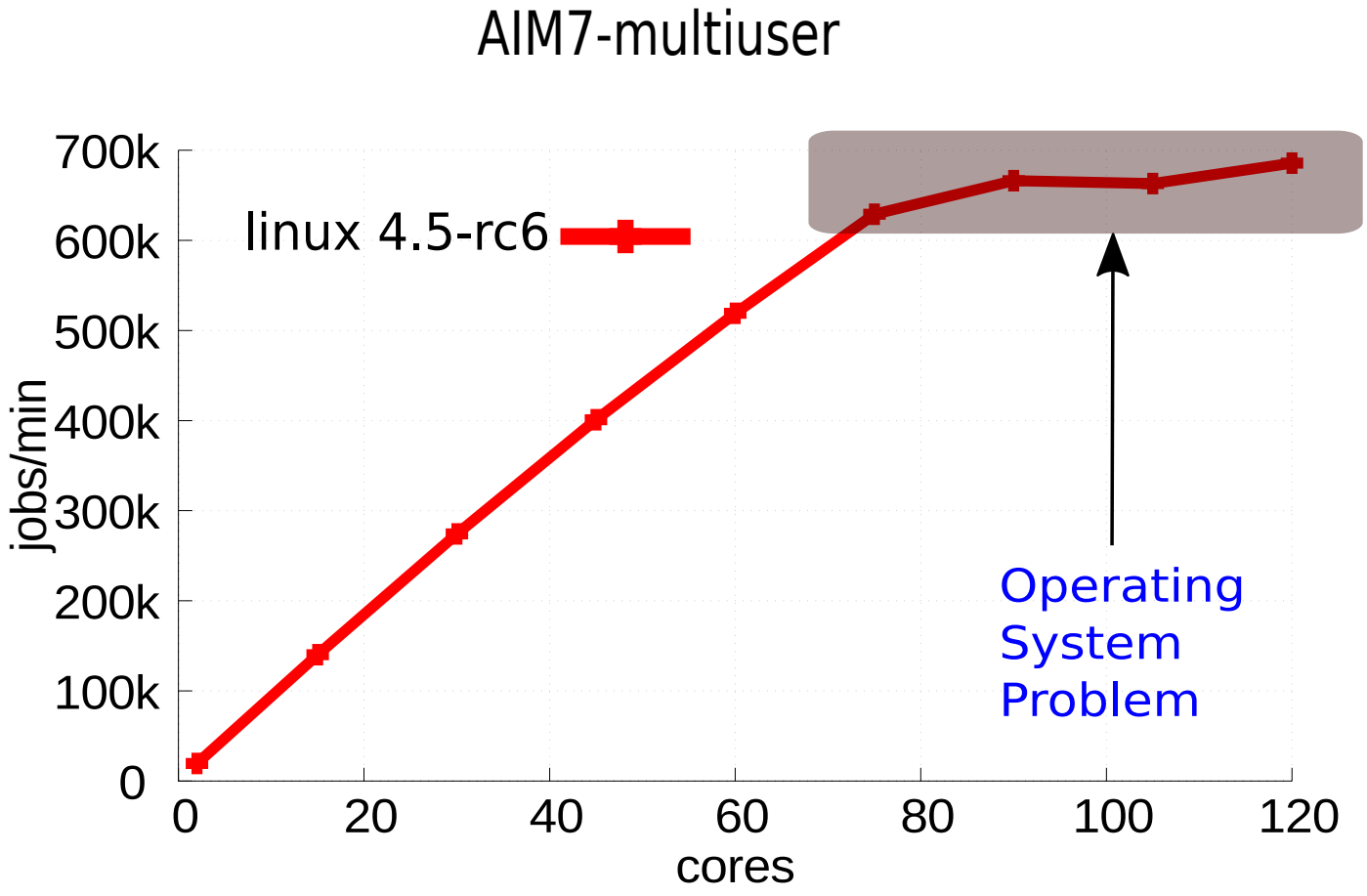
Performance Scalability



Performance Scalability



Performance Scalability



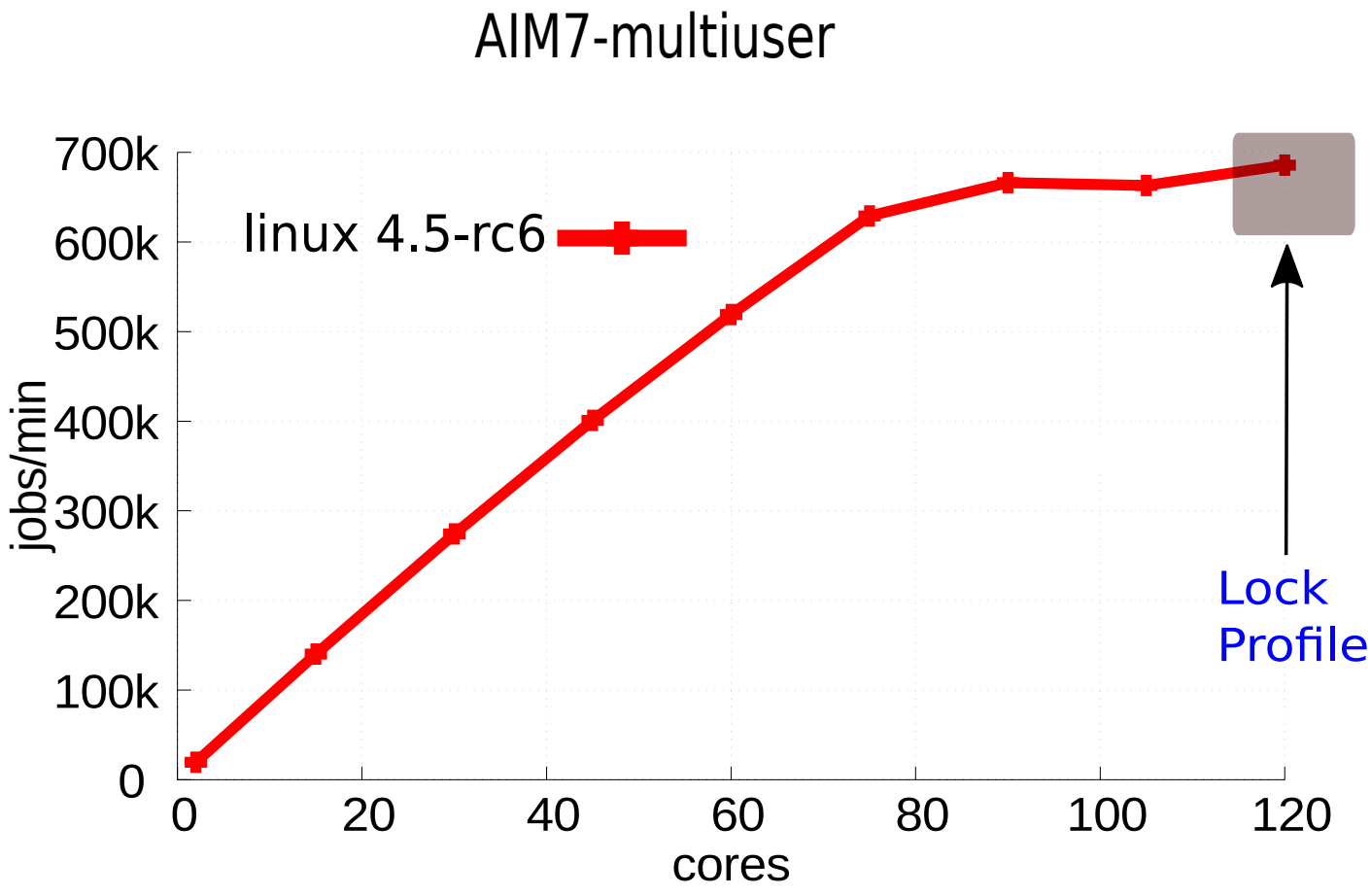
OS Kernel Scalability

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

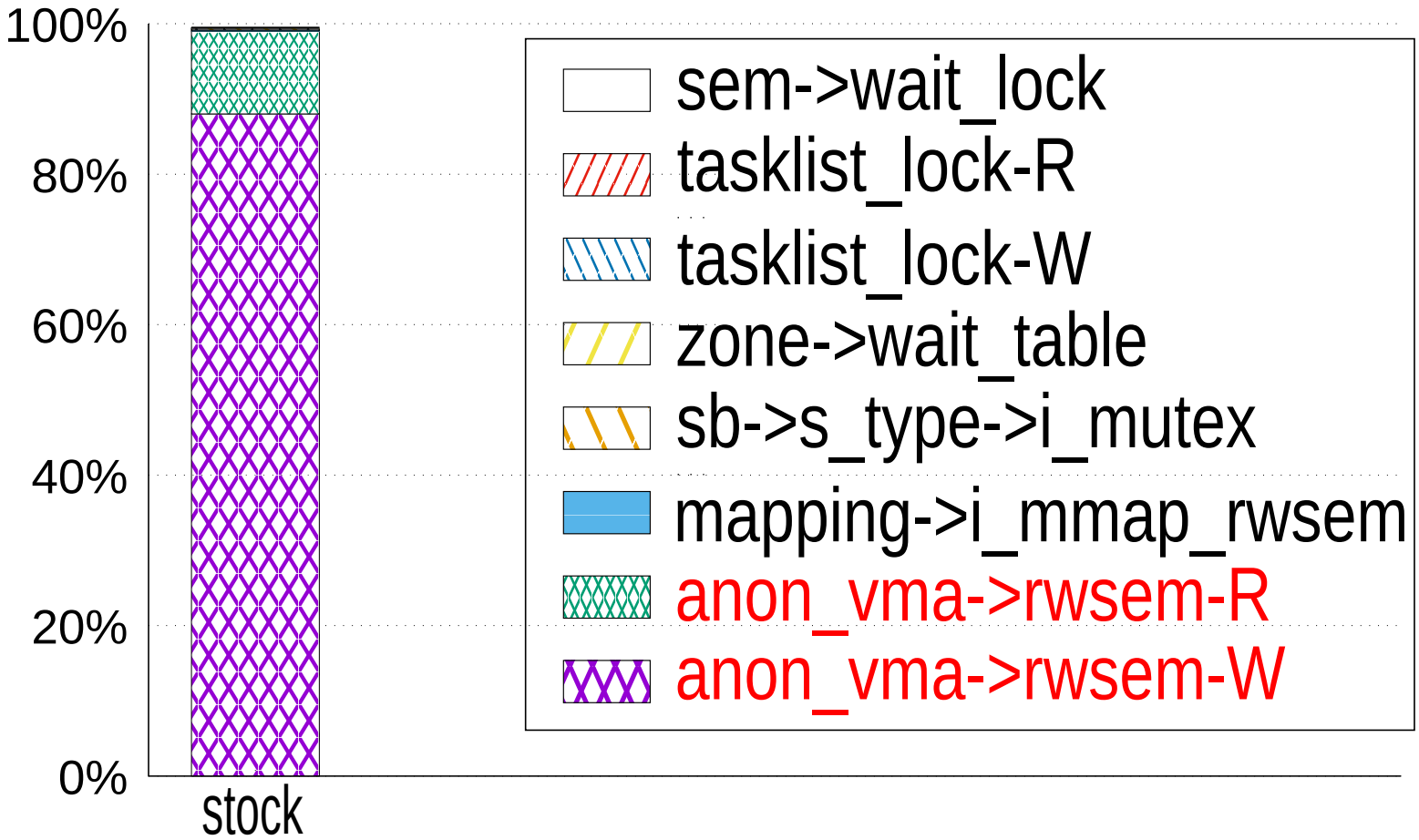
OS Kernel Scalability

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

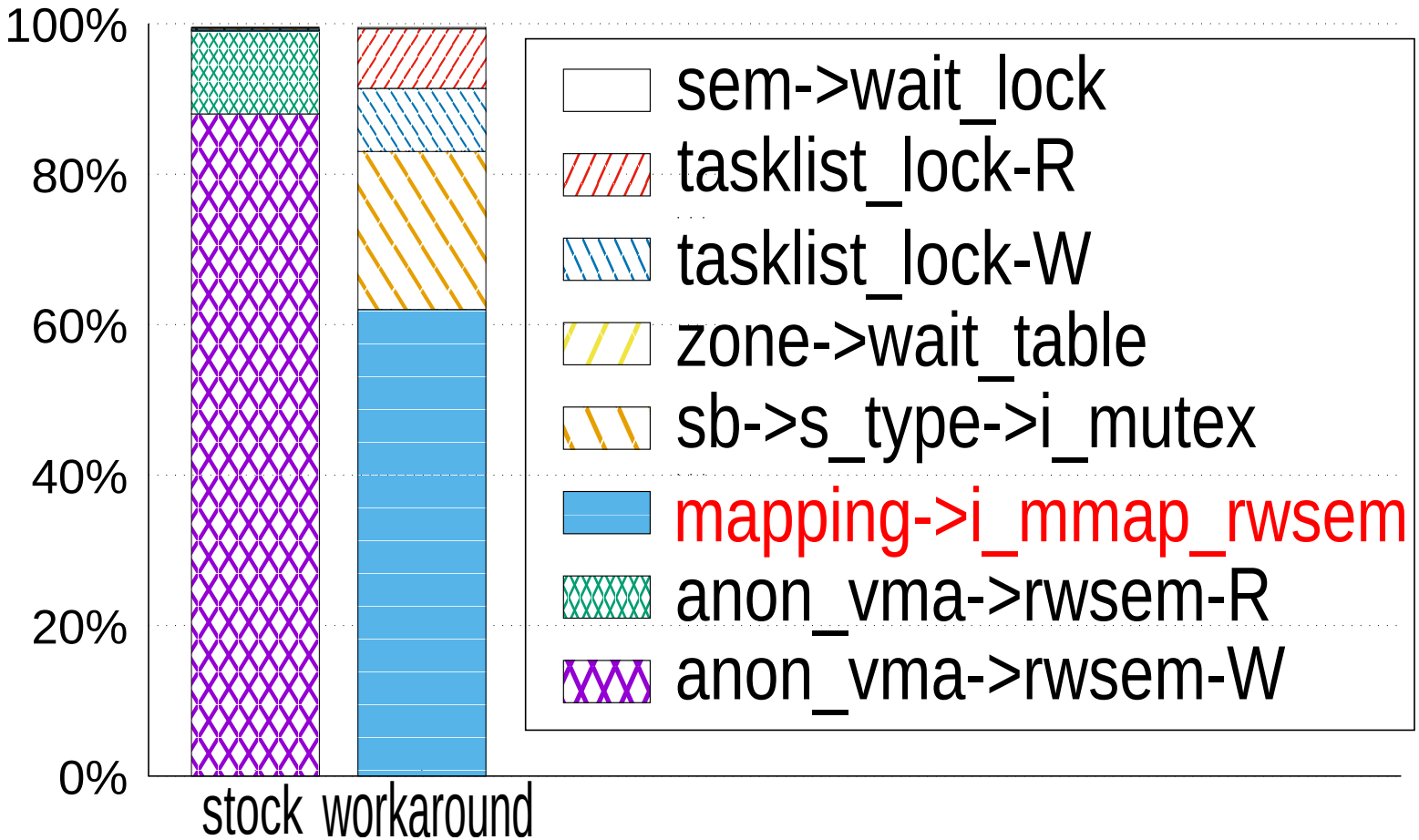
Lock Profile on 120 core



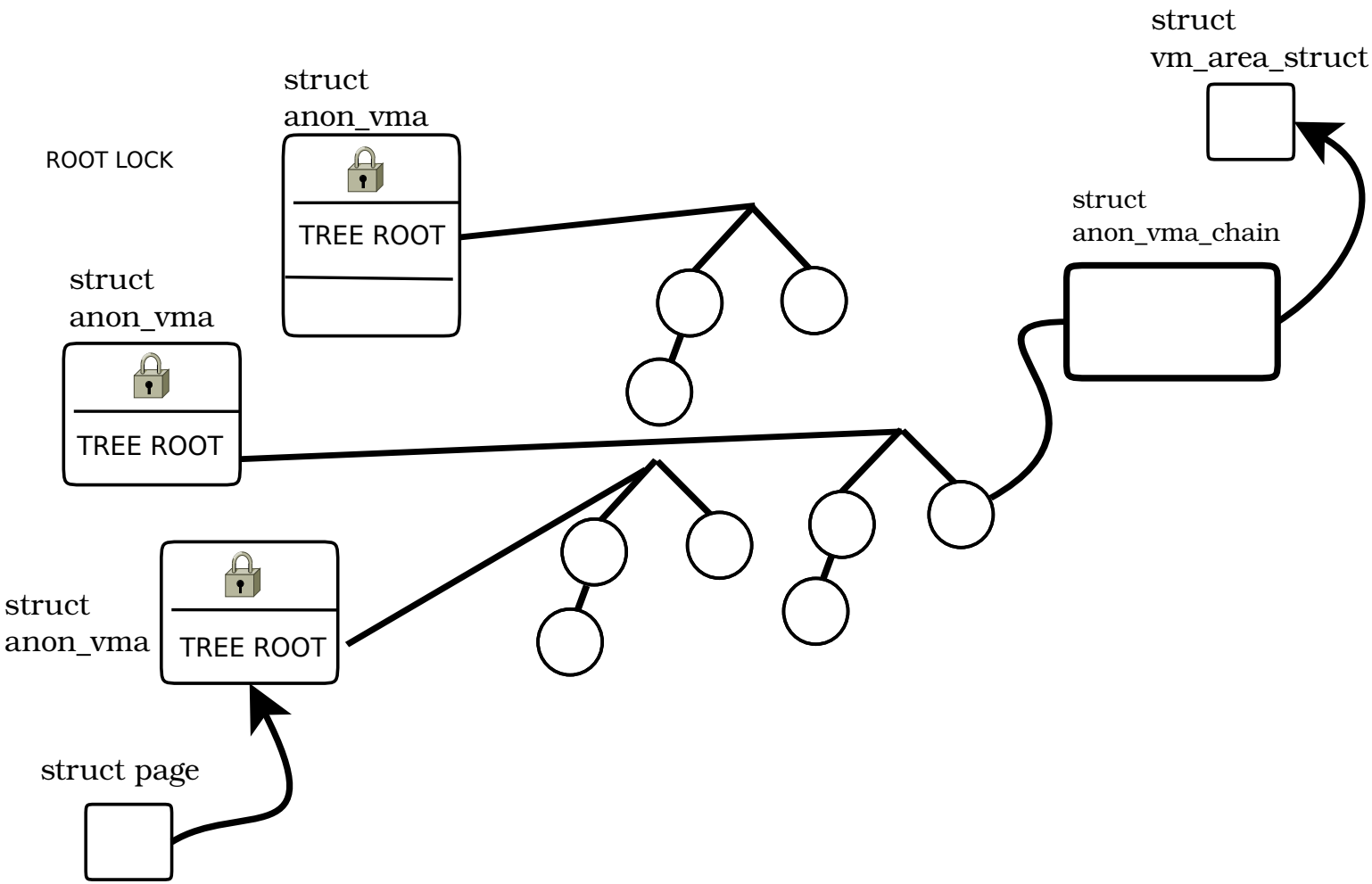
Wait time to acquire the lock



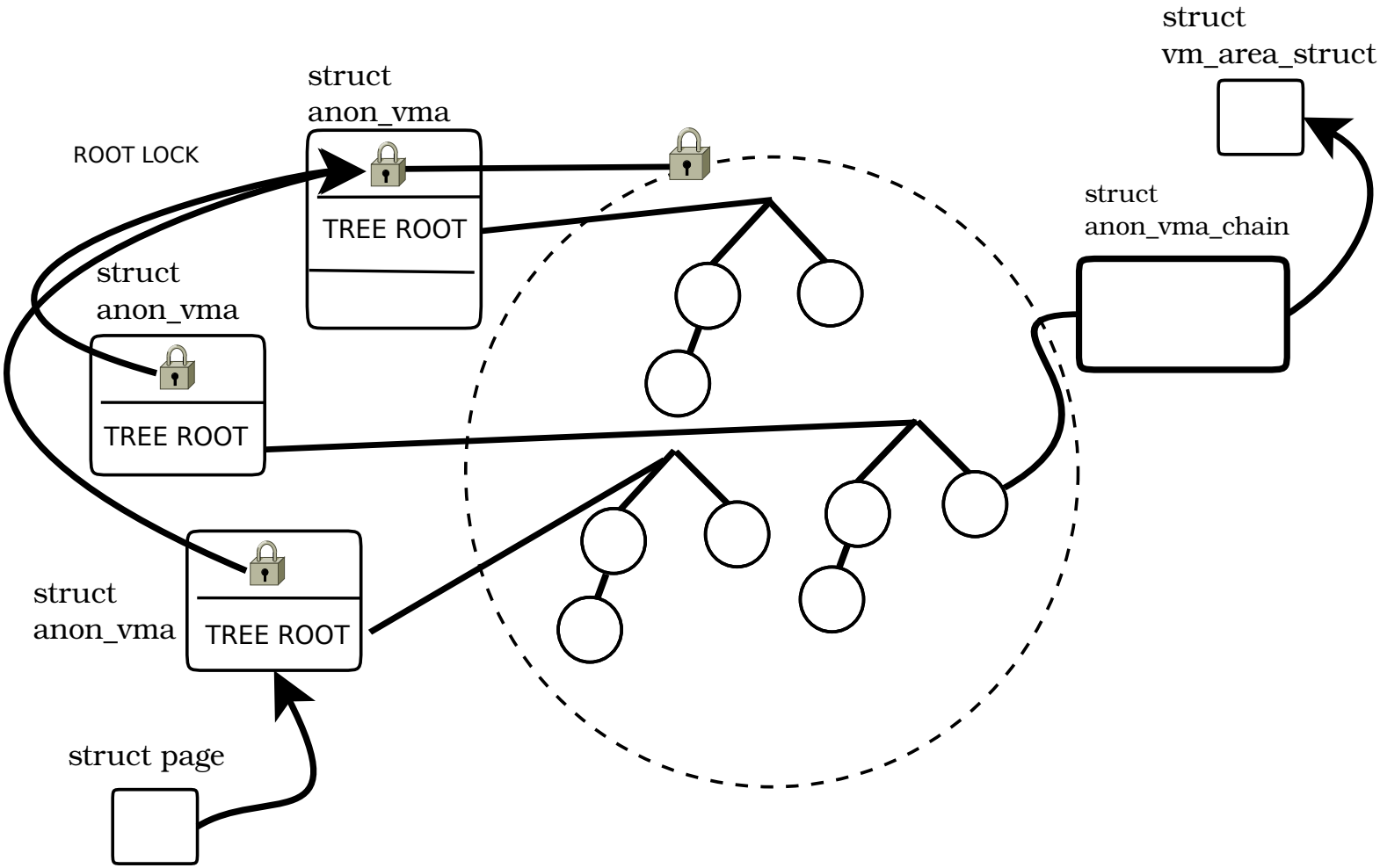
Wait time to acquire the lock



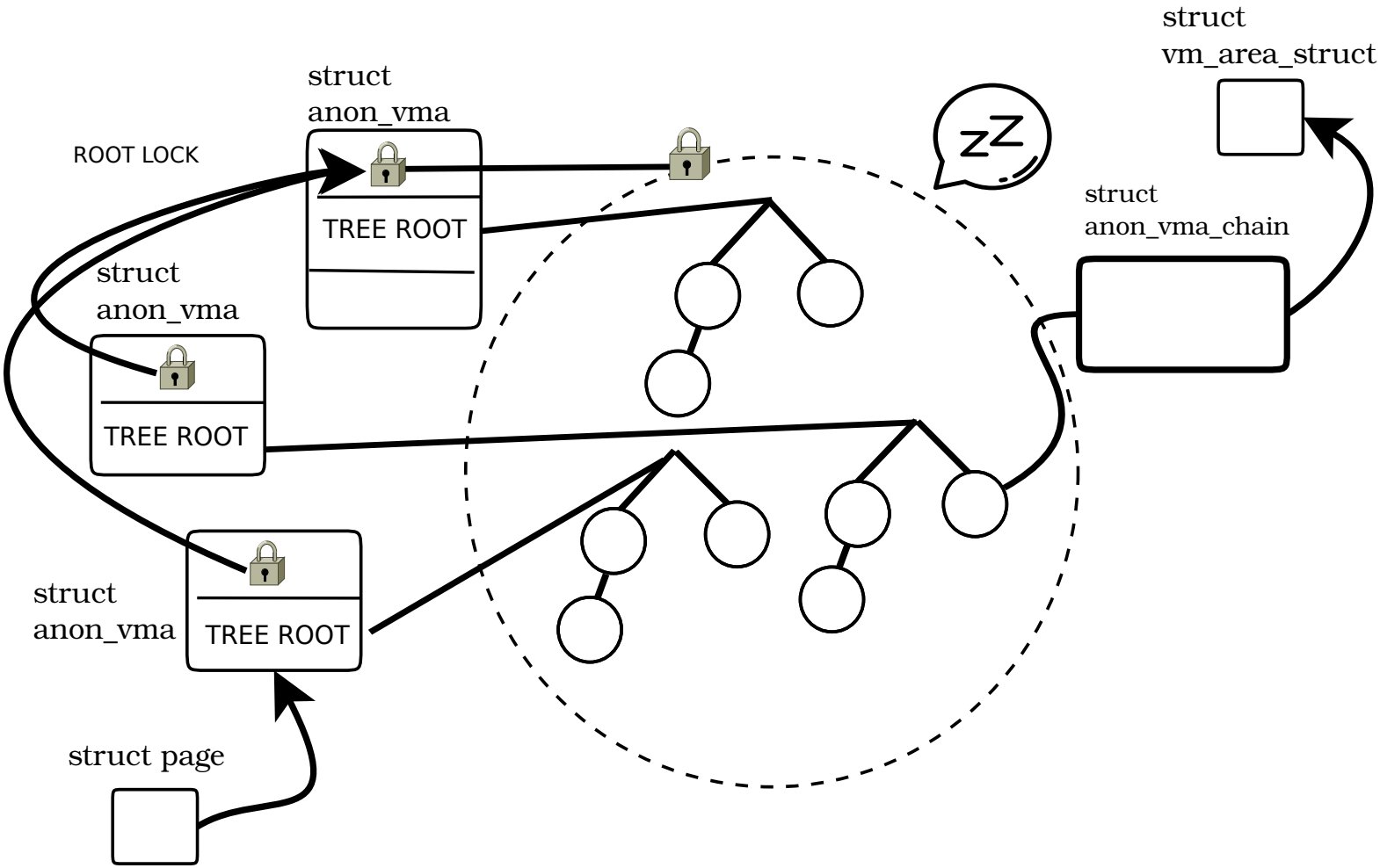
Anonymous reverse mapping



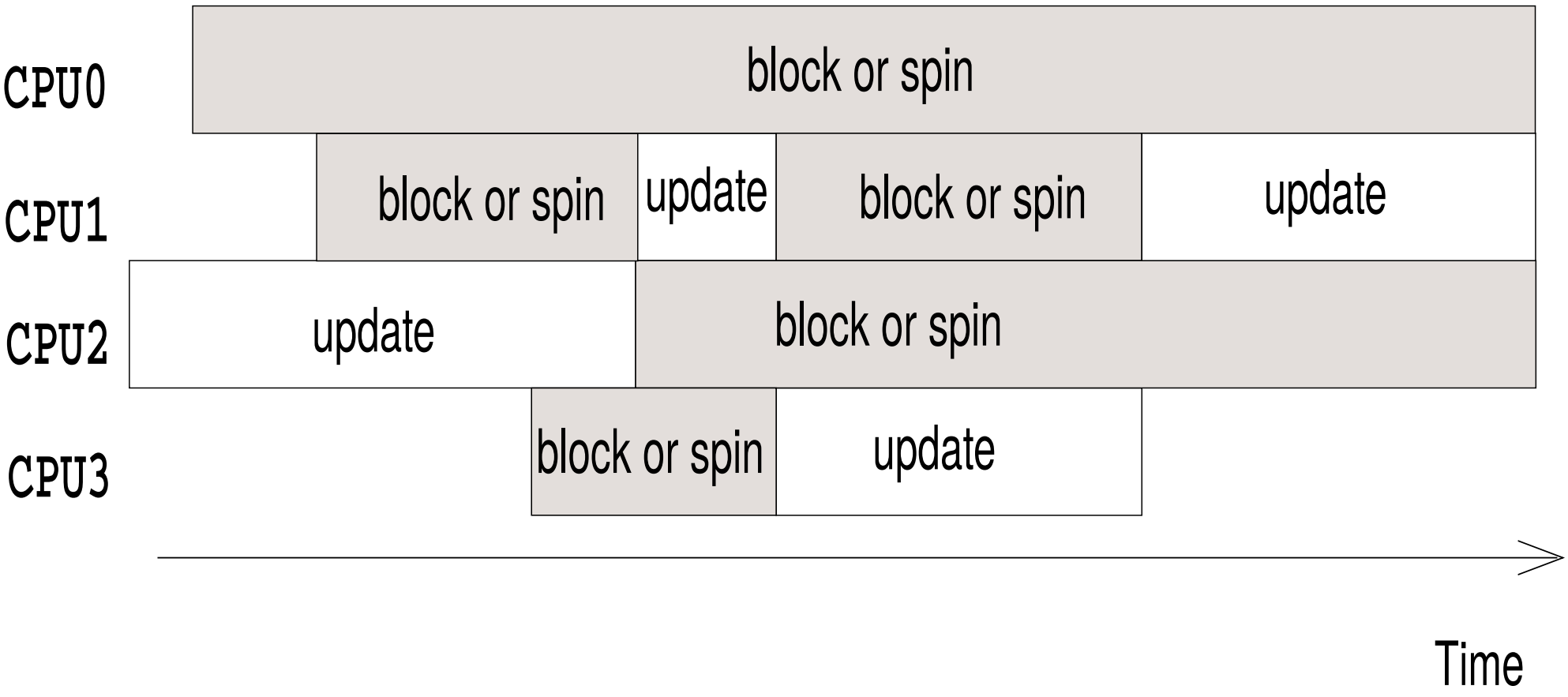
Anonymous reverse mapping



Anonymous reverse mapping



Update Serialization



Solution for High Update Rate Data Structure

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

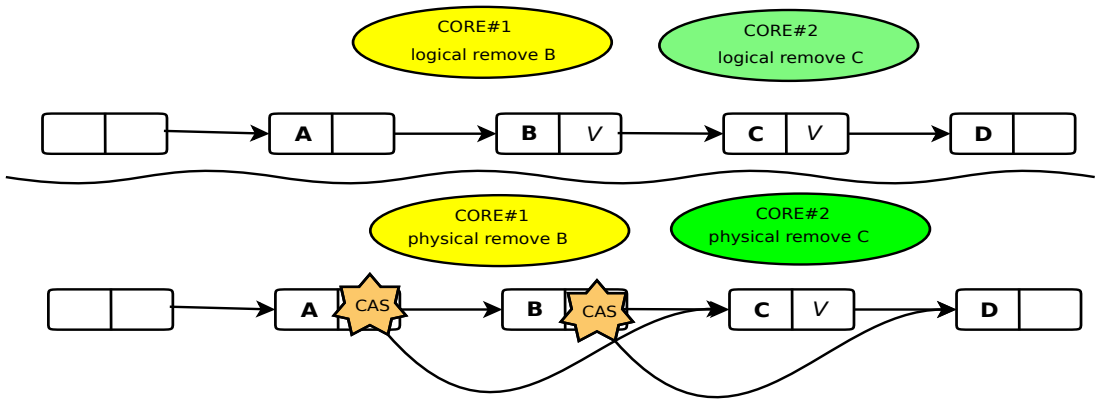
Solution for High Update Rate Data Structure

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

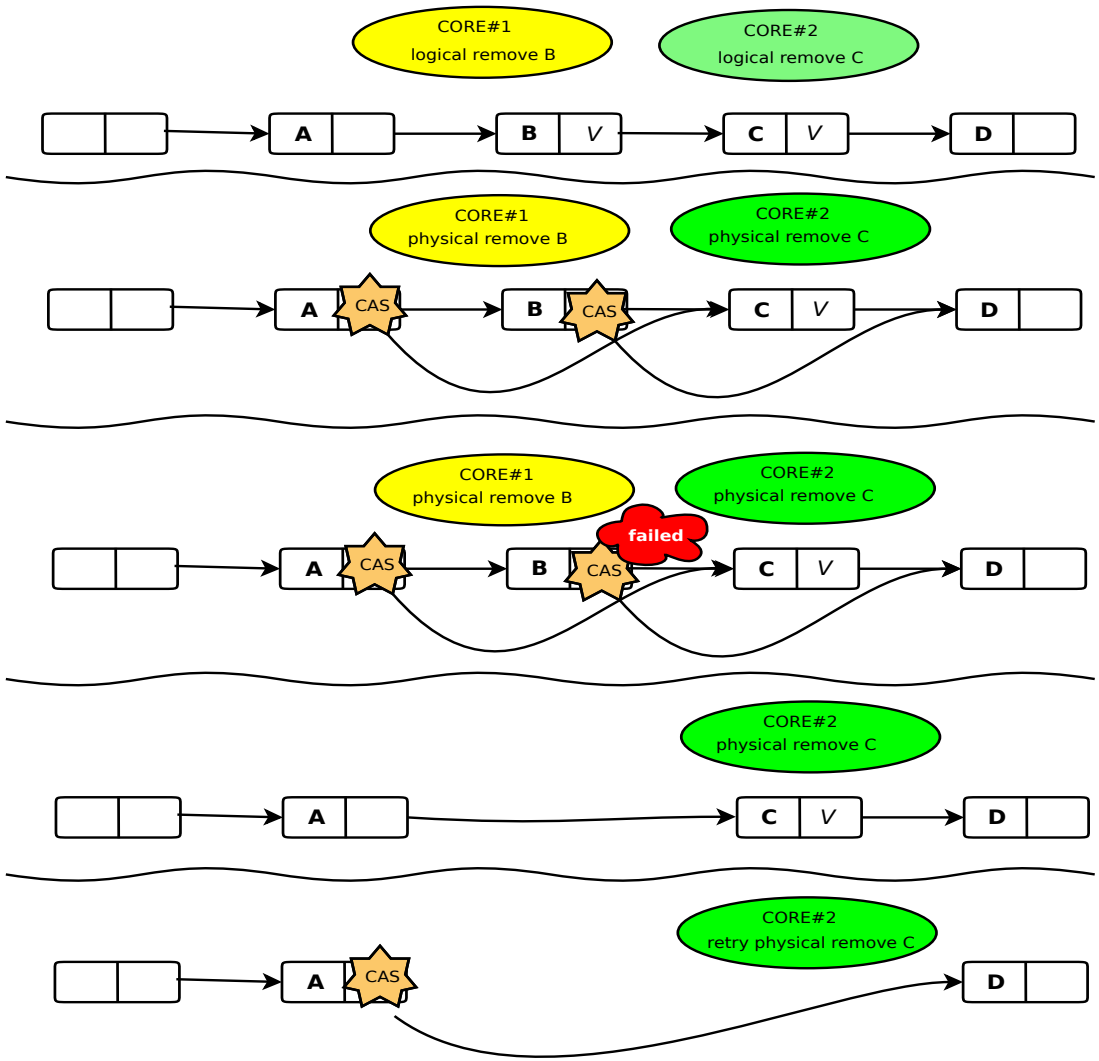
Solution for High Update Rate Data Structure

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

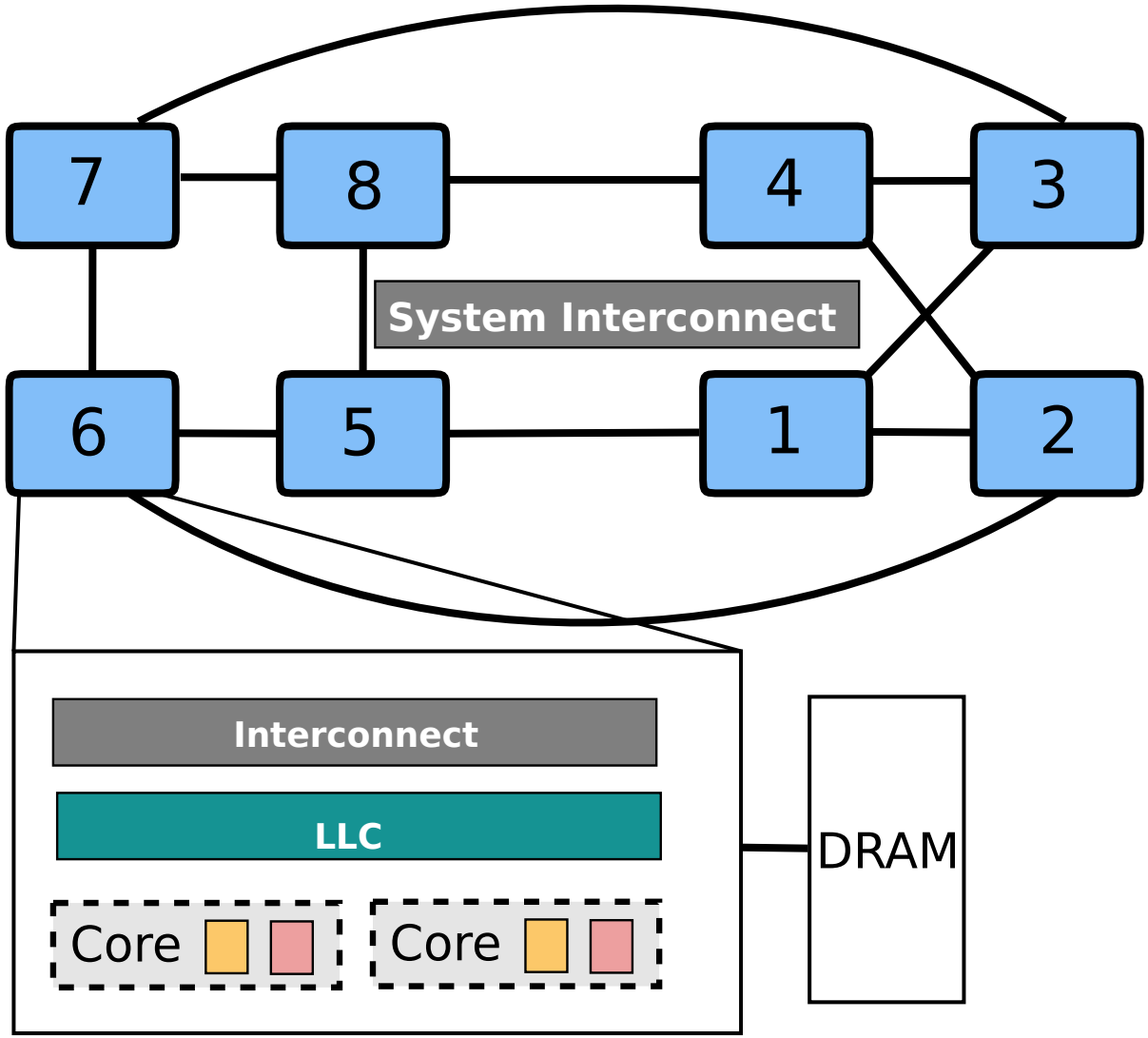
Non-blocking Data Structure



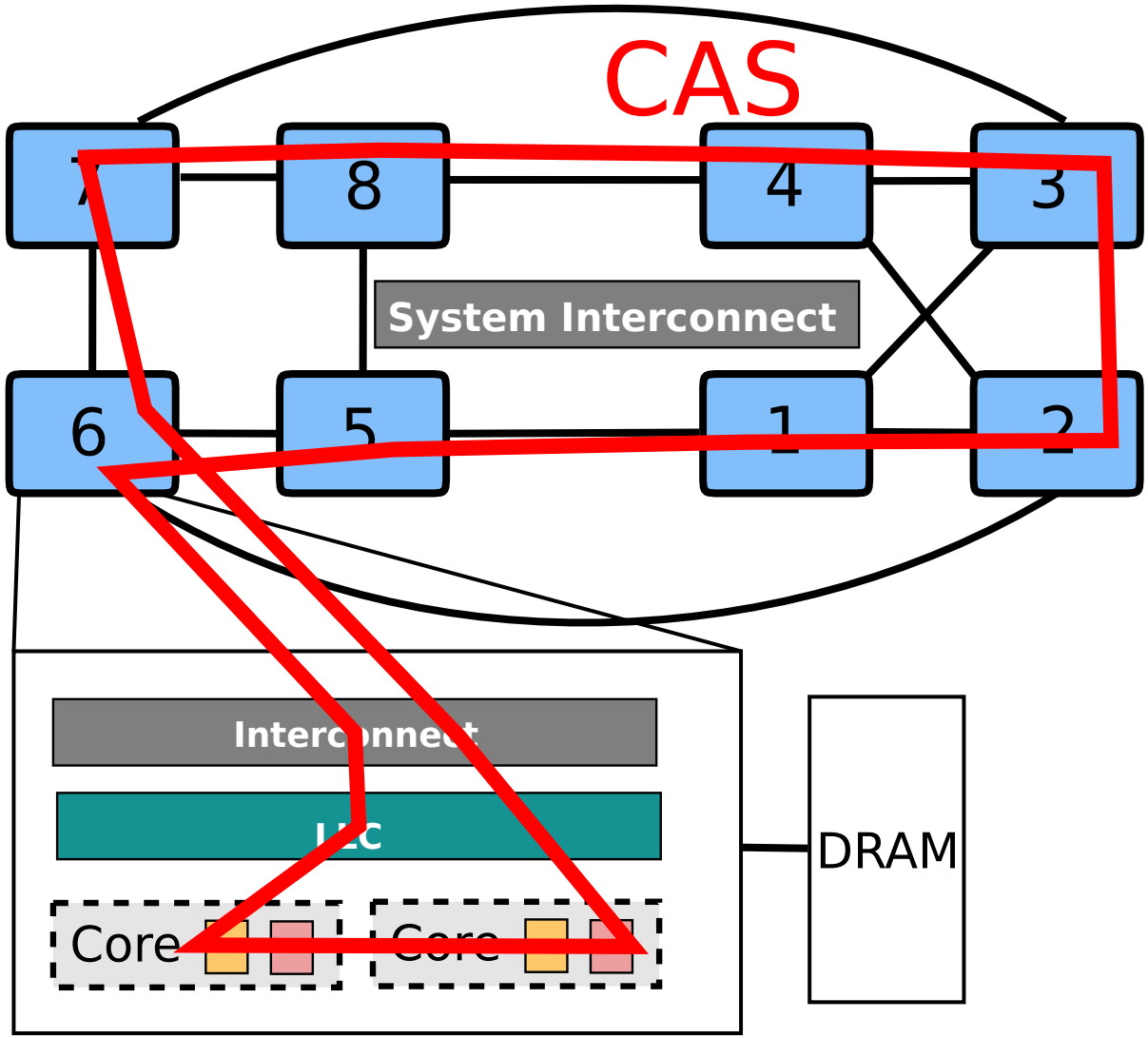
Non-blocking Data Structure



Cache communication bottlenecks



Cache communication bottlenecks



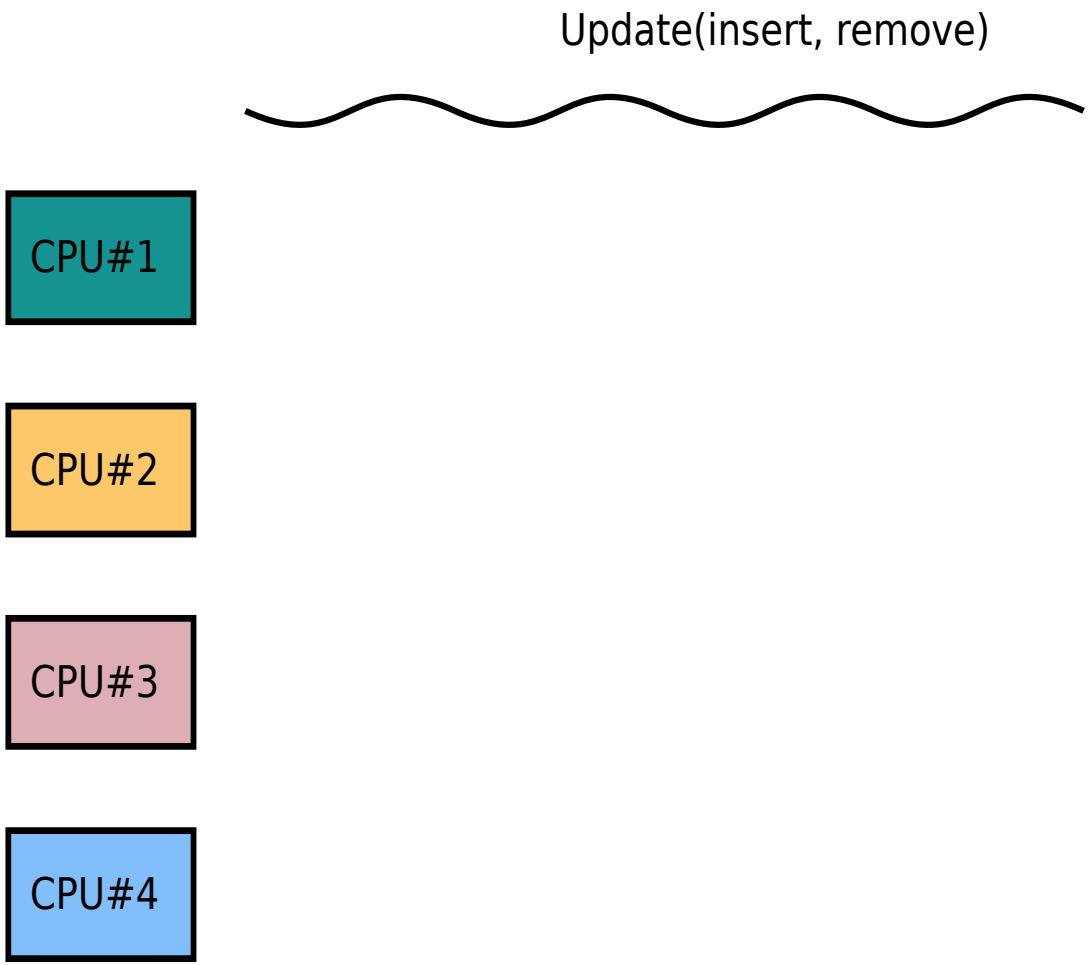
Log-based Algorithms

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

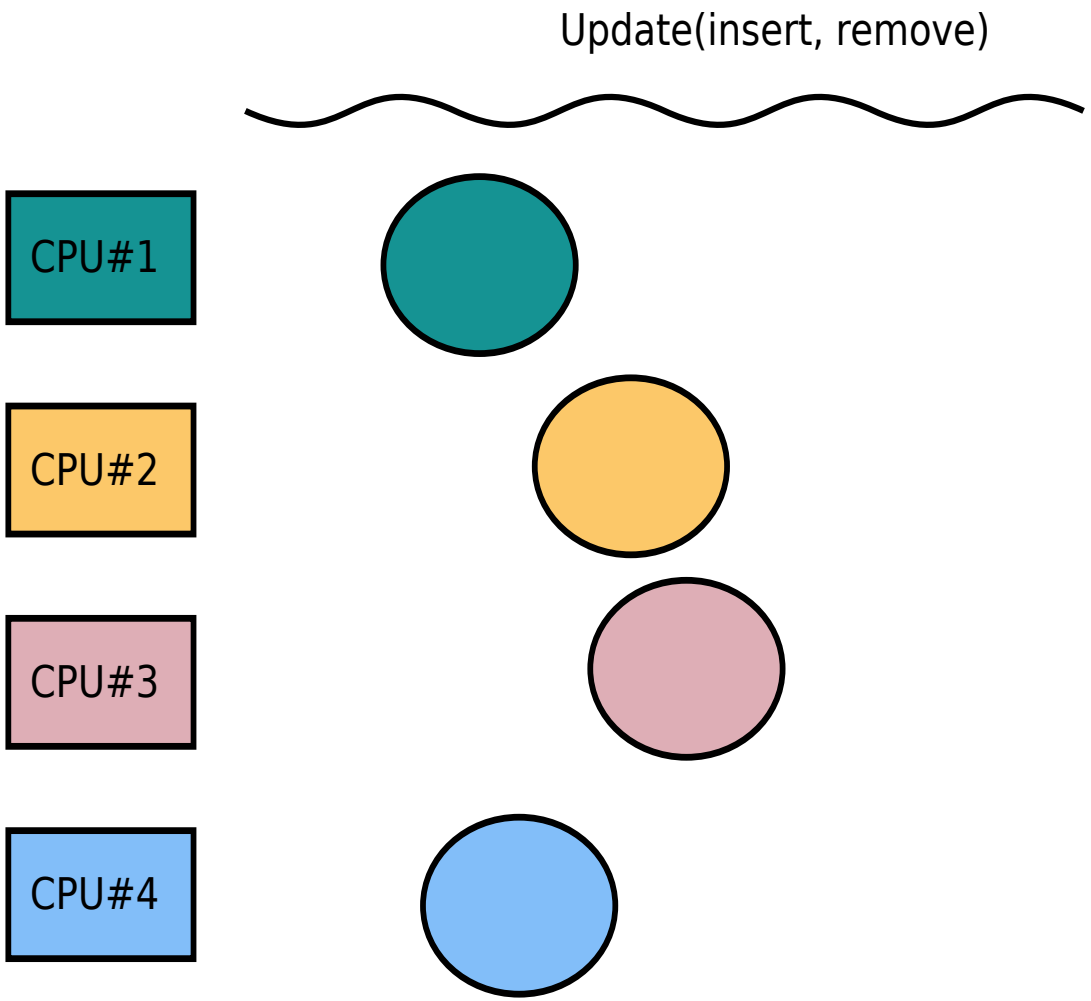
Log-based Algorithms

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

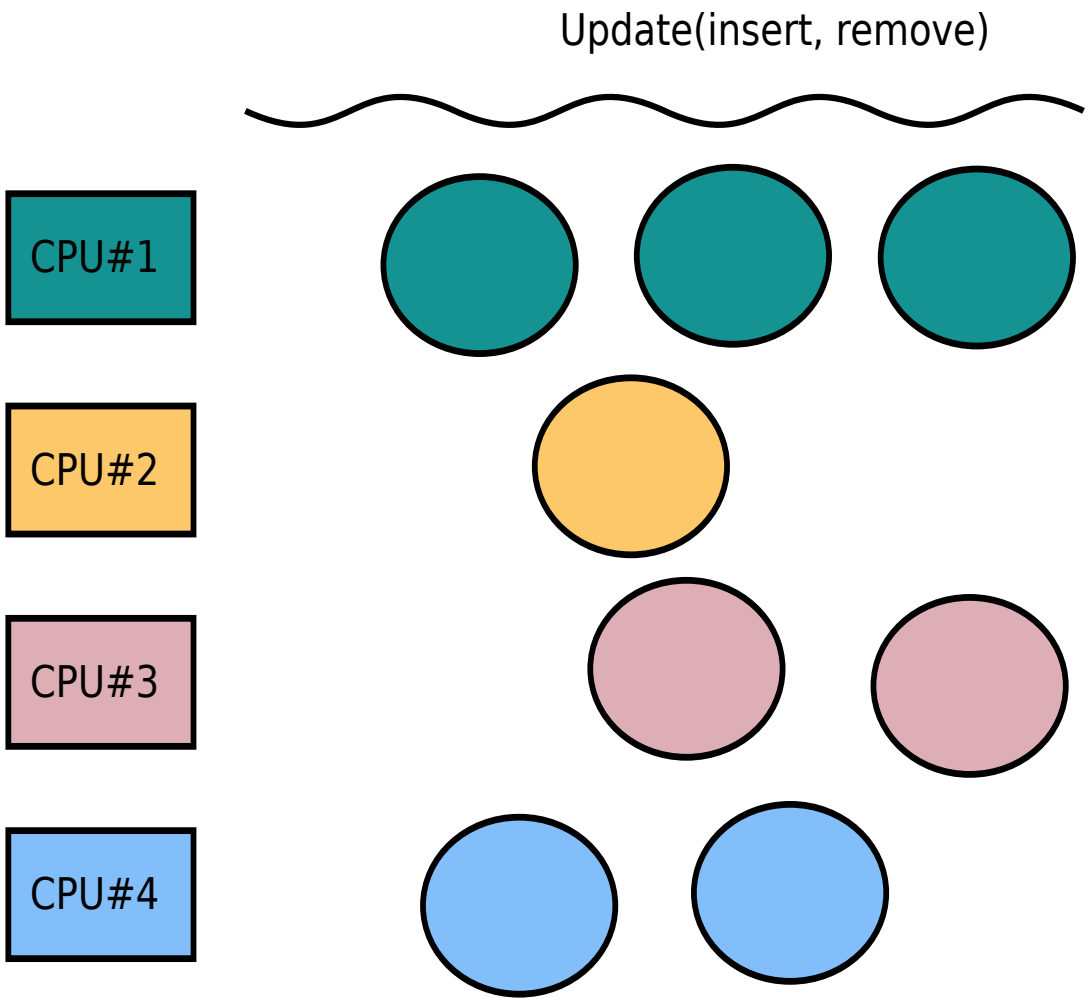
Log-based Algorithms: OpLog



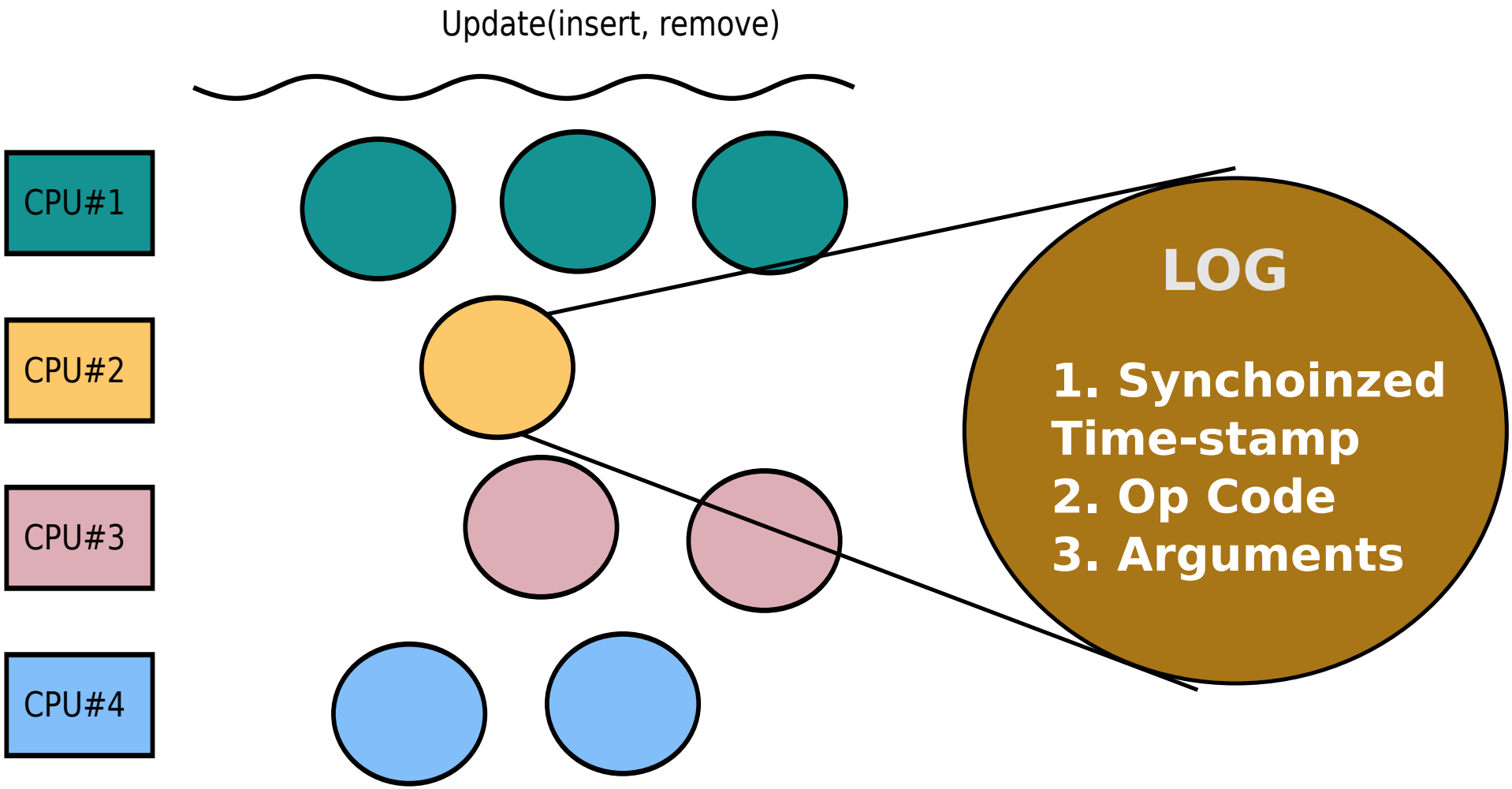
Log-based Algorithms: OpLog



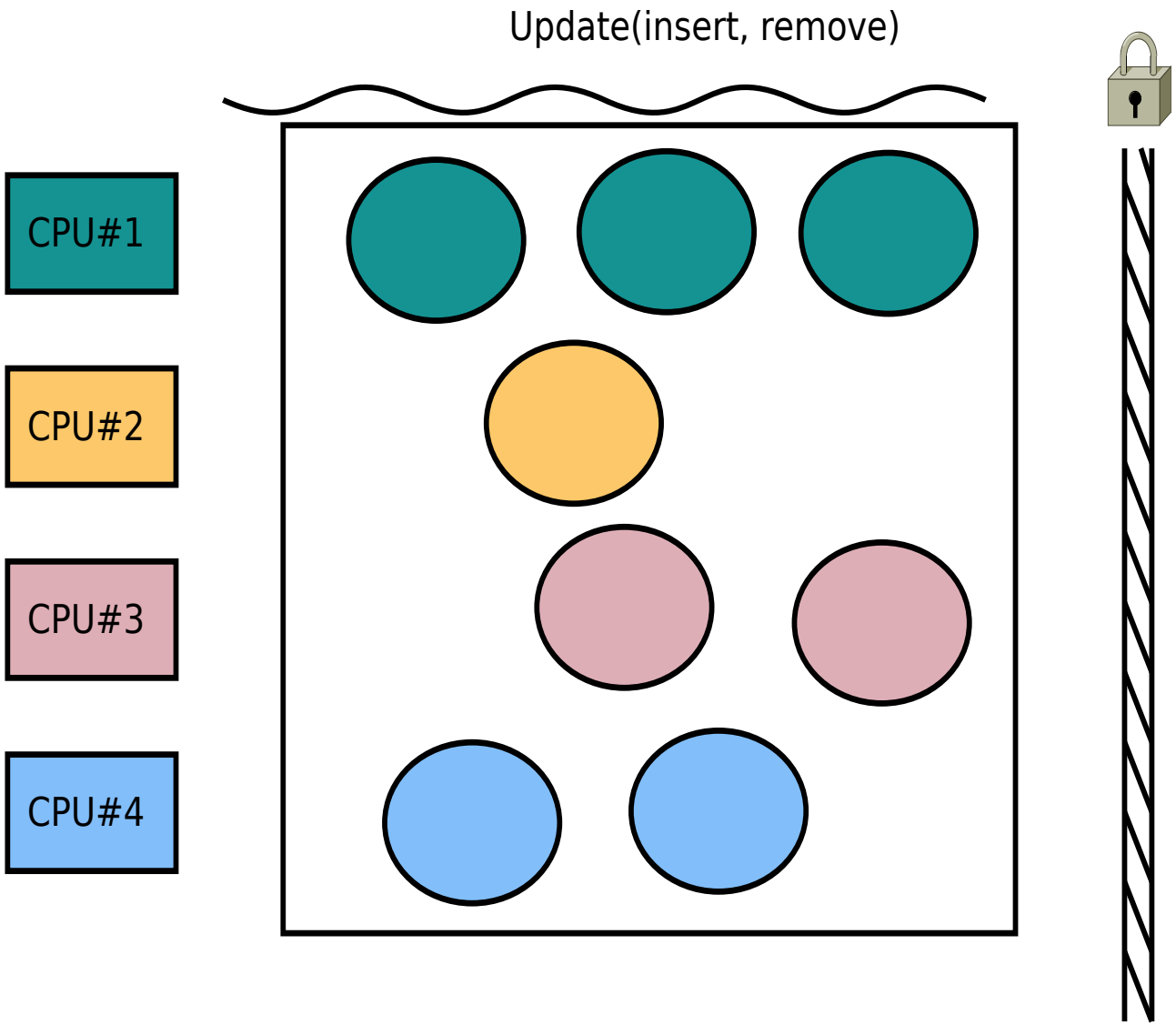
Log-based Algorithms: OpLog



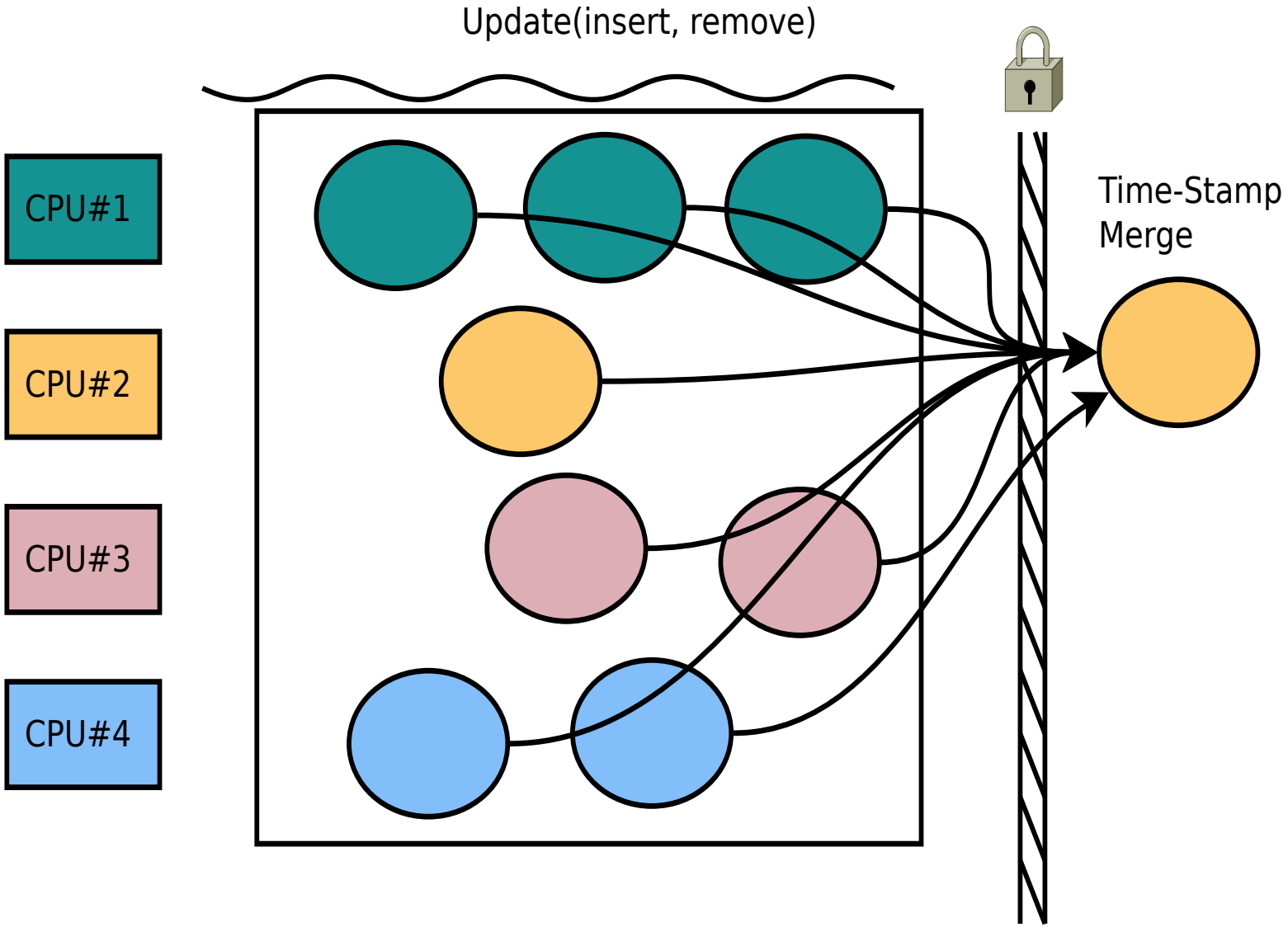
Log-based Algorithms: OpLog



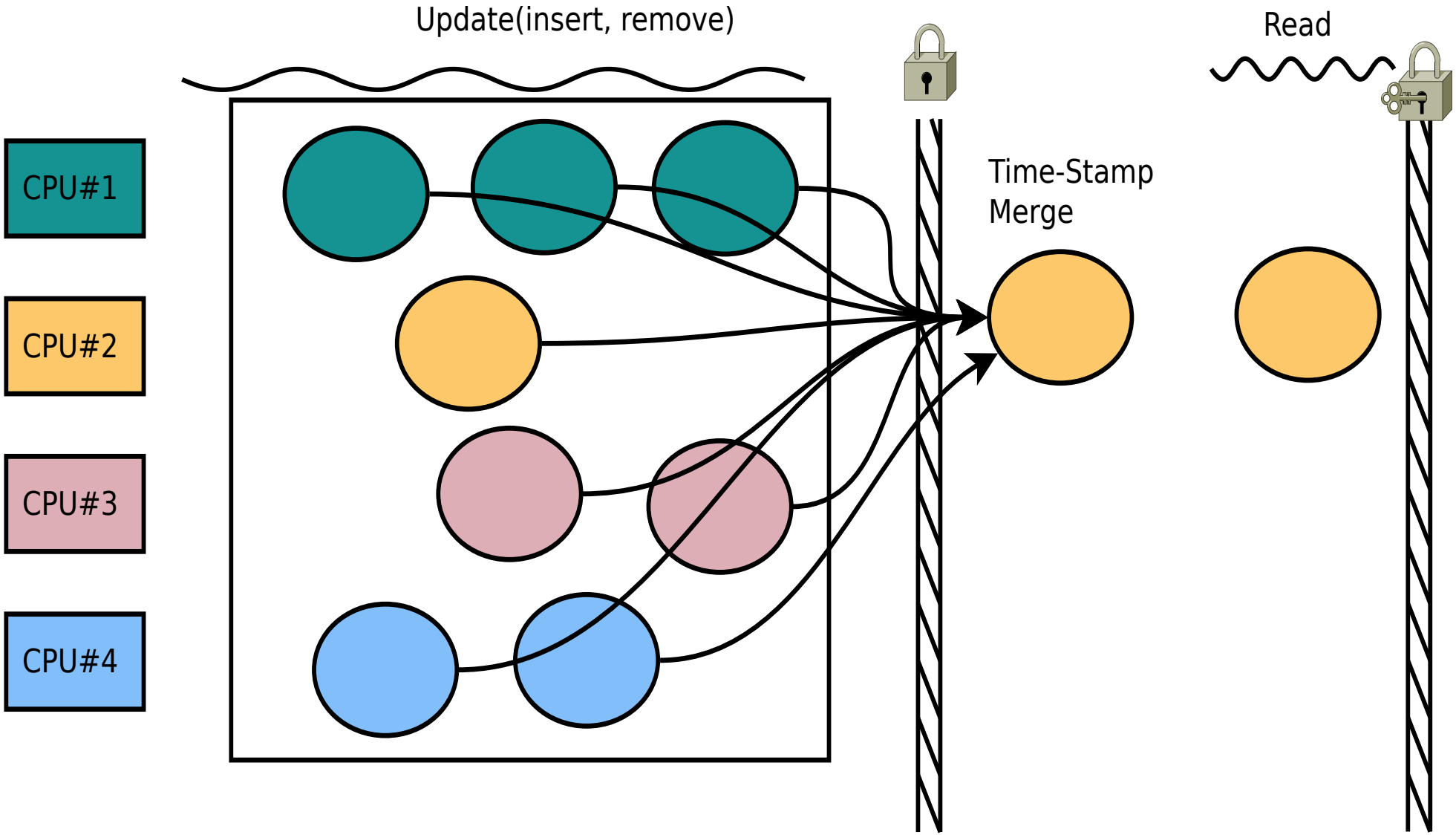
Log-based Algorithms: OpLog



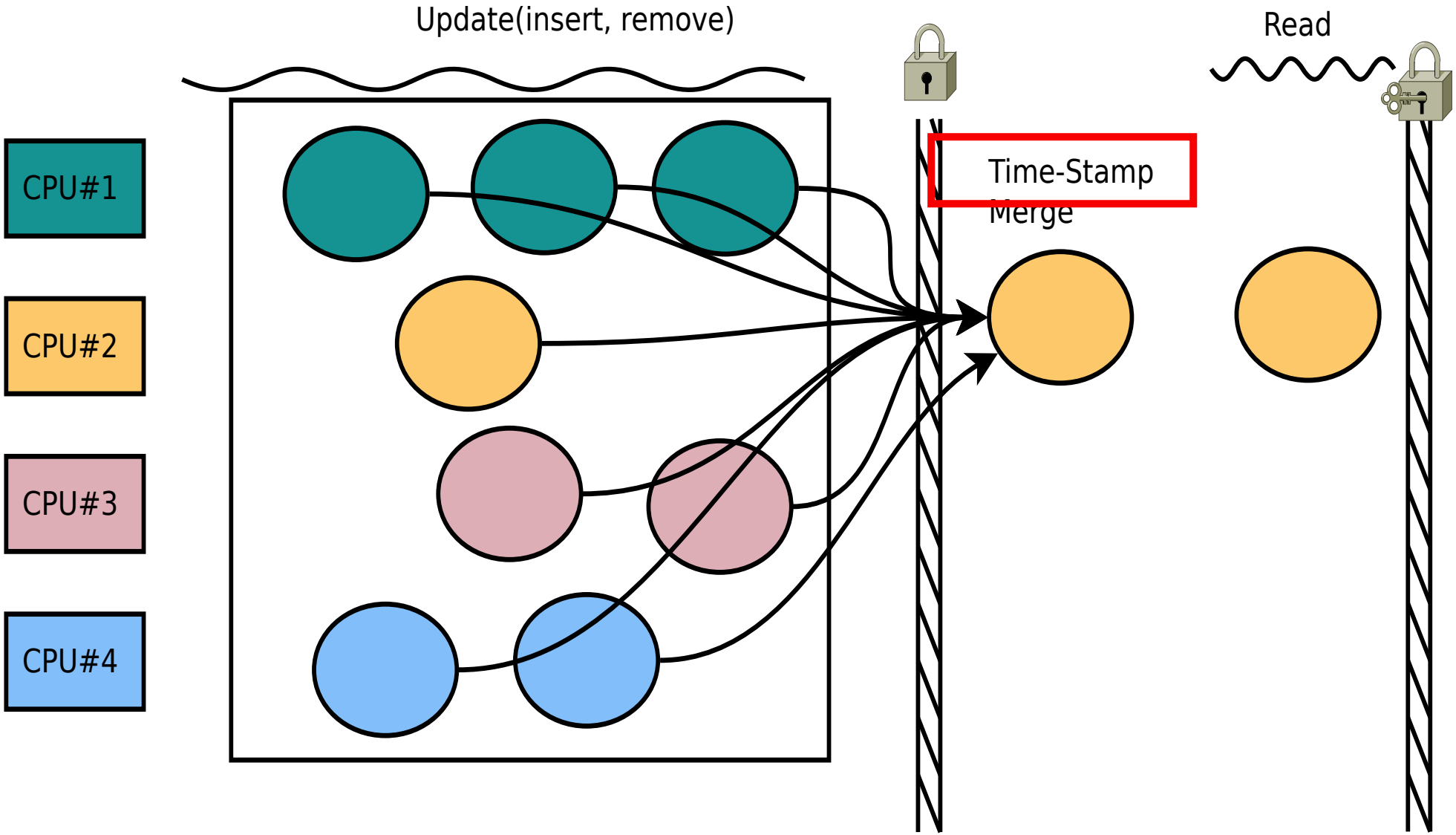
Log-based Algorithms: OpLog



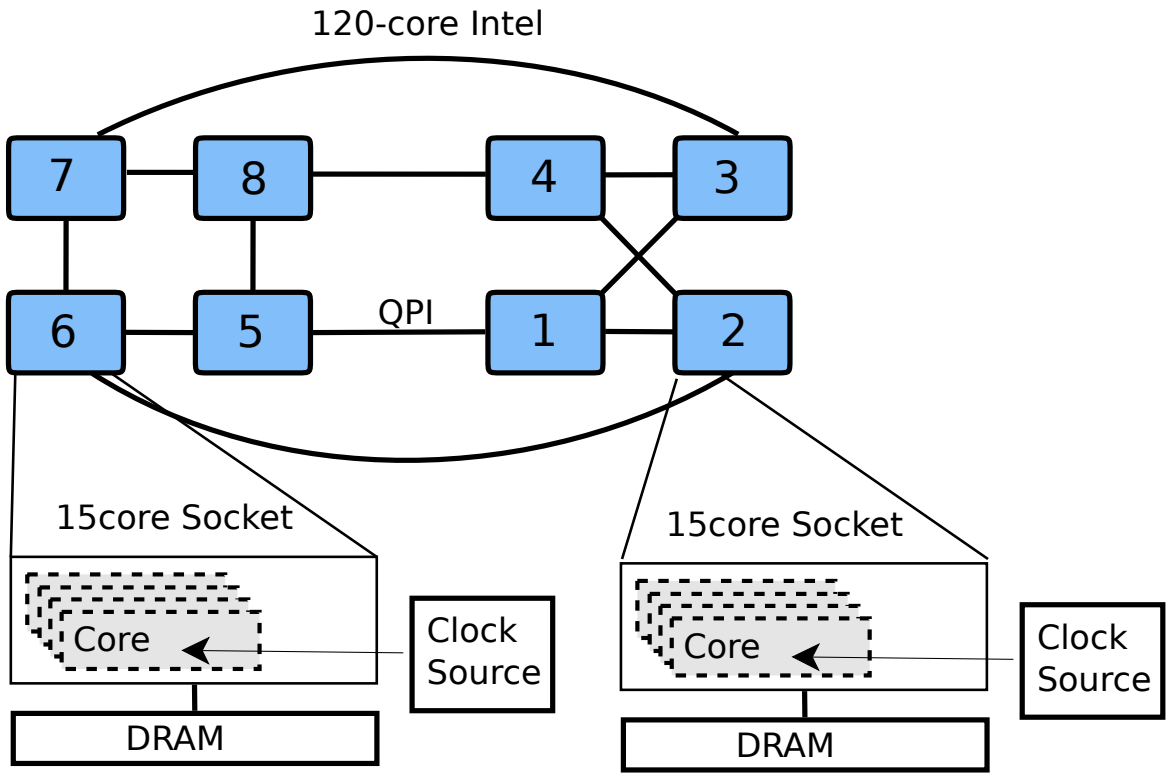
Log-based Algorithms: OpLog



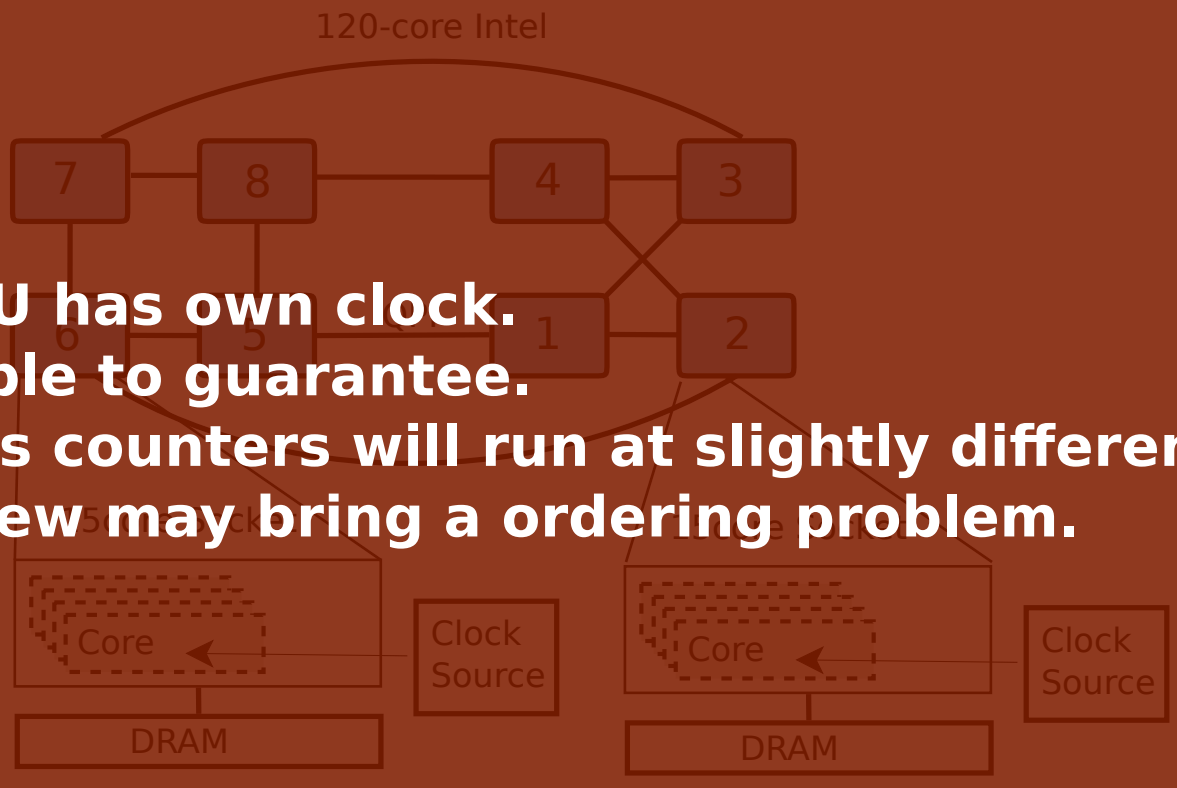
Log-based Algorithms: OpLog



Time-stamp counters



Time-stamp counters



Each CPU has own clock.
Impossible to guarantee.
All clocks counters will run at slightly different rates.
Clock skew may bring a ordering problem.

Simple Solution?

Simple Solution

- A log-based deferred update +
- Minimal hardware synchronization(Lightweight) =
- A Lightweight log-based Deferred Update(LDU).

Simple Solution

- A log-based deferred update +
- Minimal hardware synchronization(Lightweight) =
- A Lightweight log-based Deferred Update(LDU).

Simple Solution

- A log-based deferred update +
- Minimal hardware synchronization(Lightweight) =
- 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

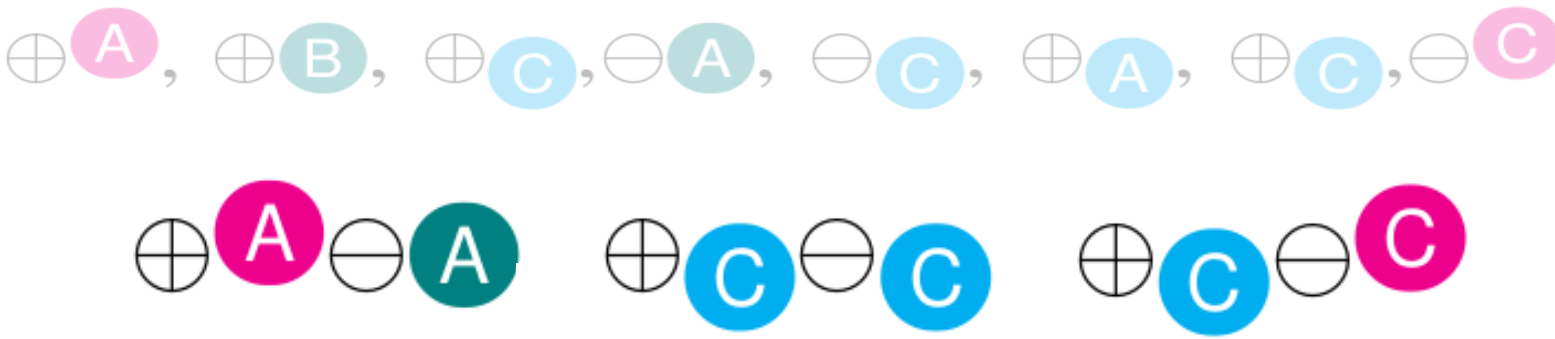
- Design
  - Approach
  - Example
- Applying the Linux kernel
- Implementation
- Evaluation

Log Example

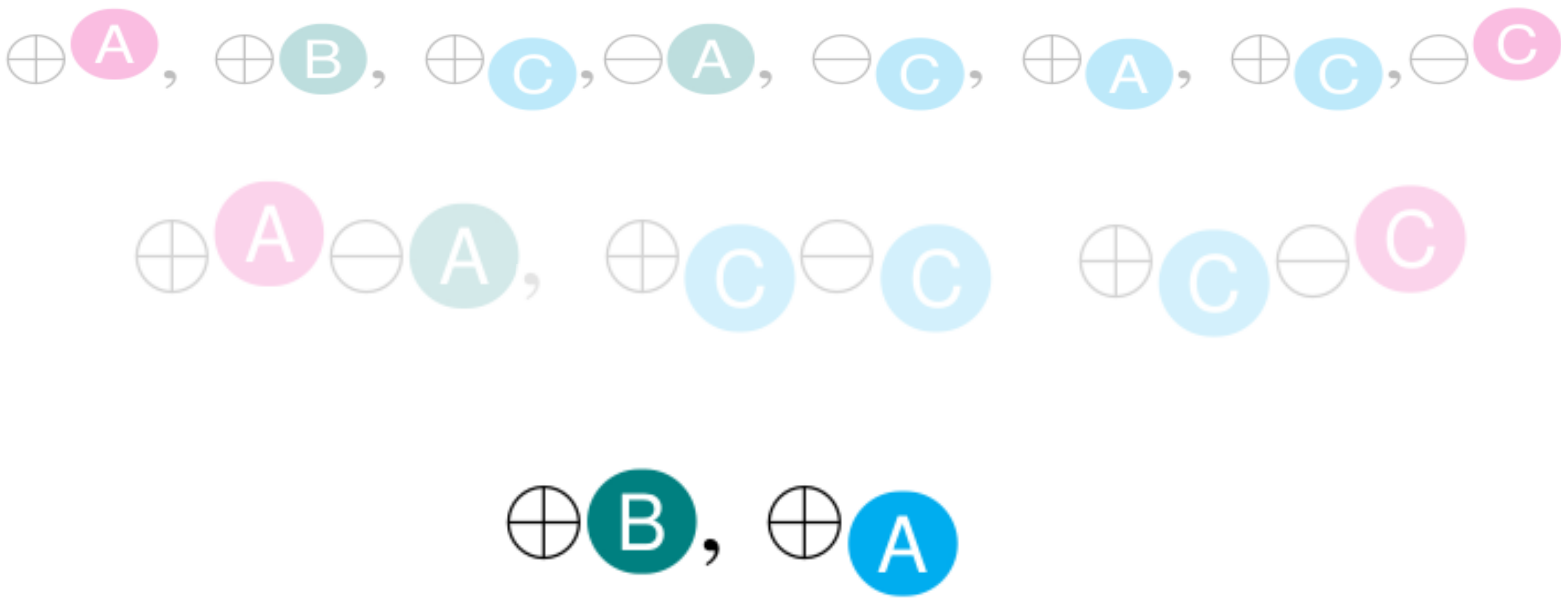
Approach

"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.

Cancelable – Log



Remain Log



Scaling despite non-commutativity – LWN

- Permit weak ordering.

Approach

LWN.net

News from the source

Overview of the scalable co

February 17, 2015

This article was contributed by Paul McKenney

We all learned about the comm
the same value as Y + X, for a
commutativity rule uses a mo
encompasses the order of exe
arguments to a single operato
atomically adds the value X to variable A at about the same time that another CPU atomically adds the value Y to this same variable. Because addition is commutative in this more concurrent sense, we know that regardless of the order of execution, the overall effect will be to add X + Y to A.

The key insight behind the scalable commutativity rule is that if the ordering of a pair of operations is irrelevant according to the API, then it should be possible to implement those operations so as to avoid scalability bottlenecks. In contrast, if the ordering of the two operations is critically important, then it is most likely impossible to avoid the bottlenecks between these operations.

For example, suppose that the two operations are inserting two objects, each with its own key, into a search data structure not already containing elements with those keys. Suppose further that this structure does not allow duplicate keys. No matter which of the two operations executes first, the result is the same: both objects are successfully inserted. Therefore, the scalable commutativity rule holds that a scalable implementation is possible, and there is, in fact, a wealth of scalable implementations of search structures, ranging from hash tables to radix trees to dense arrays.

Approach

LWN.net

News from the source

Overview of the scalable co

February 17, 2015

This article was contributed by Paul McKenney

We all learned about the comm
the same value as Y + X, for a
commutativity rule uses a mo
encompasses the order of exe
arguments to a single operato
atomically adds the value X to variable A at about the same time that another CPU atomically adds the value Y to this same variable. Because addition is commutative in this more concurrent sense, we know that regardless of the order of execution, the overall effect will be to add X + Y to A.

The key insight behind the scalable commutativity rule is that if the ordering of a pair of operations is irrelevant according to the API, then it should be possible to implement those operations so as to avoid scalability bottlenecks. In contrast, if the ordering of the two operations is critically important, then it is most likely impossible to avoid the bottlenecks between these operations.

For example, suppose that the two operations are inserting two objects, each with its own key, into a search data structure not already containing elements with those keys. Suppose further that this structure does not allow duplicate keys. No matter which of the two operations executes first, the result is the same: both objects are successfully inserted. Therefore, the scalable commutativity rule holds that a scalable implementation is possible, and there is, in fact, a wealth of scalable implementations of search structures, ranging from hash tables to radix trees to dense arrays.

Approach

LWN.net

News from the source

Content
Weekly Edition
Archives
Search
Kernel
Kernel page

Overview of the scalable co

We all learned about the comm
the same value as Y + X, for a
commutativity rule uses a mo
encompasses the order of exe
arguments to a single operato

February 17, 2015

This article was contributed by Paul McKenney

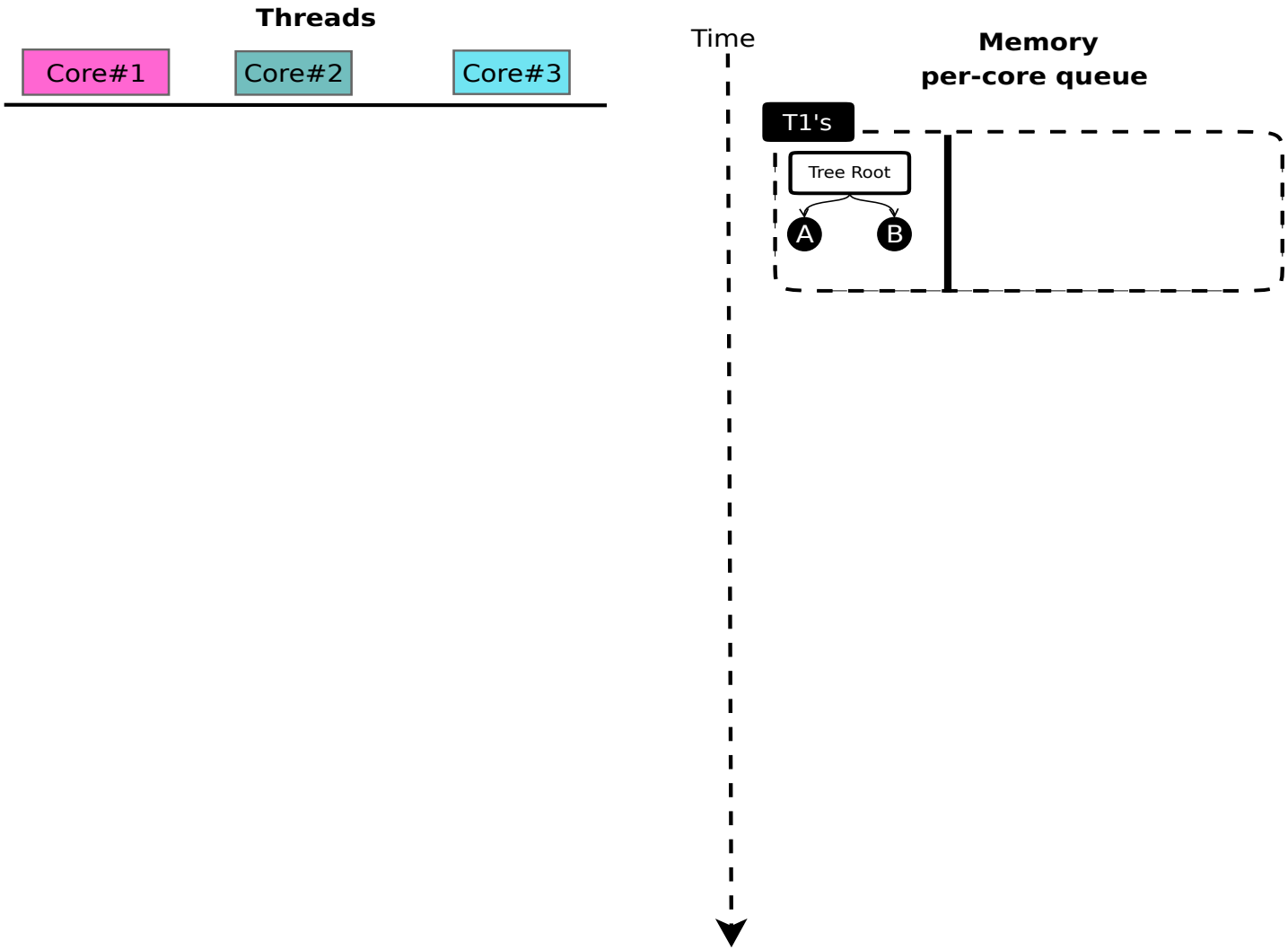
atomically adds the value X to variable A at about the same time that another CPU atomically adds the value Y to this same variable. Because addition is commutative in this more concurrent sense, we know that regardless of the order of execution, the overall effect will be to add X + Y to A.

For example, suppose that the two operations are inserting two objects, each with its own key, into a search data structure not already containing elements with those keys. Suppose further that this structure does not allow duplicate keys. No matter which of the two operations executes first, the result is the same: both objects are successfully inserted. Therefore, the scalable commutativity rule holds

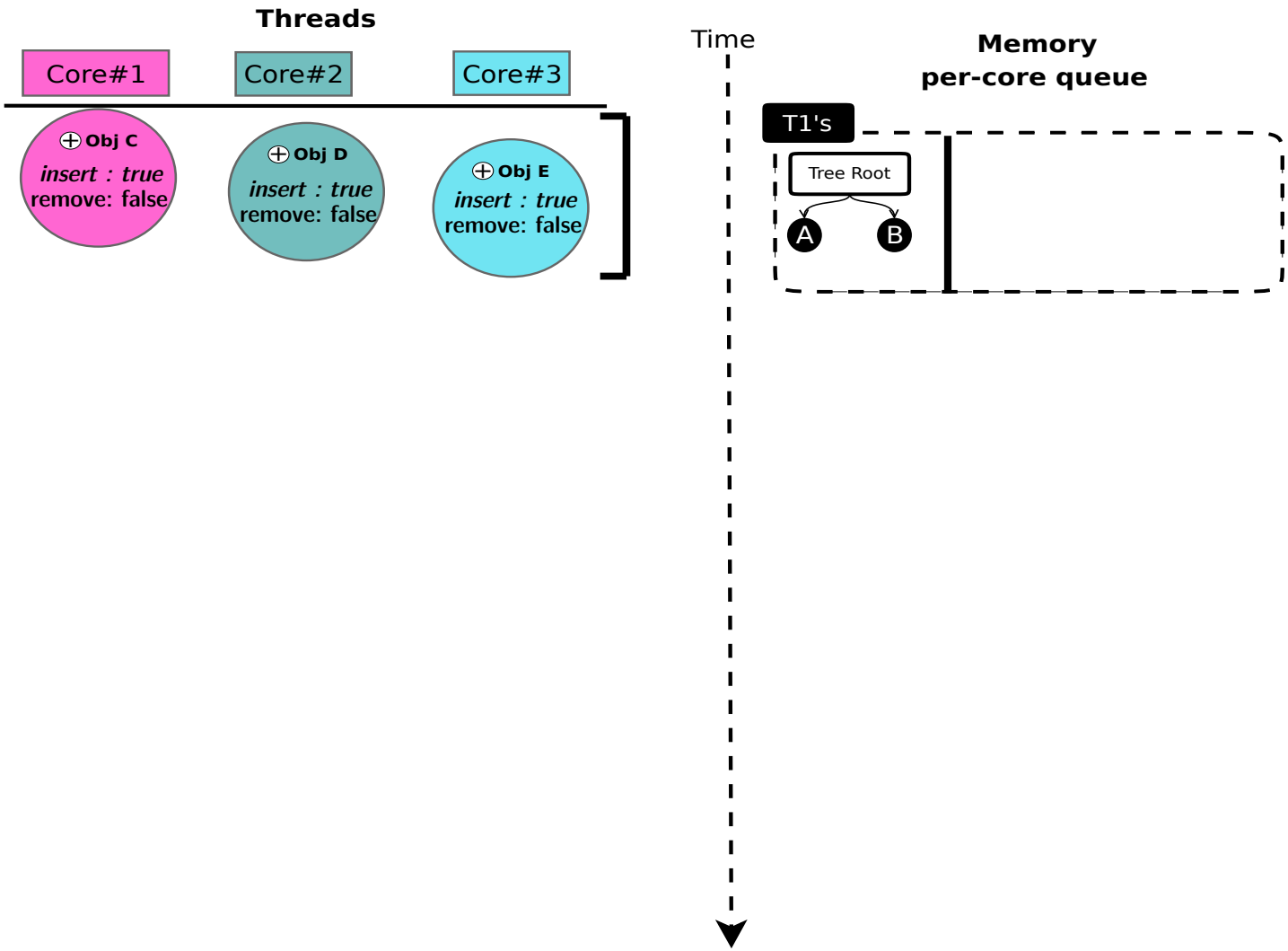
successfully inserted. Therefore, the scalable commutativity rule holds that a scalable implementation is possible, and there is, in fact, a wealth of scalable implementations of search structures, ranging from hash tables to radix trees to dense arrays.

Example

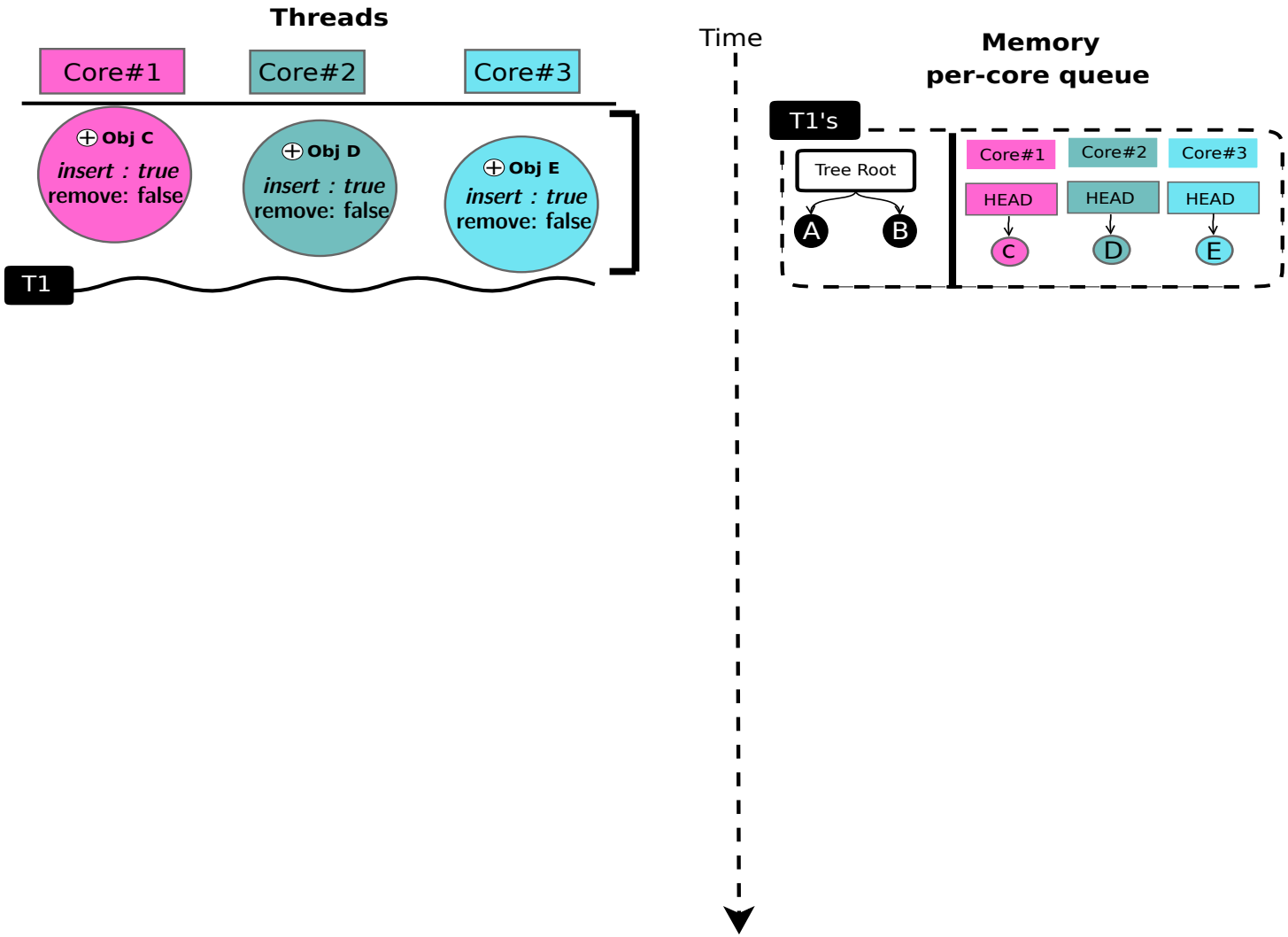
Example


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

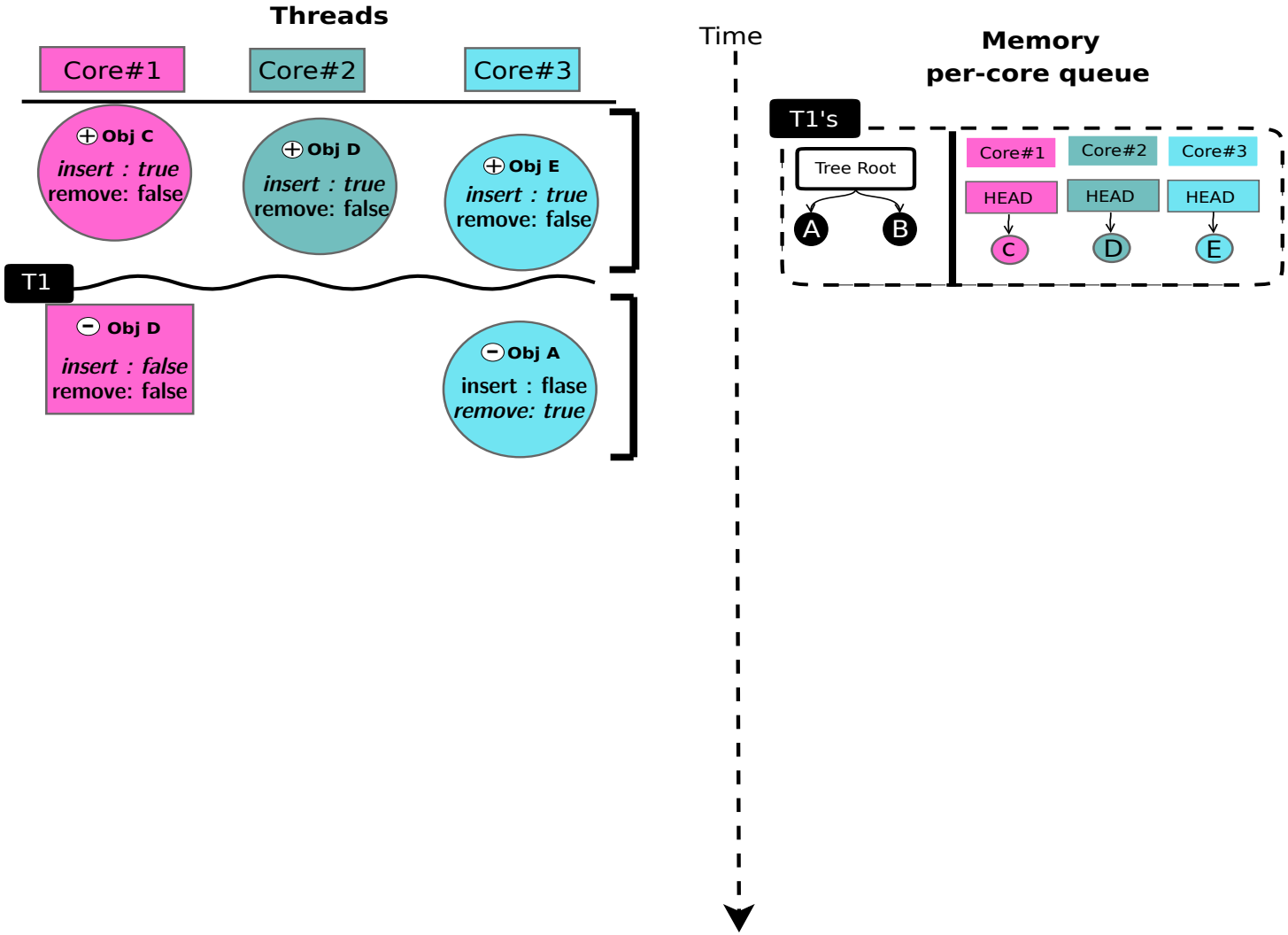
Example


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
Time
Memory
per-core queue
T1's
Tree Root
A
B

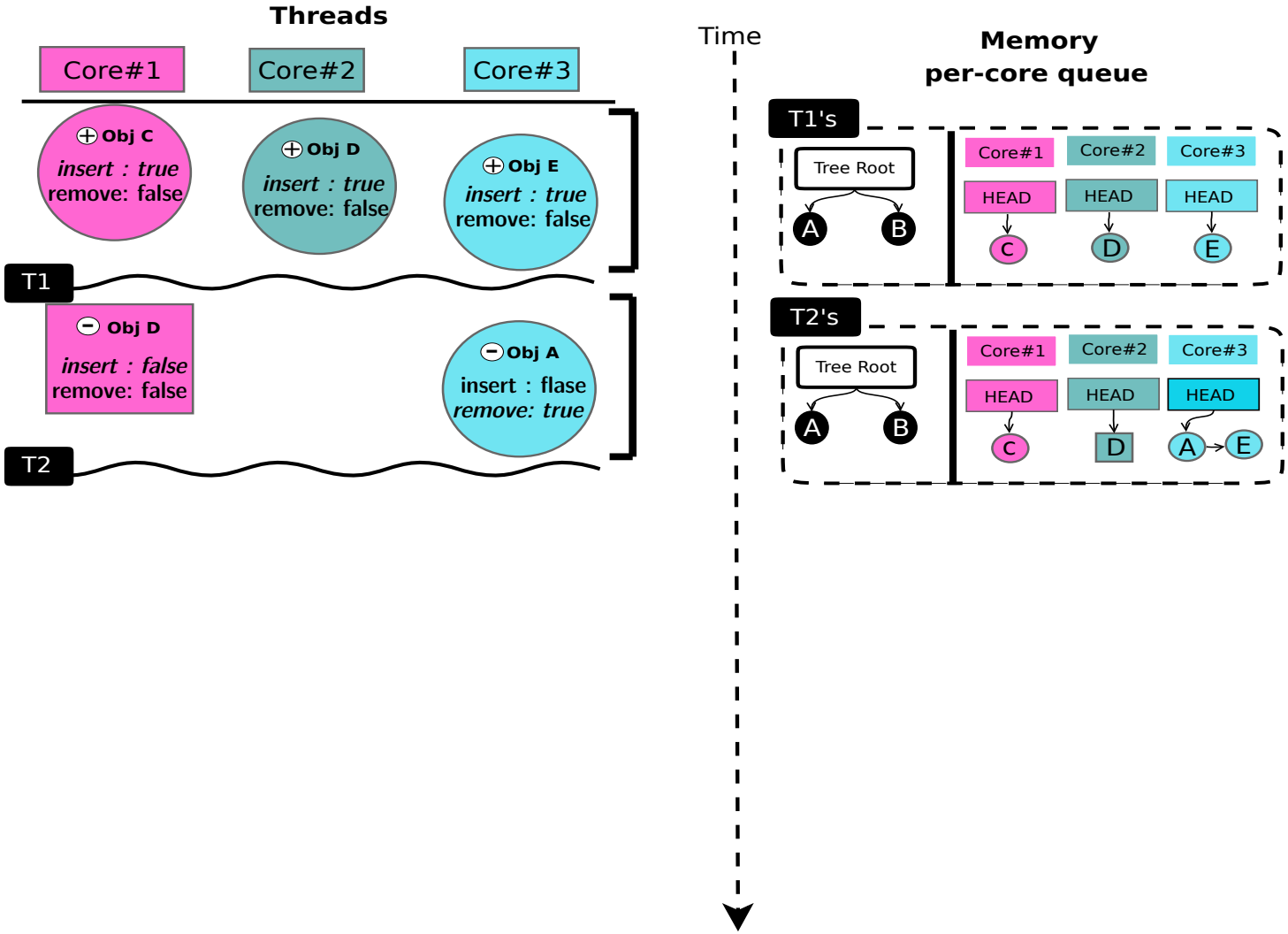
Example


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
T1
Time
Memory
per-core queue
T1's
Tree Root
A
B
Core#1
Core#2
Core#3
HEAD
HEAD
HEAD
C
D
E

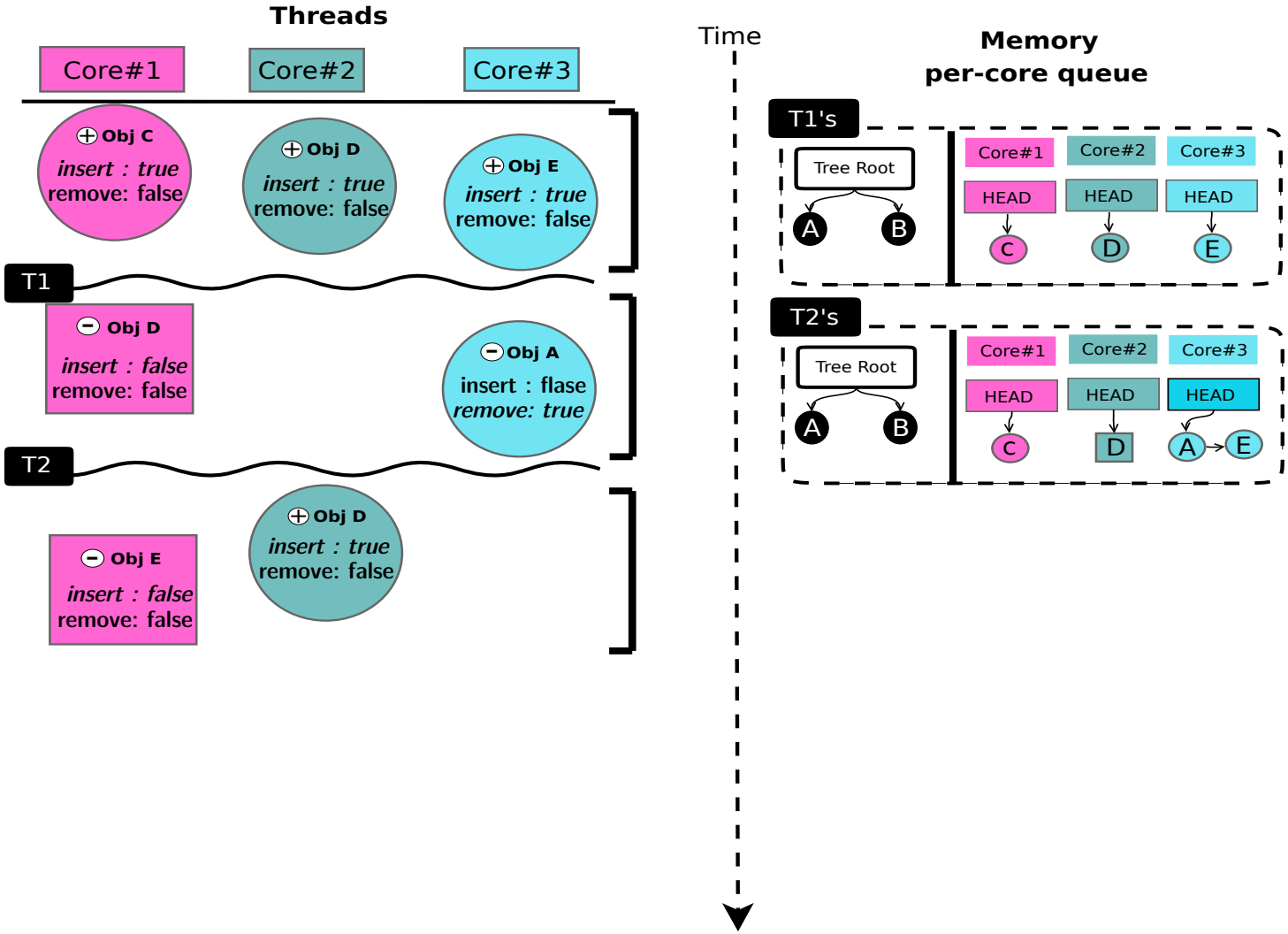
Example


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
T1
⊖ Obj D
insert : false
remove: false
⊖ Obj A
insert : flase
remove: true
Time
Memory
per-core queue
T1's
Tree Root
A
B
Core#1
Core#2
Core#3
HEAD
HEAD
HEAD
C
D
E

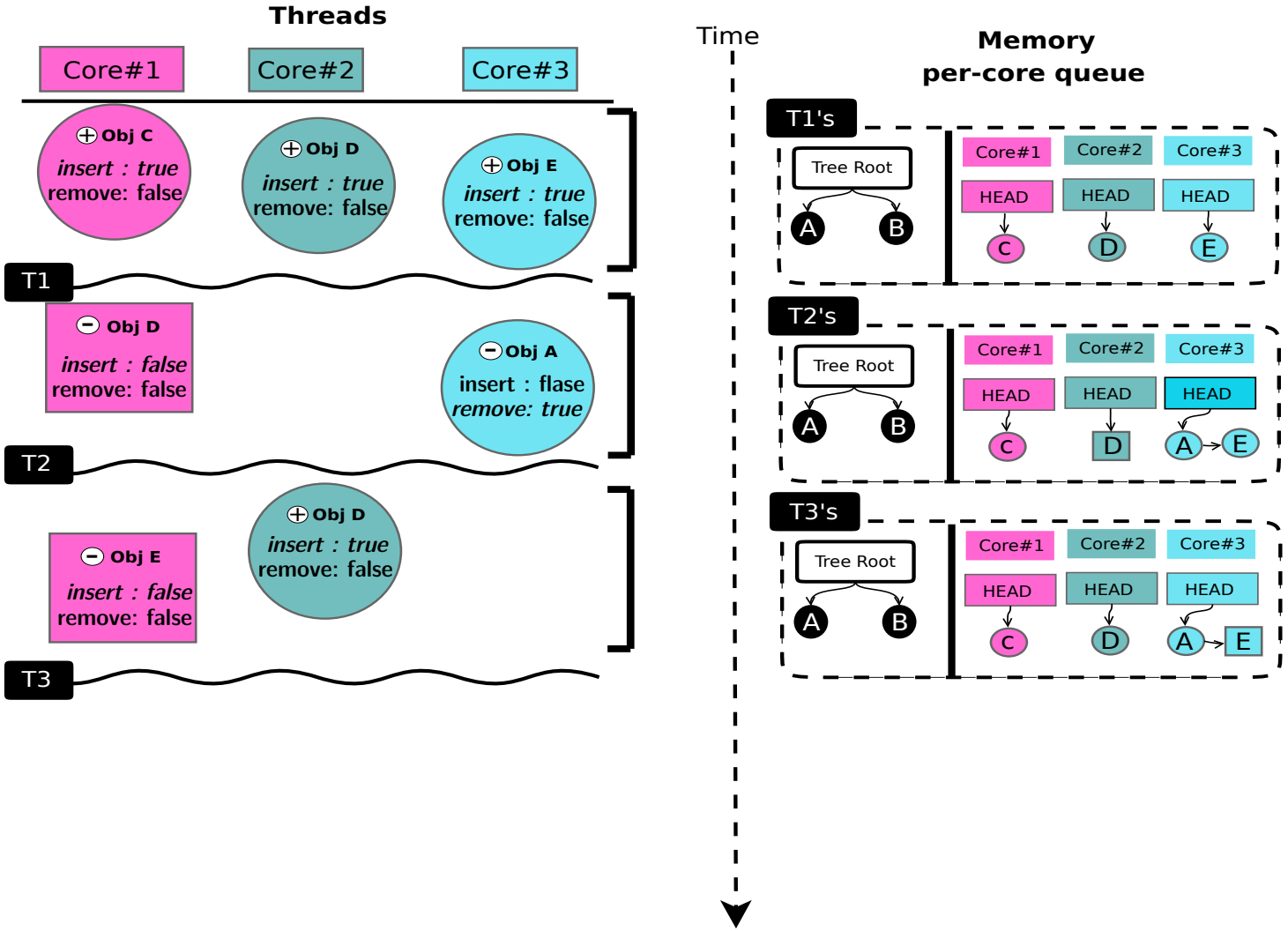
Example


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
T1
Obj D
insert : false
remove: false
Obj A
insert : flase
remove: true
T2
Time
Memory
per-core queue
T1's
Tree Root
A
B
Core#1
Core#2
Core#3
HEAD
HEAD
HEAD
C
D
E
T2's
Tree Root
A
B
Core#1
Core#2
Core#3
HEAD
HEAD
HEAD
C
D
A
E

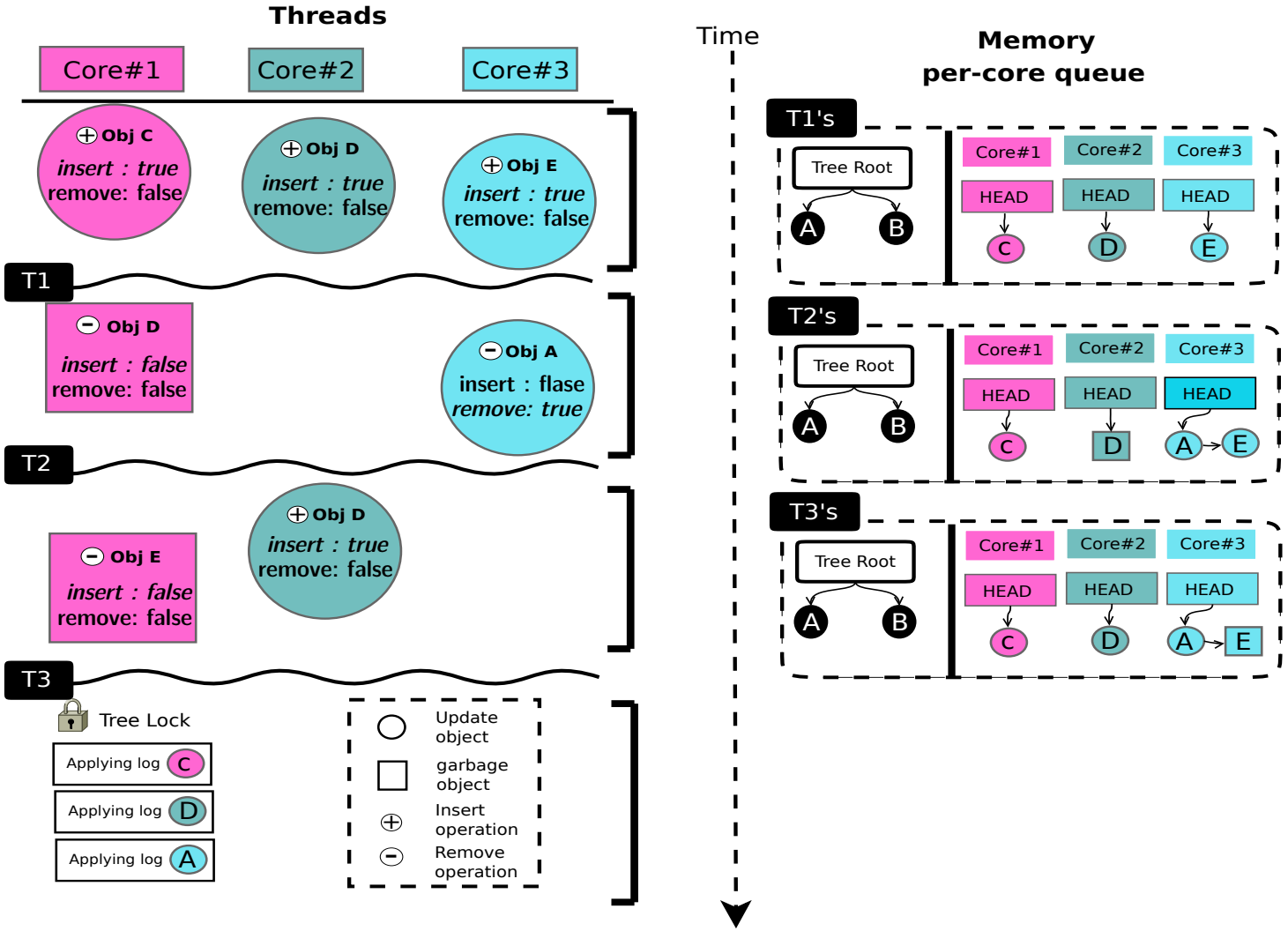
Example


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
T1
Obj D
insert : false
remove: false
Obj A
insert : flase
remove: true
T2
Obj D
insert : true
remove: false
Obj E
insert : false
remove: false
Time
Memory
per-core queue
T1's
Tree Root
A
B
Core#1
Core#2
Core#3
HEAD
HEAD
HEAD
C
D
E
T2's
Tree Root
A
B
Core#1
Core#2
Core#3
HEAD
HEAD
HEAD
C
D
A
E

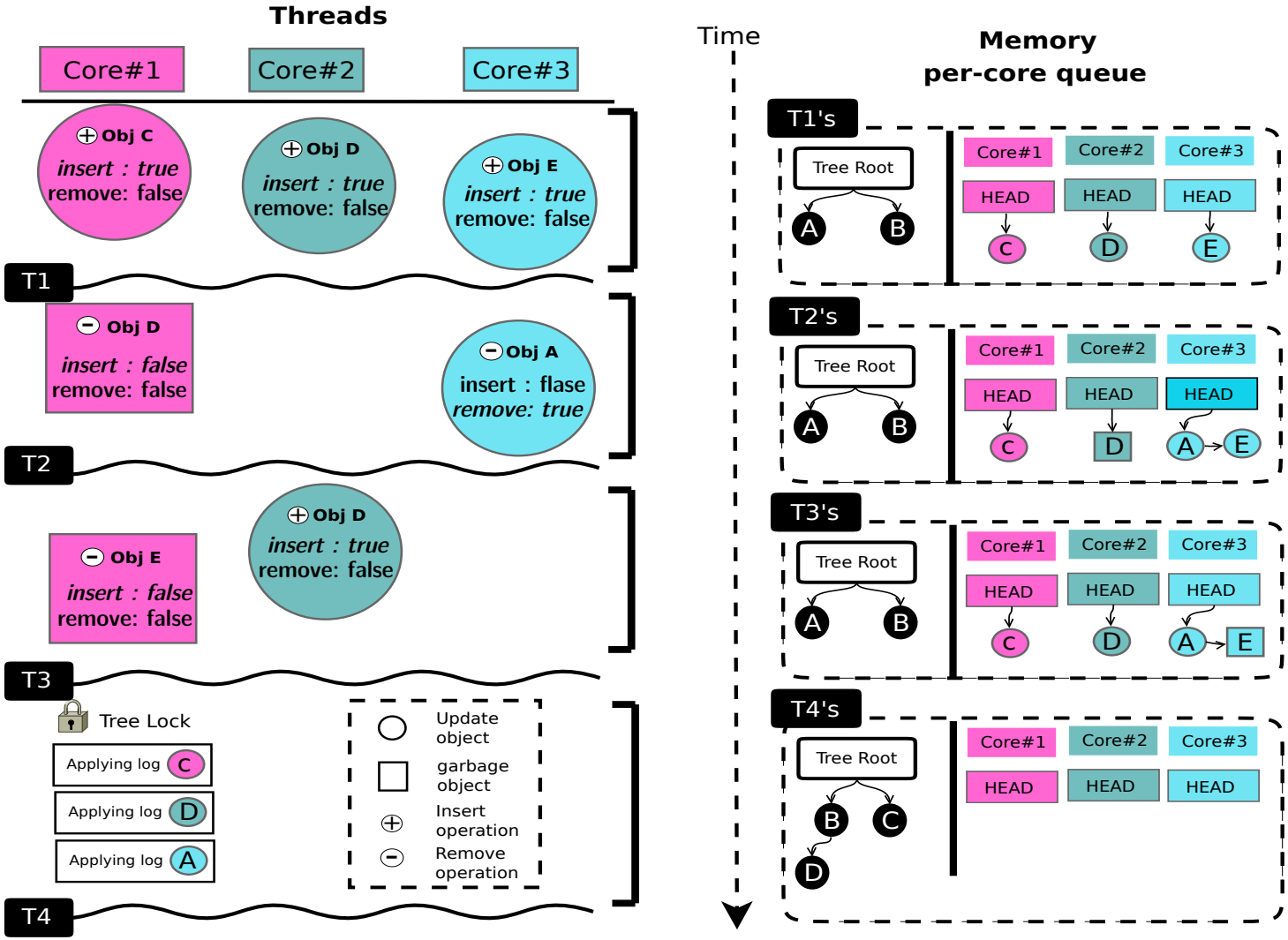
Example


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
T1
Obj D
insert : false
remove: false
Obj A
insert : flase
remove: true
T2
Obj D
insert : true
remove: false
Obj E
insert : false
remove: false
T3
Time
Memory
per-core queue
T1's
Tree Root
A
B
Core#1
Core#2
Core#3
HEAD
HEAD
HEAD
C
D
E
T2's
Tree Root
A
B
Core#1
Core#2
Core#3
HEAD
HEAD
HEAD
C
D
A
E
T3's
Tree Root
A
B
Core#1
Core#2
Core#3
HEAD
HEAD
HEAD
C
D
A
E

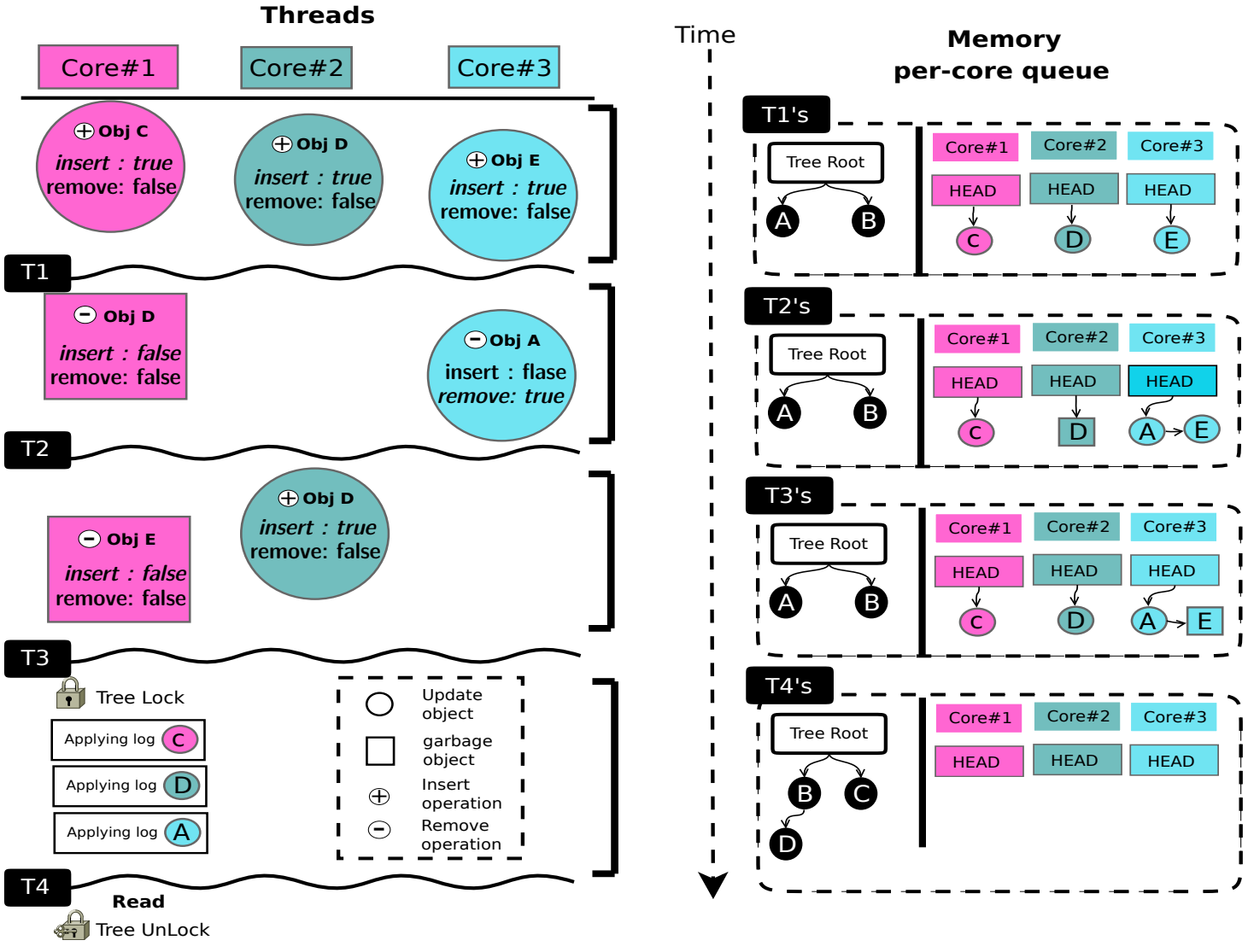
Example


Threads
Core#1
Core#2
Core#3
Time
Memory
per-core queue
Obj C
insert : true
remove: false
Obj D
insert : true
remove: false
Obj E
insert : true
remove: false
T1
Obj D
insert : false
remove: false
Obj A
insert : flase
remove: true
T2
Obj D
insert : true
remove: false
Obj E
insert : false
remove: false
T3
Tree Lock
Applying log C
Applying log D
Applying log A
Update object
garbage object
Insert operation
Remove operation
T1's
T2's
T3's
Tree Root
A
B
HEAD
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
T1
Obj D
insert : false
remove: false
Obj A
insert : flase
remove: true
T2
Obj D
insert : true
remove: false
Obj E
insert : false
remove: false
T3
Tree Lock
Applying log C
Applying log D
Applying log A
Update object
garbage object
Insert operation
Remove operation
T4
Time
Memory
per-core queue
T1's
Tree Root
A
B
Core#1
Core#2
Core#3
HEAD
HEAD
HEAD
C
D
E
T2's
Tree Root
A
B
Core#1
Core#2
Core#3
HEAD
HEAD
HEAD
C
D
A
E
T3's
Tree Root
A
B
Core#1
Core#2
Core#3
HEAD
HEAD
HEAD
C
D
A
E
T4's
Tree Root
B
C
D
Core#1
Core#2
Core#3
HEAD
HEAD
HEAD

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
T1
Obj D
insert : false
remove: false
Obj A
insert : flase
remove: true
T2
Obj D
insert : true
remove: false
Obj E
insert : false
remove: false
T3
Tree Lock
Applying log C
Applying log D
Applying log A
Update object
garbage object
Insert operation
Remove operation
T4
Read
Tree UnLock
Time
Memory
per-core queue
T1's
T2's
T3's
T4's
Tree Root
HEAD
A
B
C
D
E

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.

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.

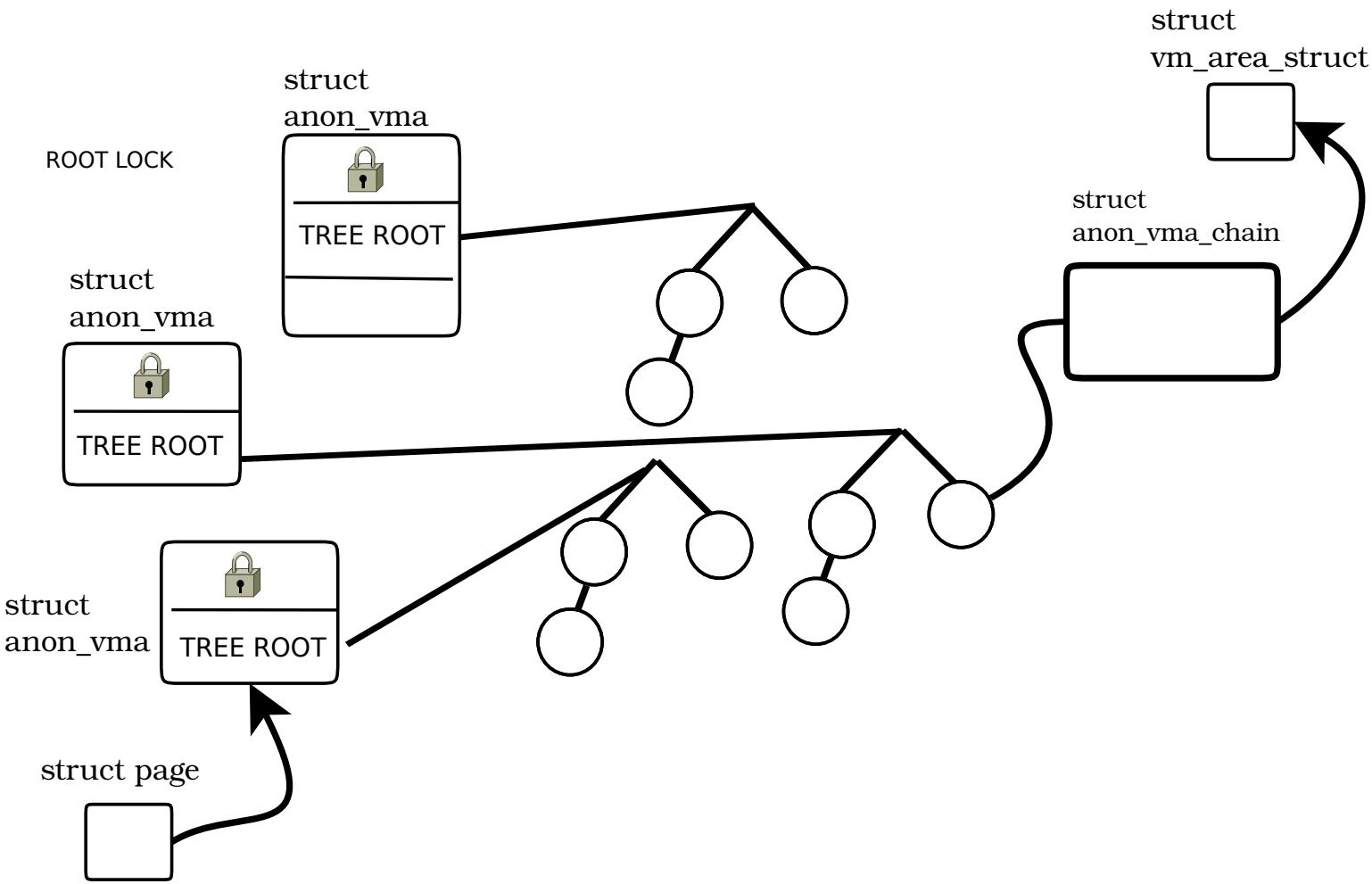
Applying the Linux kernel

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

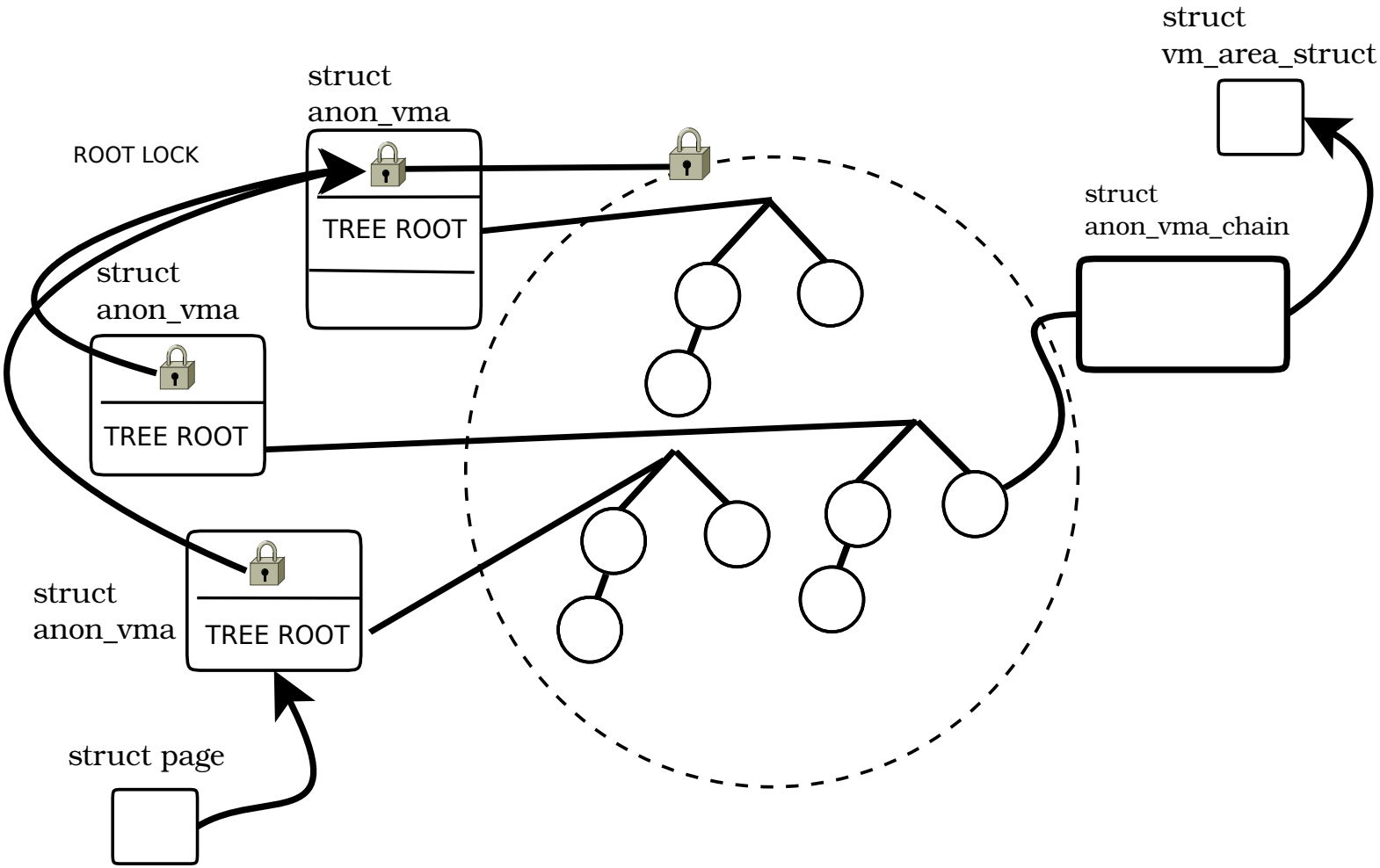
Applying the Linux kernel

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

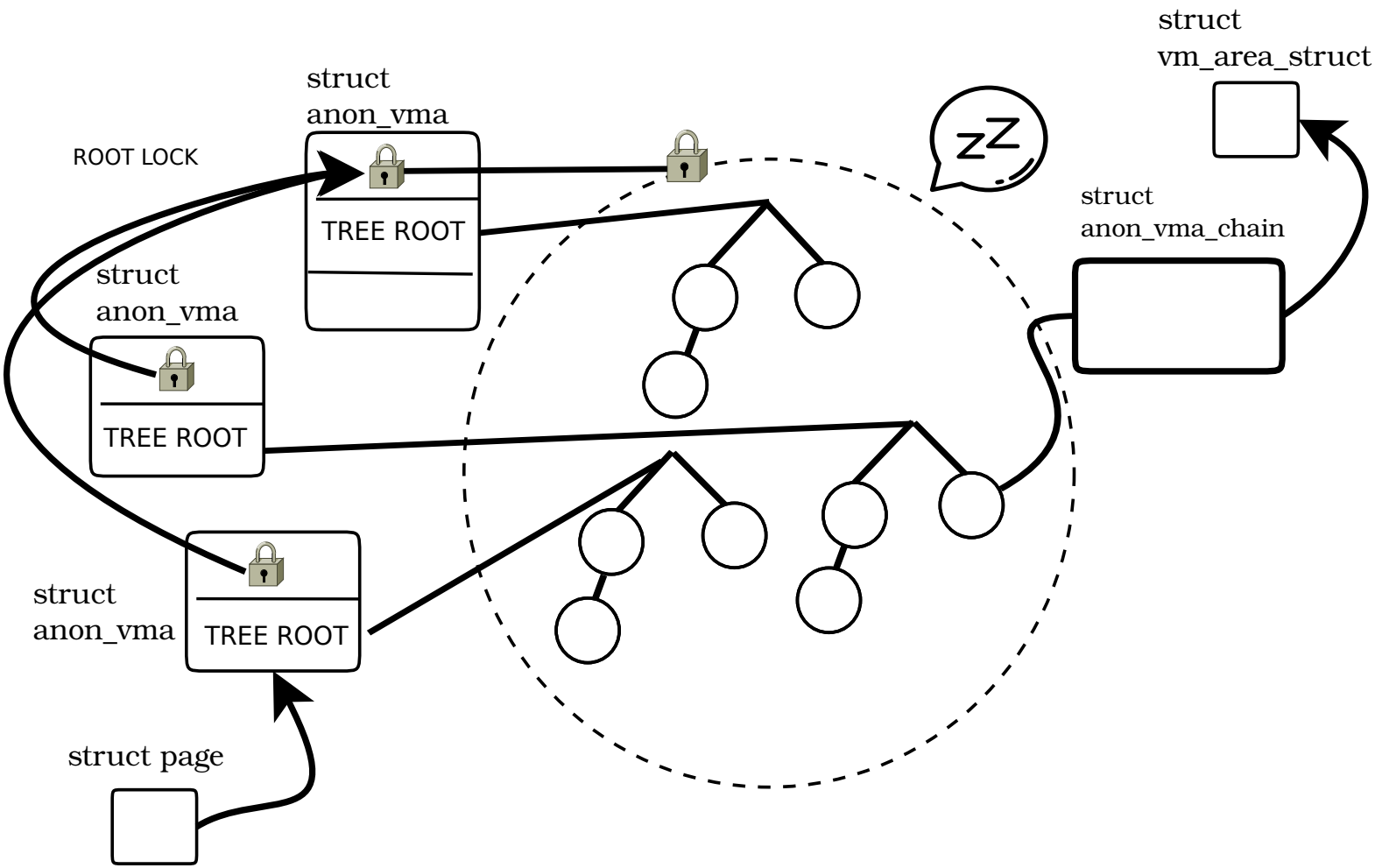
Anonymous reverse mapping



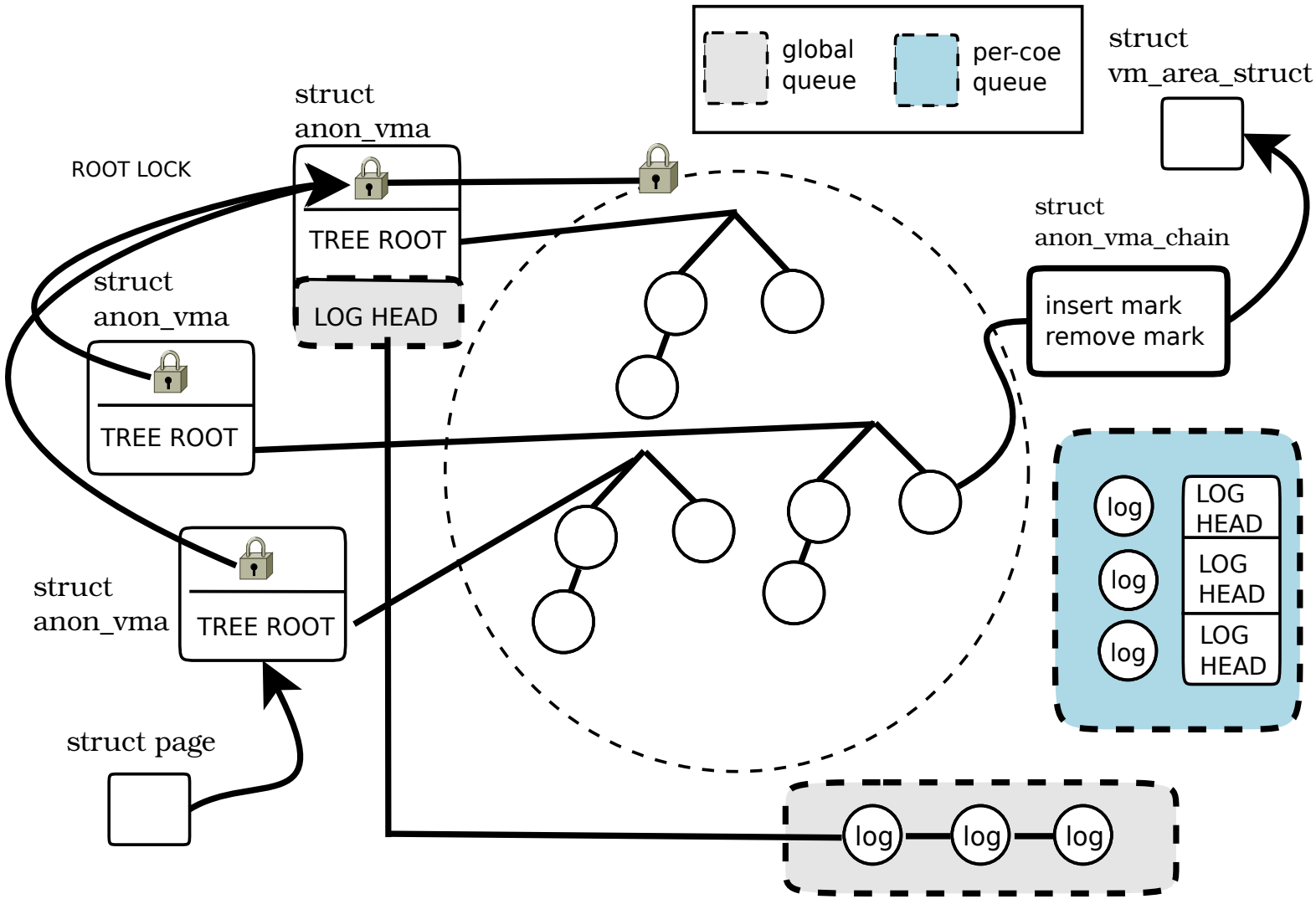
Anonymous reverse mapping



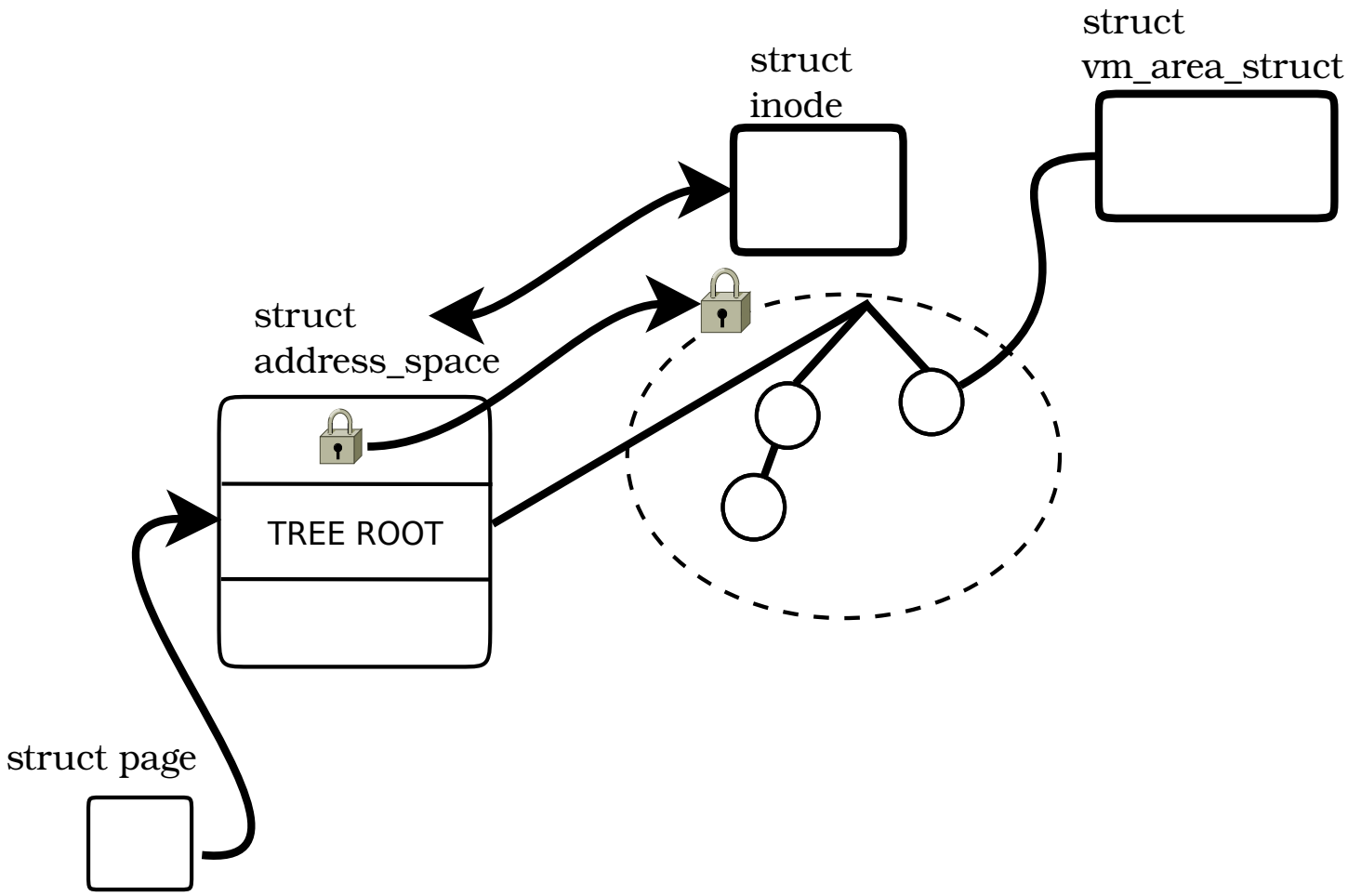
Anonymous reverse mapping



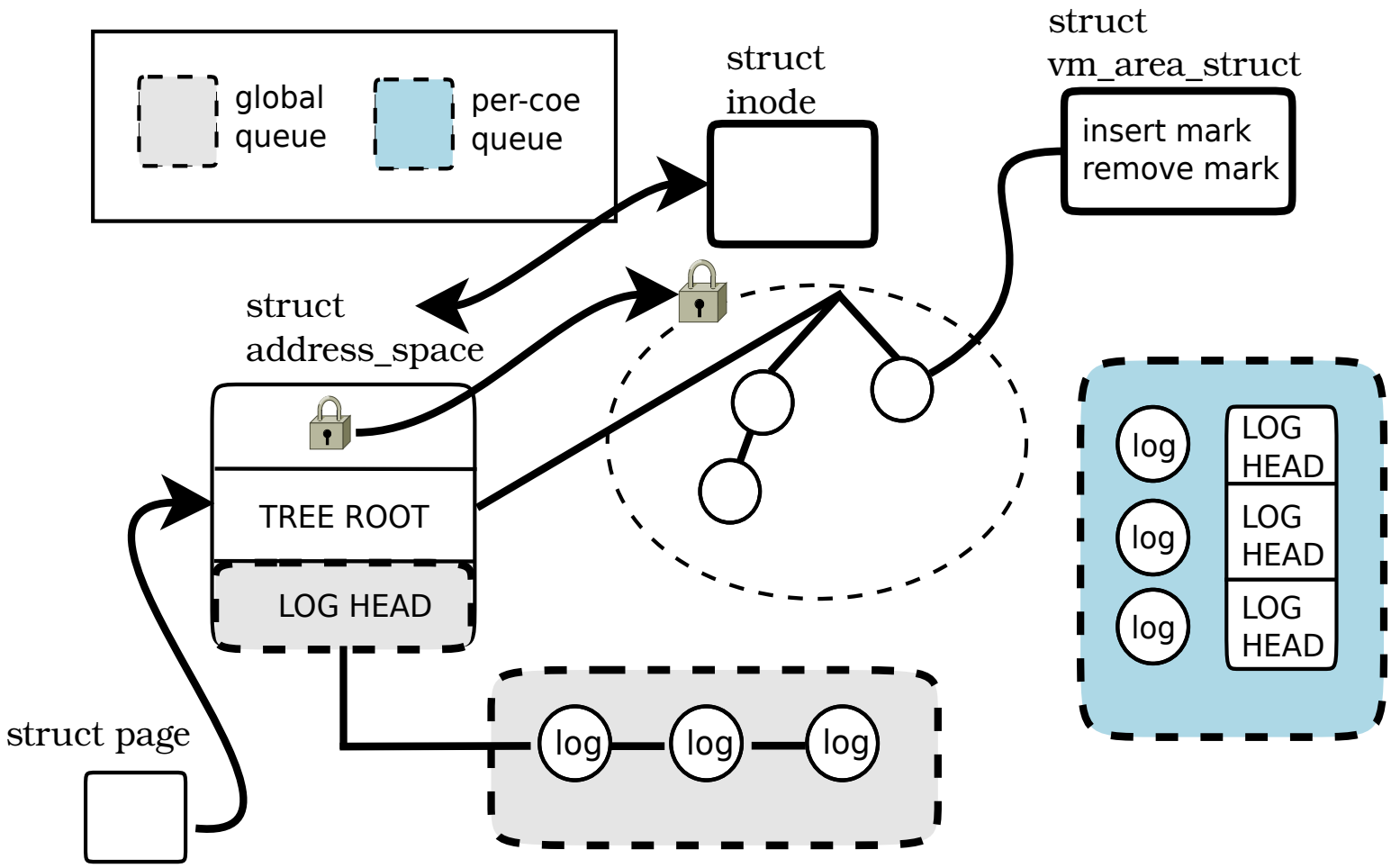
Anonymous reverse mapping



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

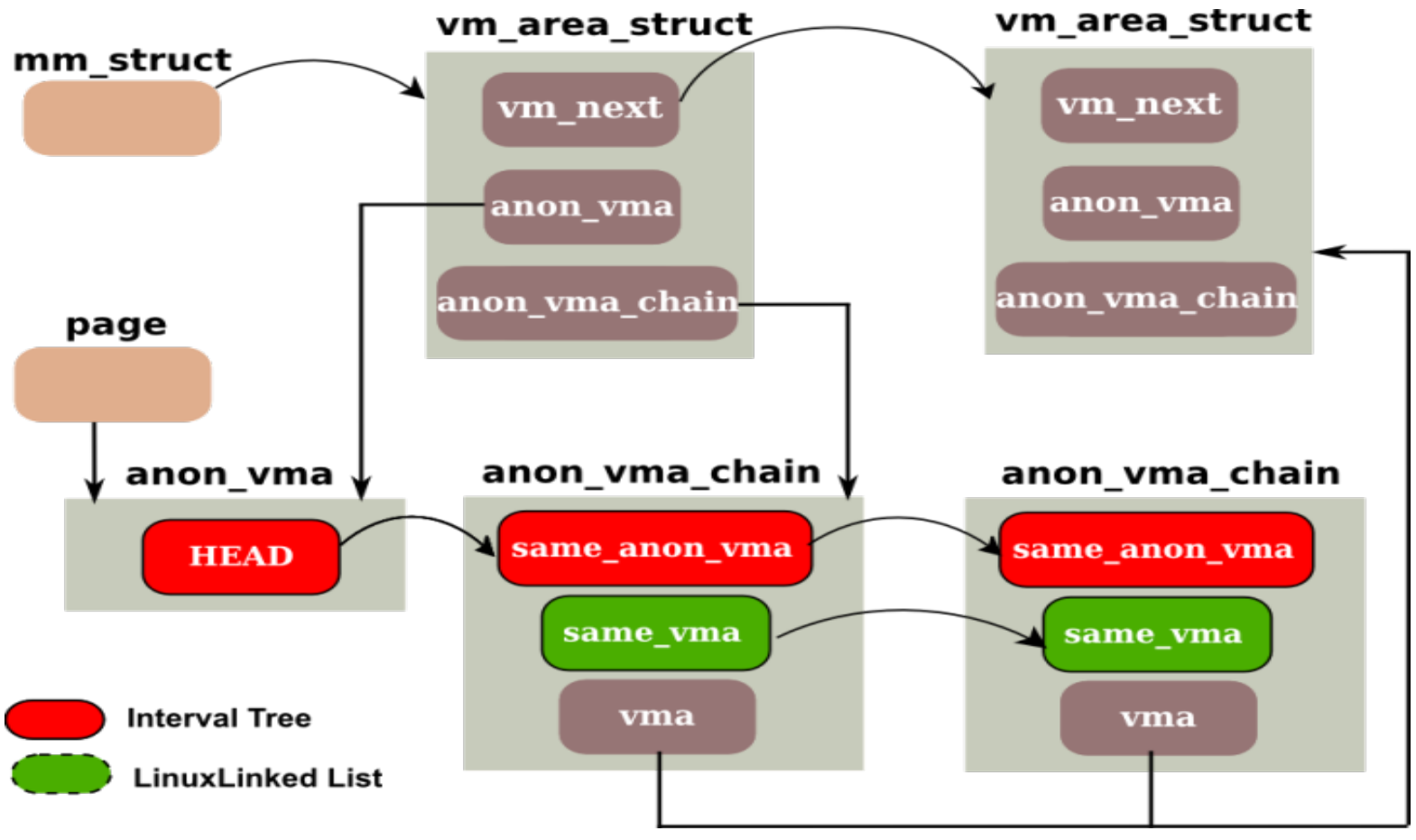
Evaluation

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

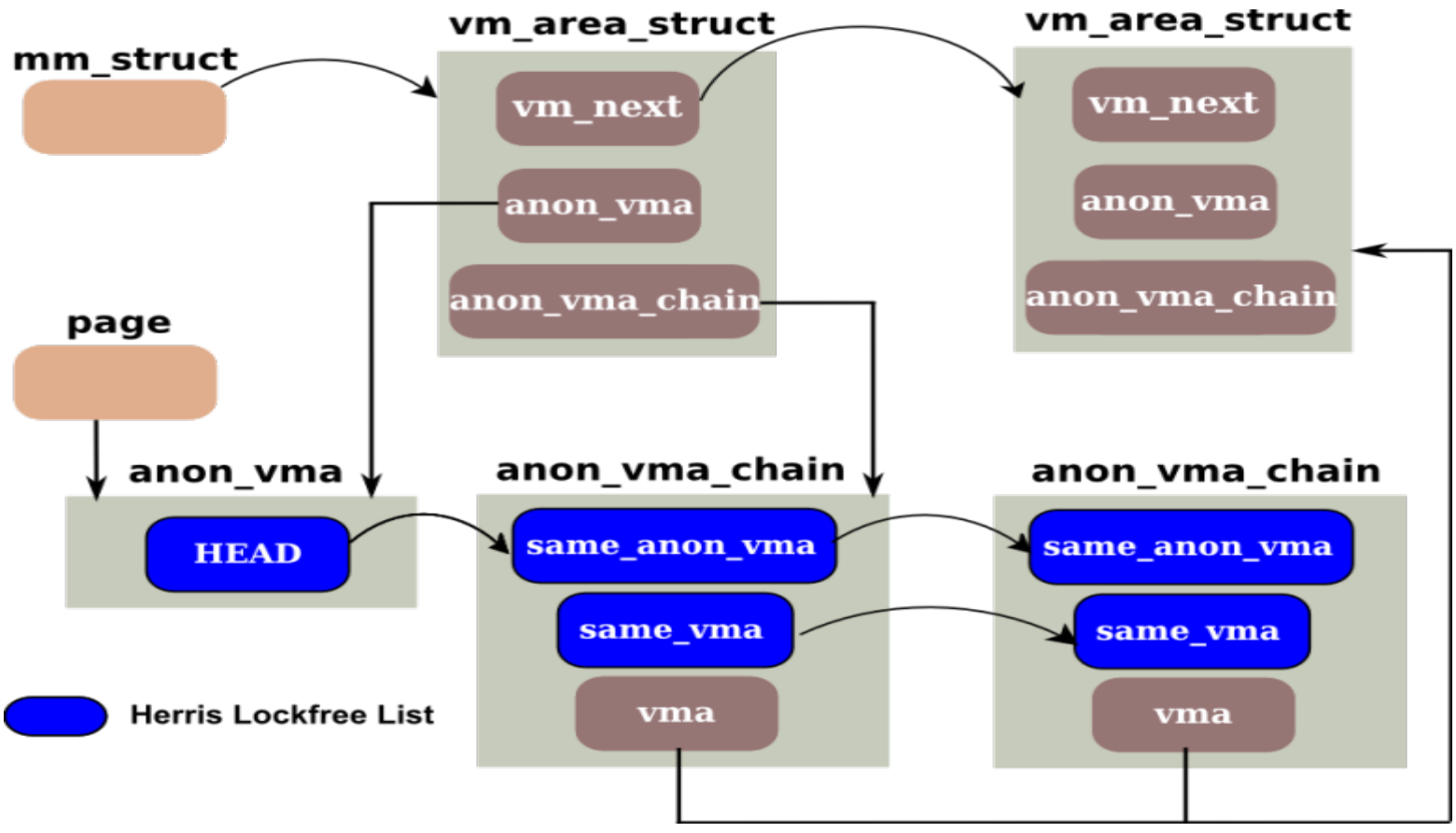
Evaluation

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

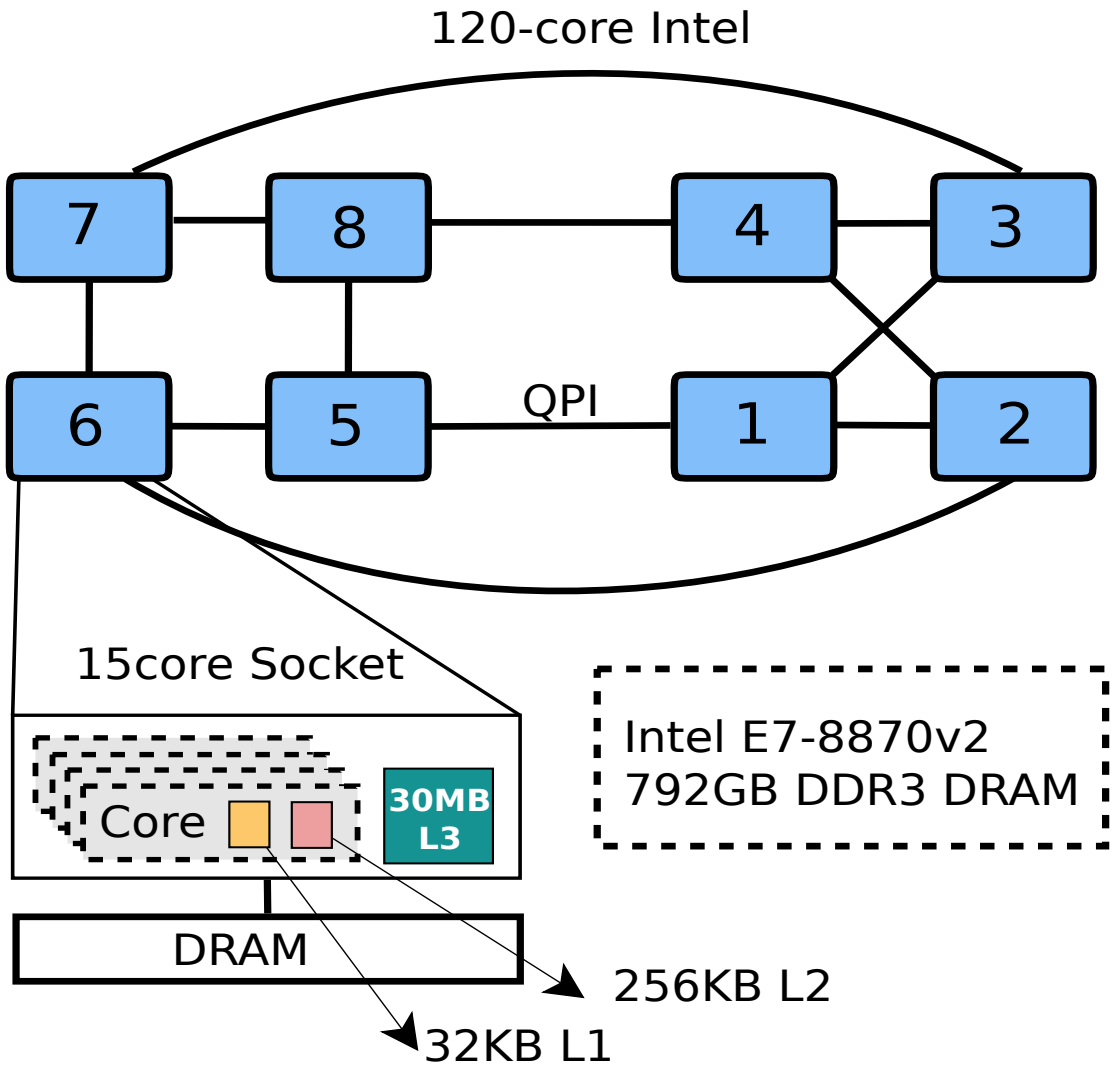
Non-blocking algorithm – Harris linked list



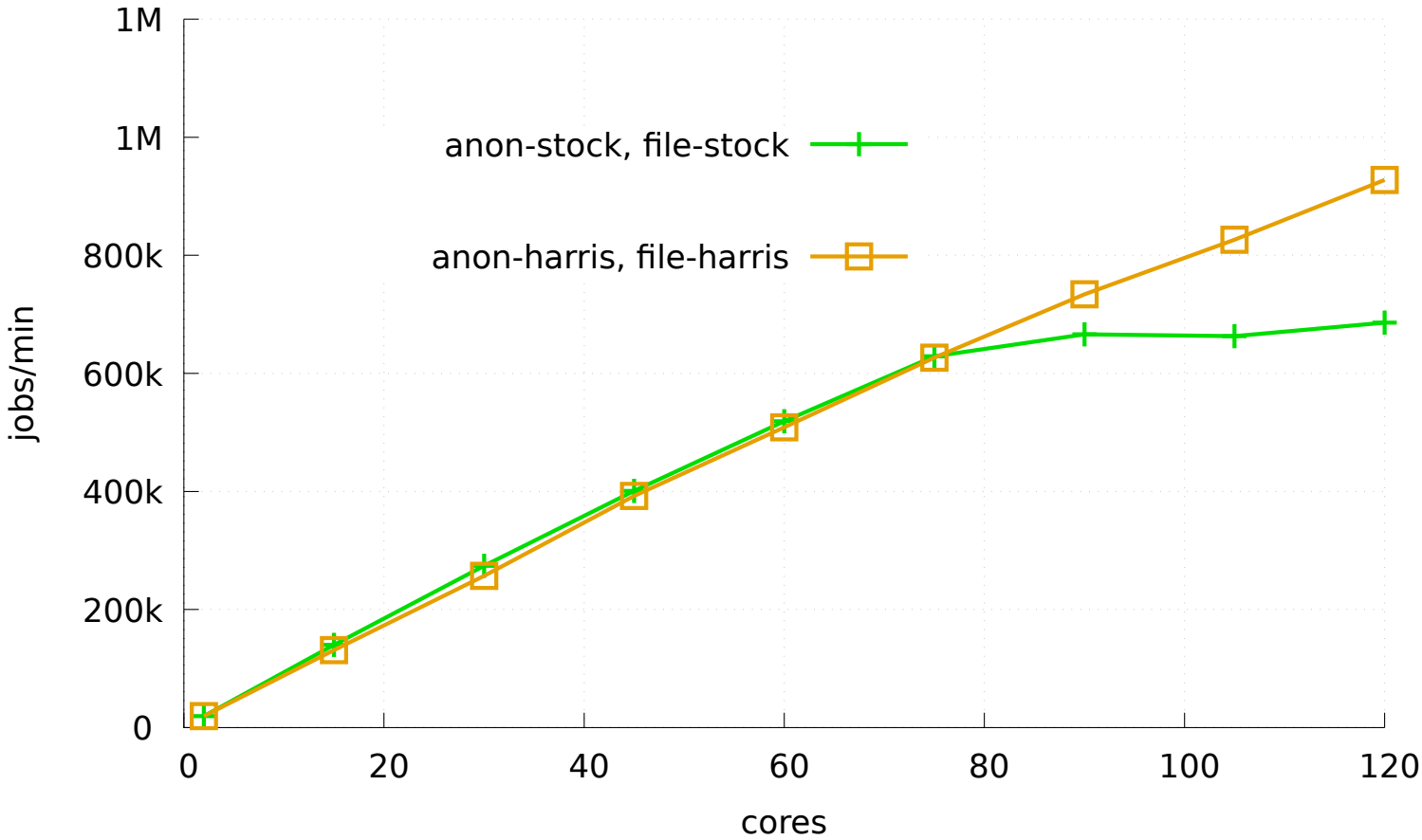
Non-blocking algorithm – Harris linked list



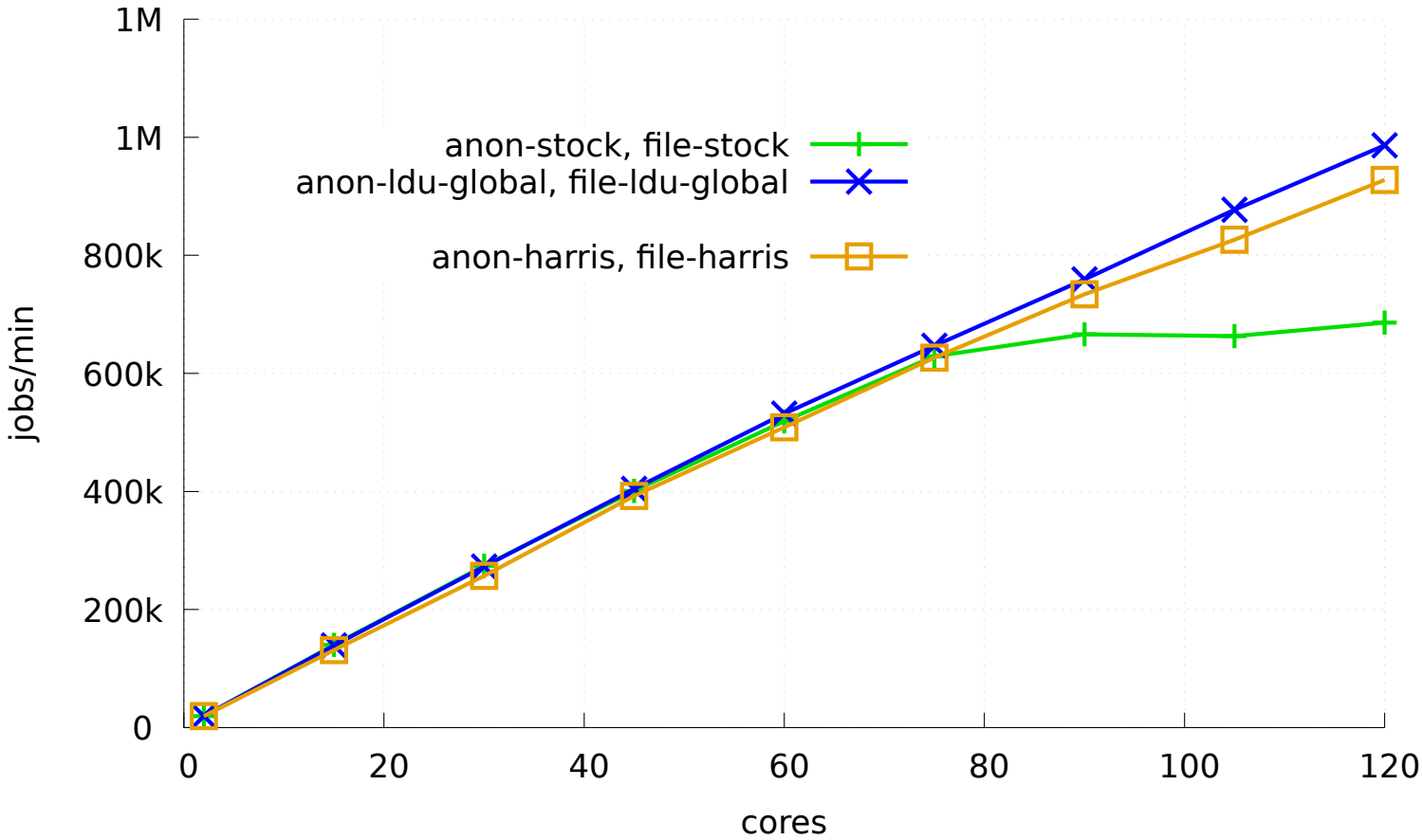
Evaluation : Hardware



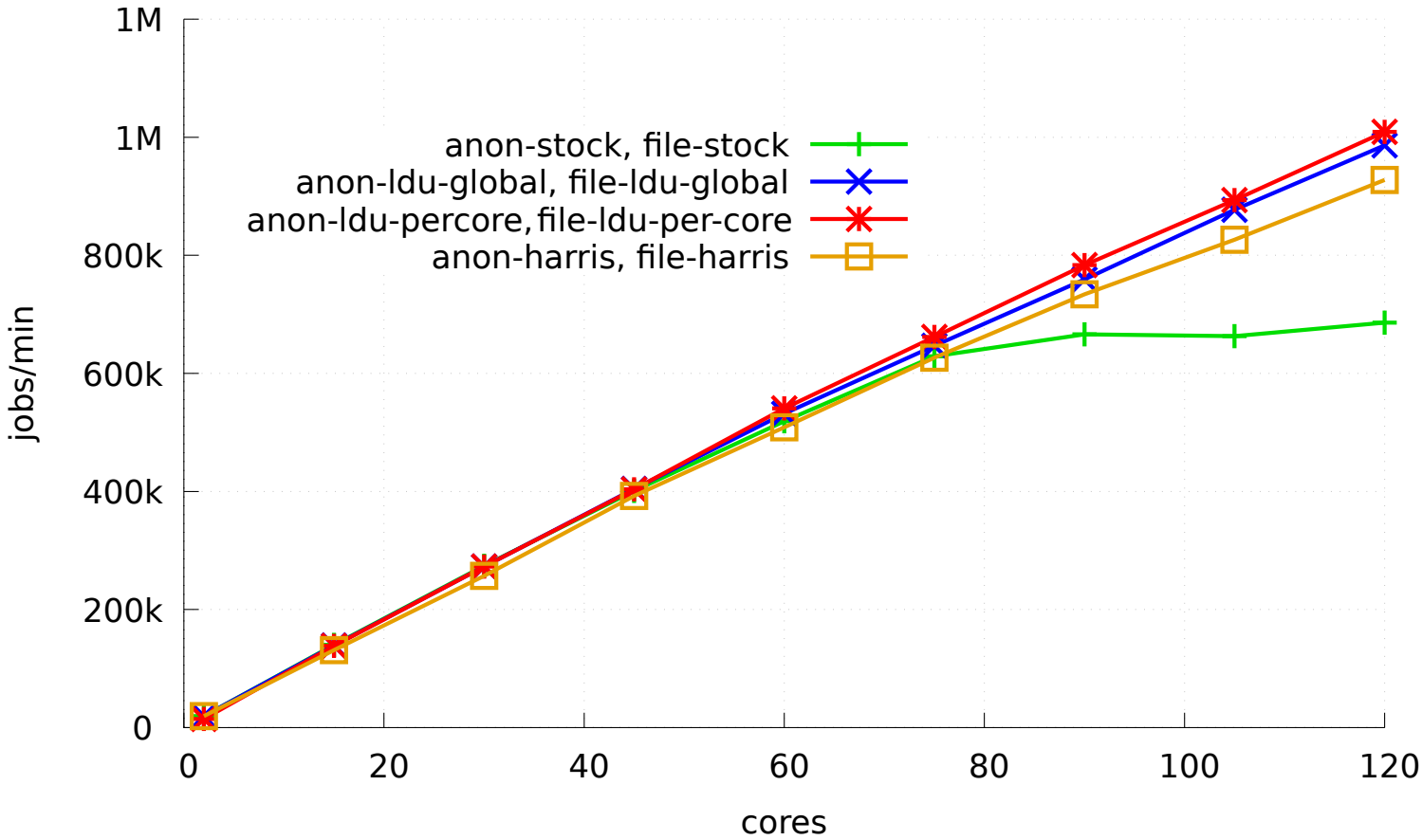
AIM7


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

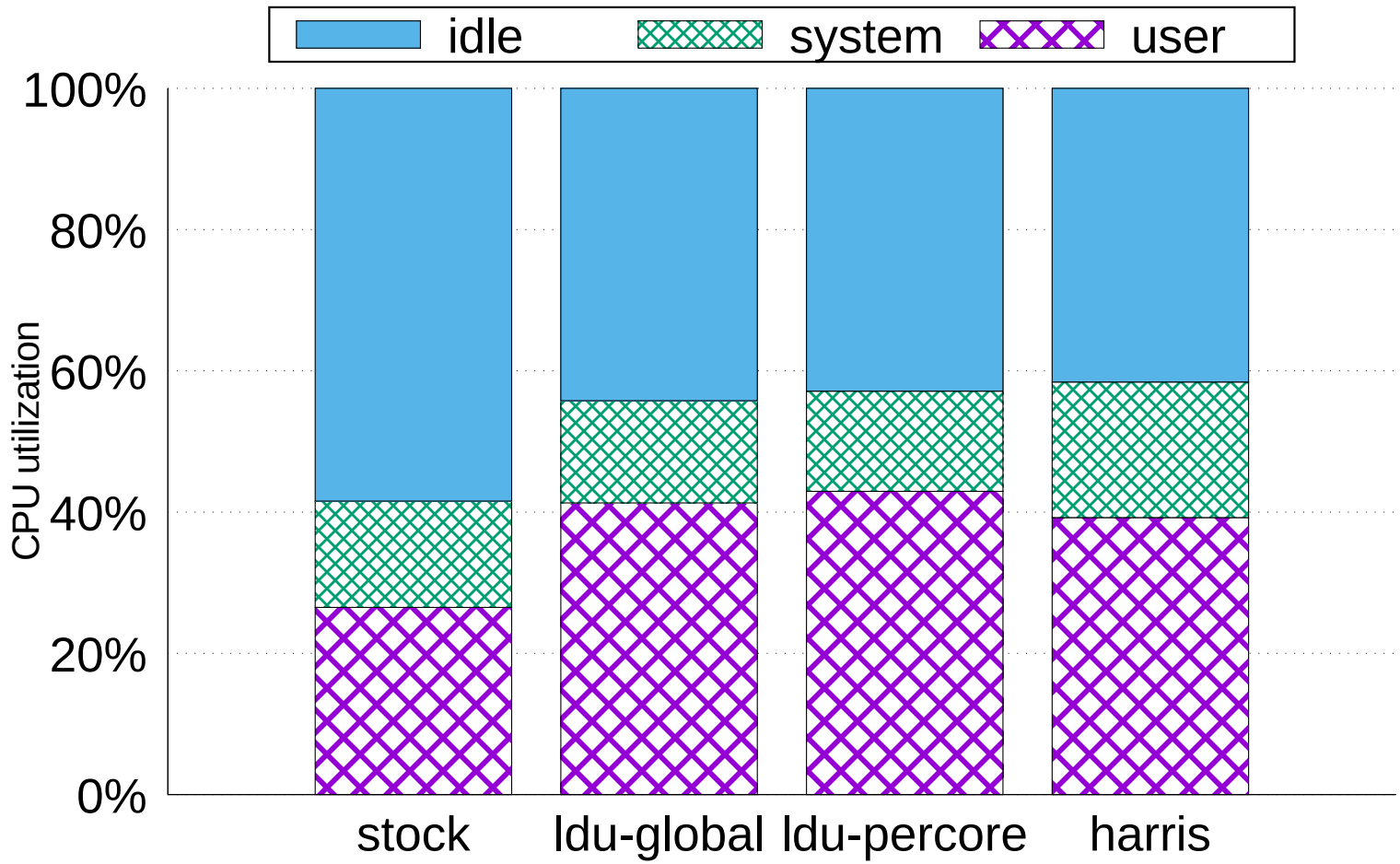
AIM7


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

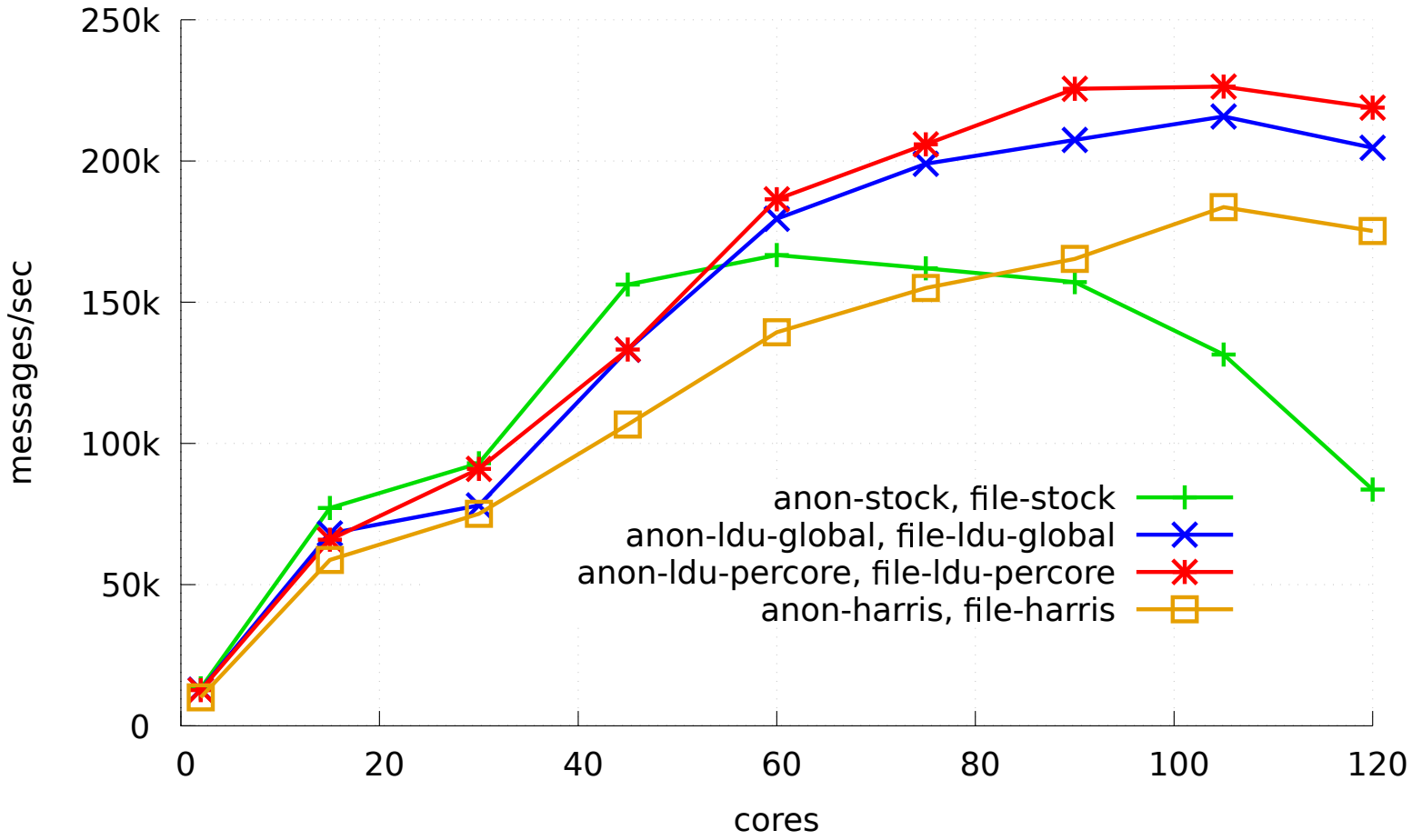
AIM7


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

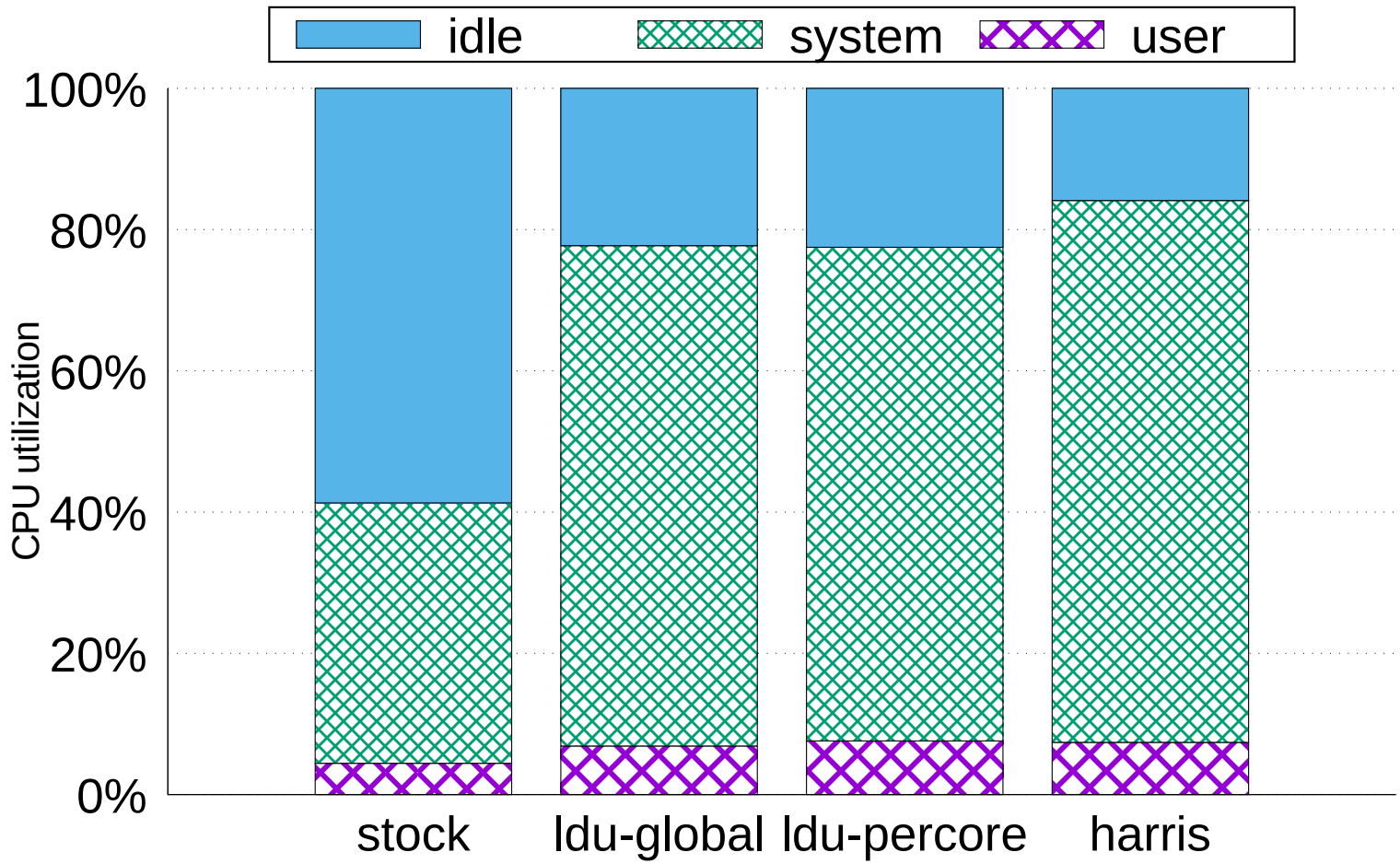
AIM7 – CPU utilization



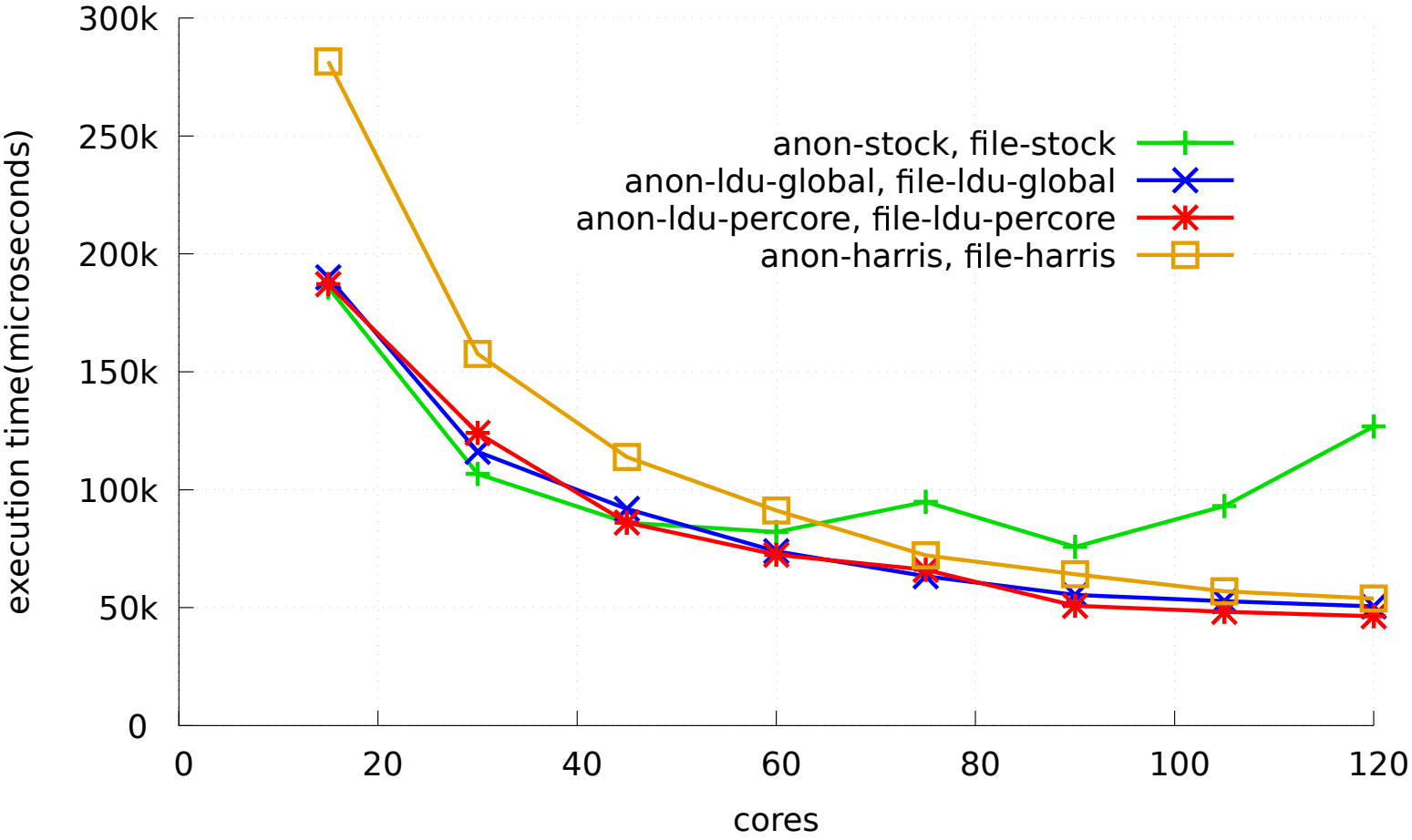
EXIM


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

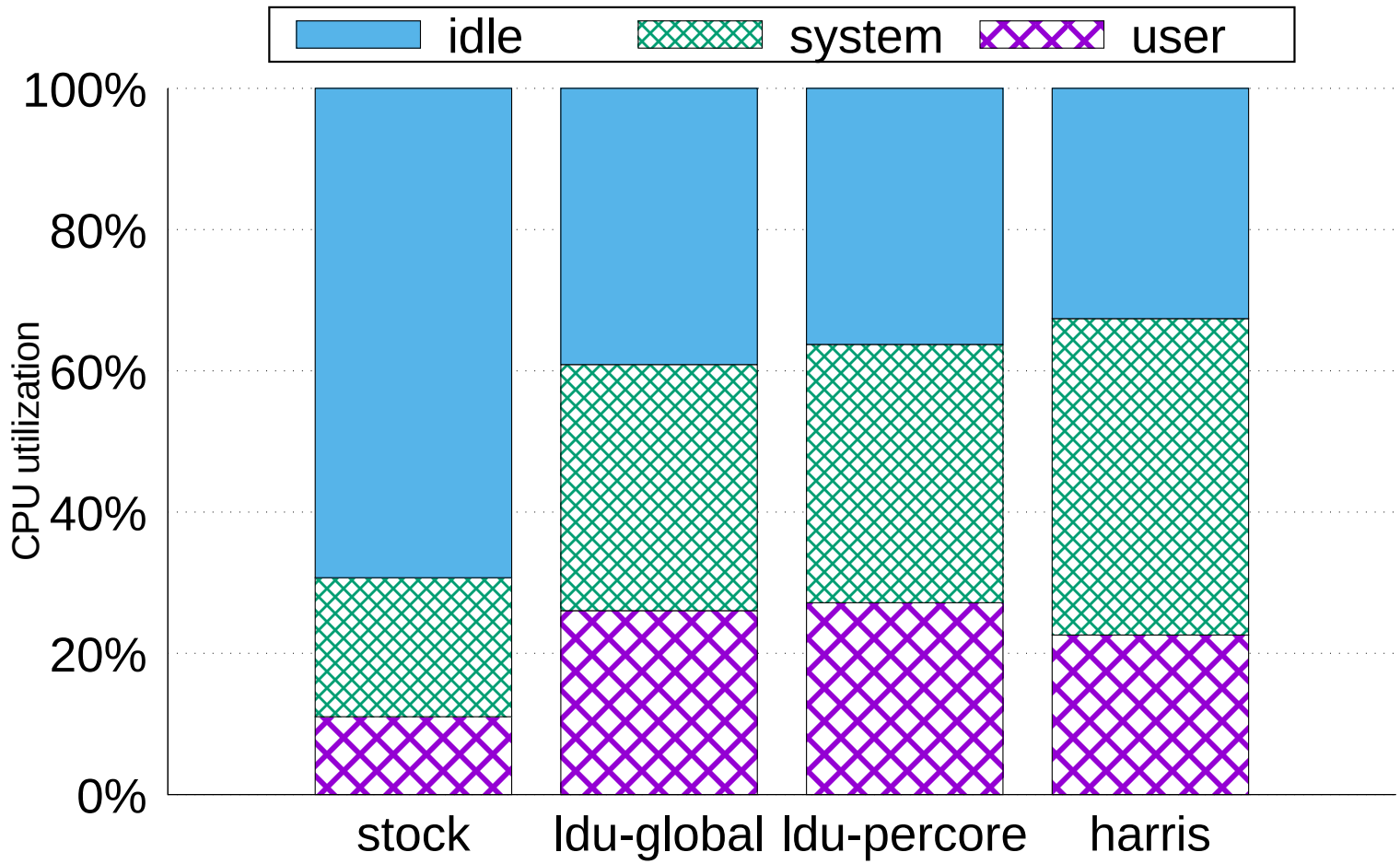
EXIM – CPU utilization



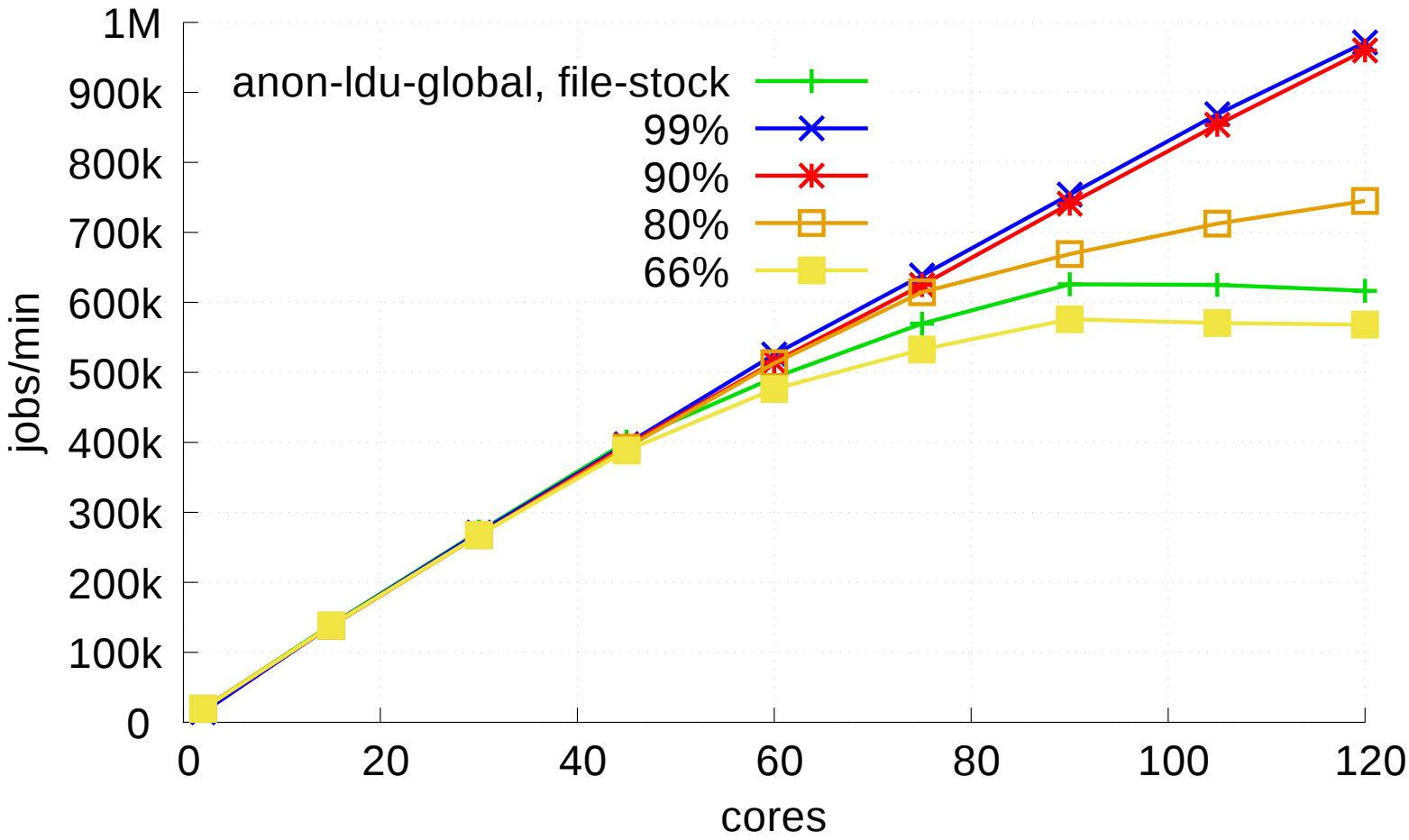
Lmbench


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

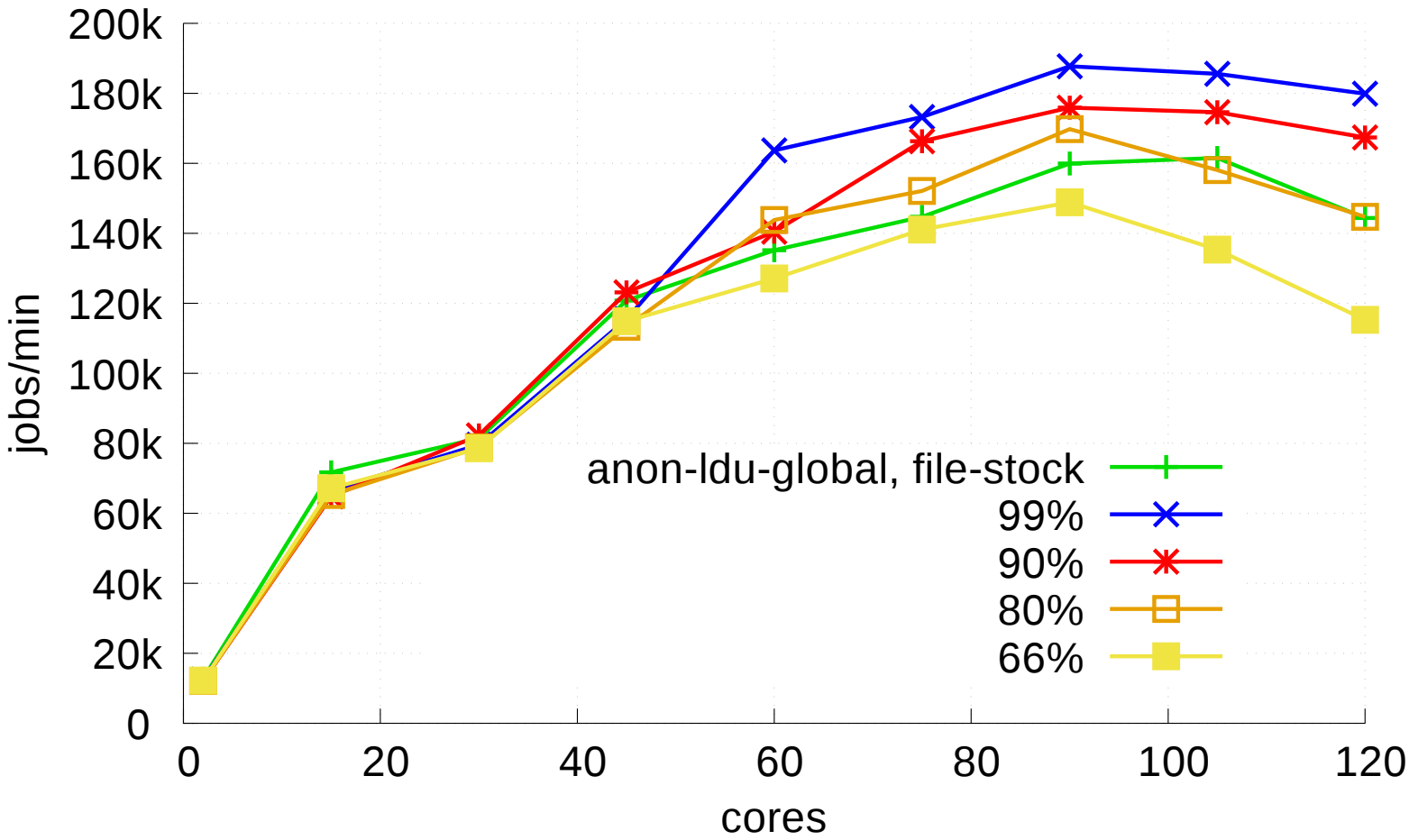
Lmbench – CPU utilization



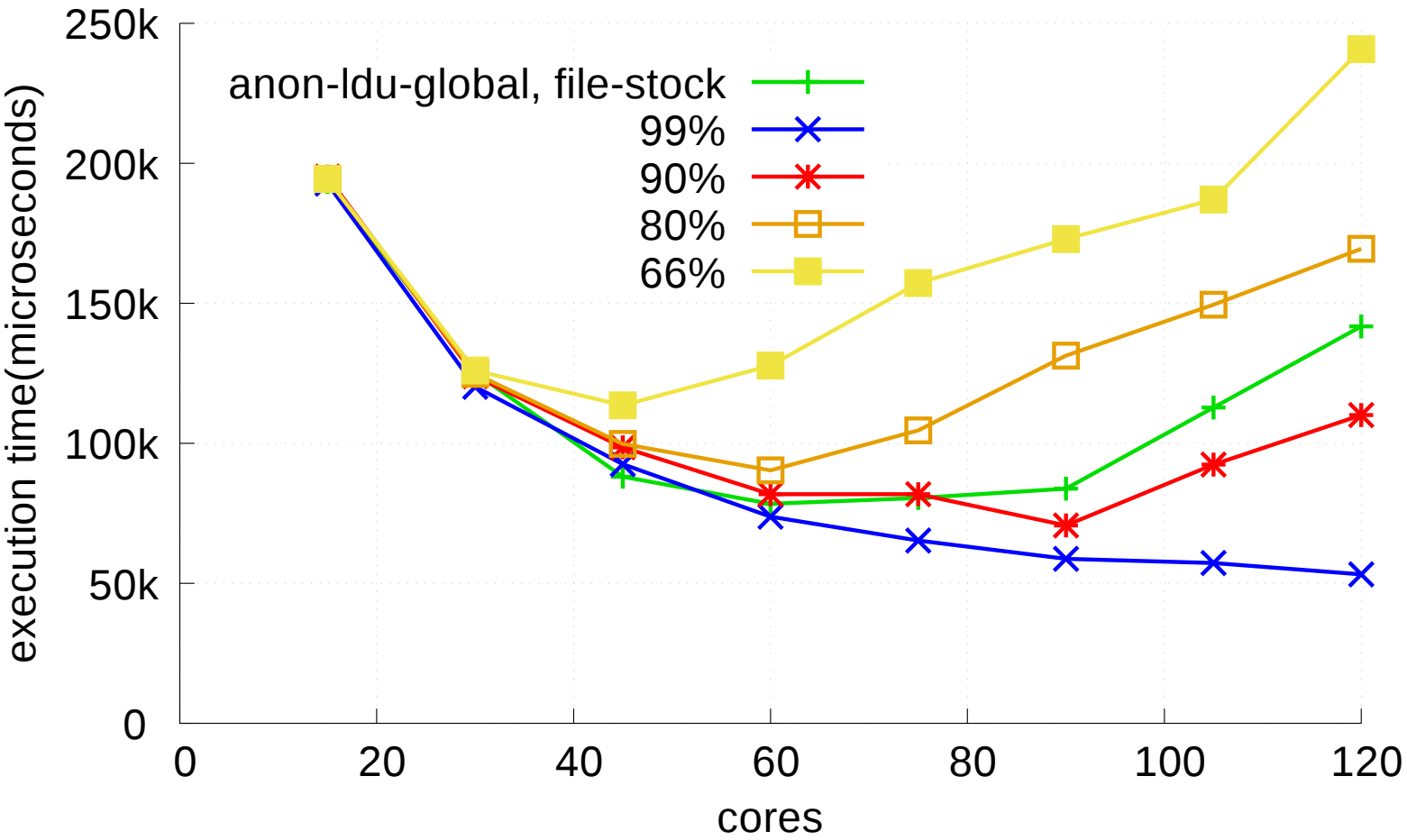
Update ratios – AIM7



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.

Conclusion

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