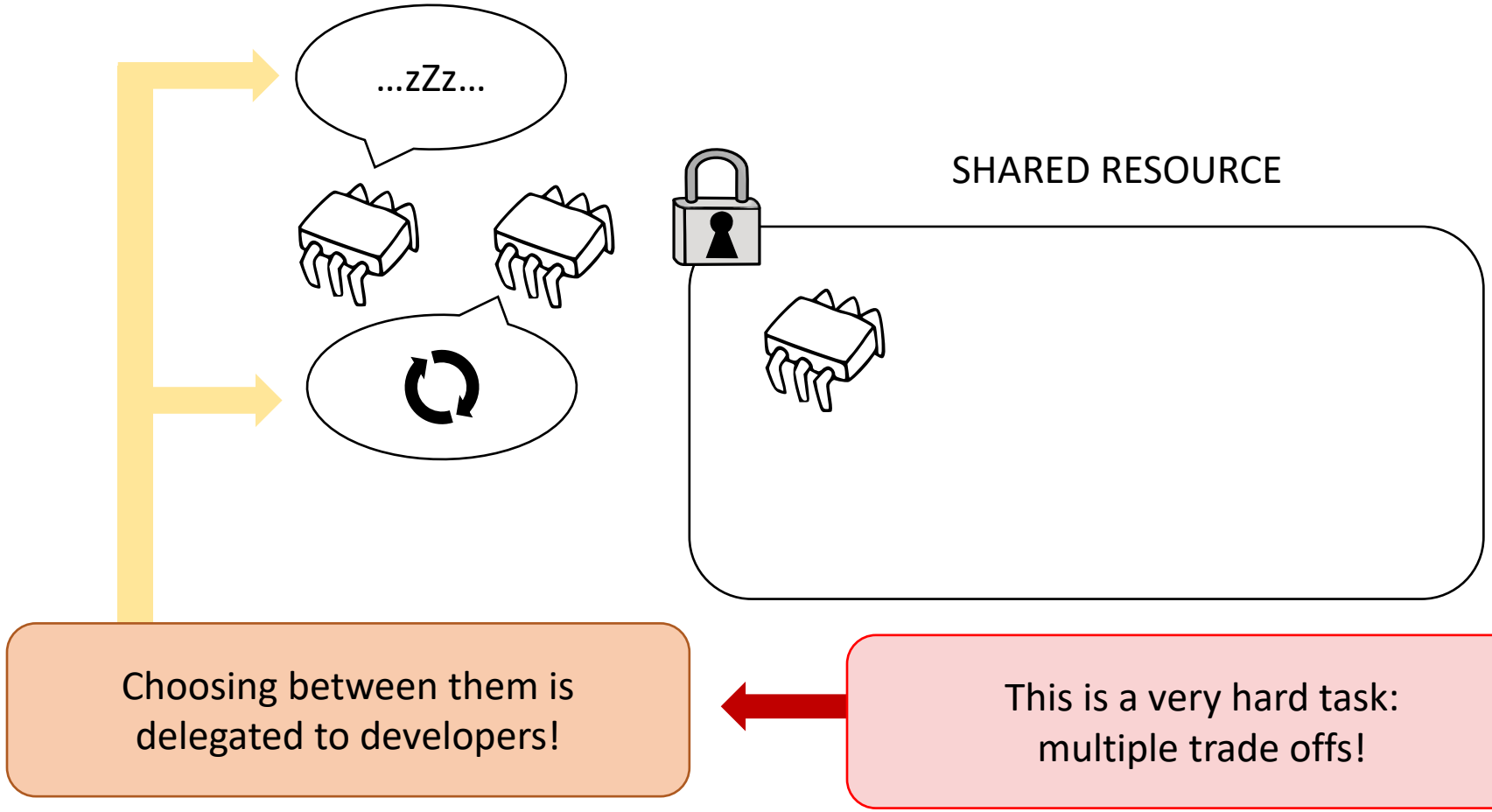
Concurrent and parallel programming

Romolo Marotta

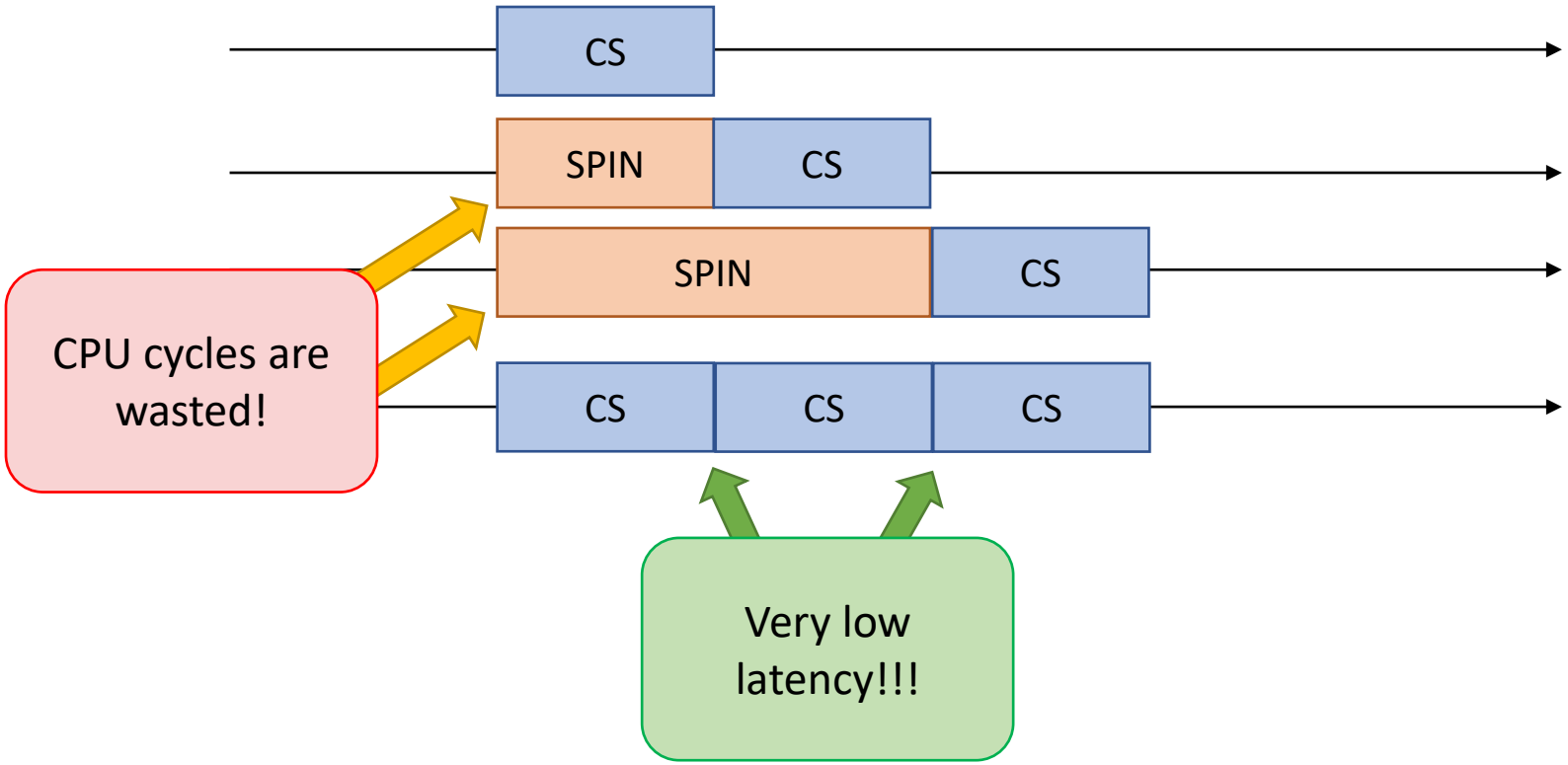
Lock implementations

Blocking coordination



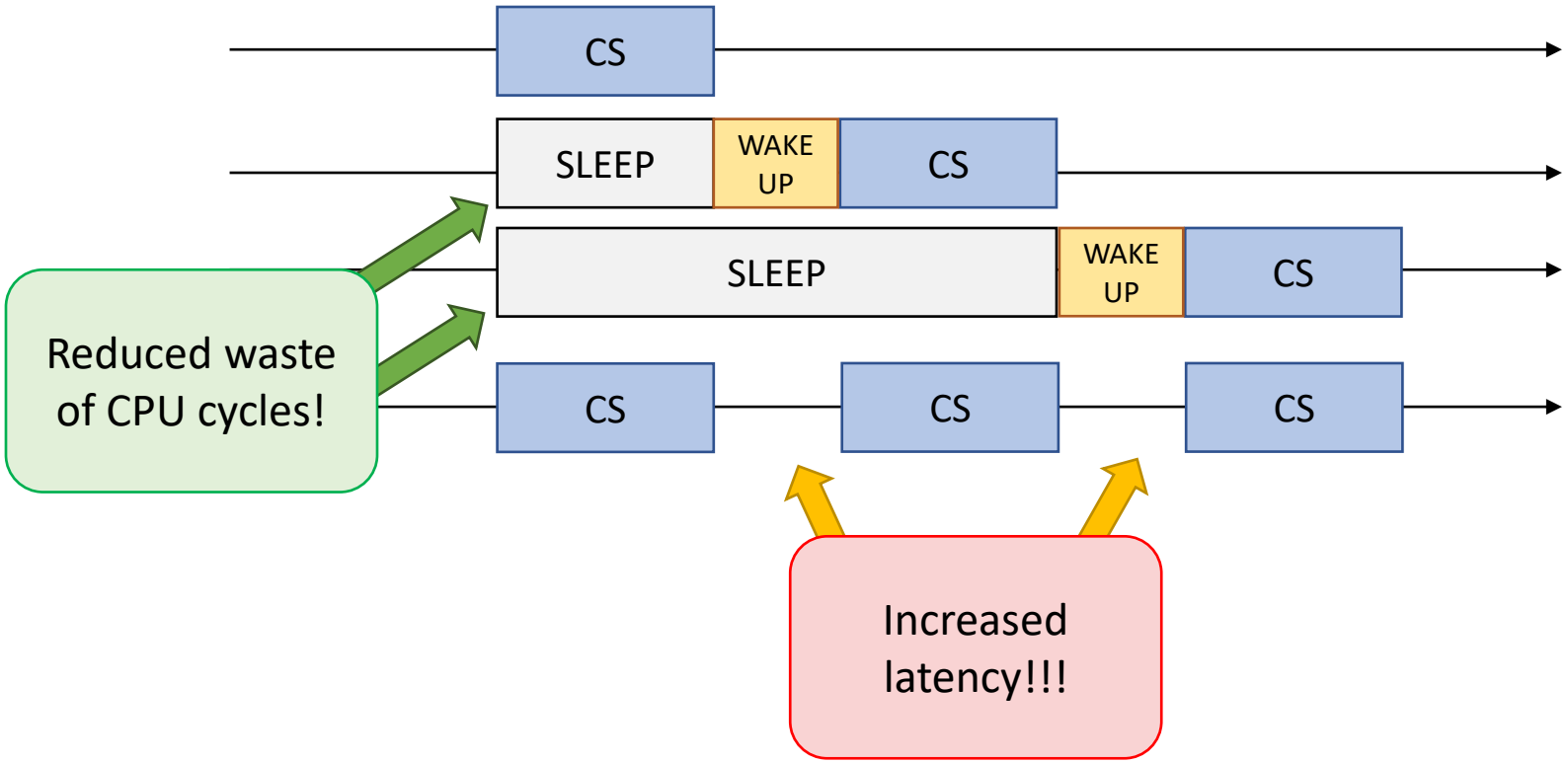
Spinning vs Sleeping

Benefits	Spinning
Guaranteed low latency	✓
Computing power savings	✗



Spinning vs Sleeping

	Waiting Policy	
Benefits	**Spinning**	**Sleeping**
Guaranteed low latency	✓	✗
Computing power savings	✗	✓
Autonomic Adaptivity	✗	✗



Spin vs Sleep – is that all?

- Choosing the proper back off scheme is very challenging
- Even implementing a simple spin lock is not trivial
  - Trade off between low and high contented case
  - You should have heard about algorithms for Mutual Exclusion in Distributed Systems lectures
    - E.g. Dijkstra, Bakery algorithm, Peterson...
  - Those algorithm essentially implements spin locks by resorting only on read/write operations
- Here, we will focus on spin locking algorithms that exploit stronger synchronization primitives… RMW!

Test-and-set spin lock

- Test-and-set lock is the simplest spin lock
- Acquiring threads always try to set a variable via RMW

```
int lock = 0;
void acquire(int *lock){
  while(XCHG(lock, 1));
}
```

```
void release(int *lock){
  *lock = 0;
}
```

A small benchmark

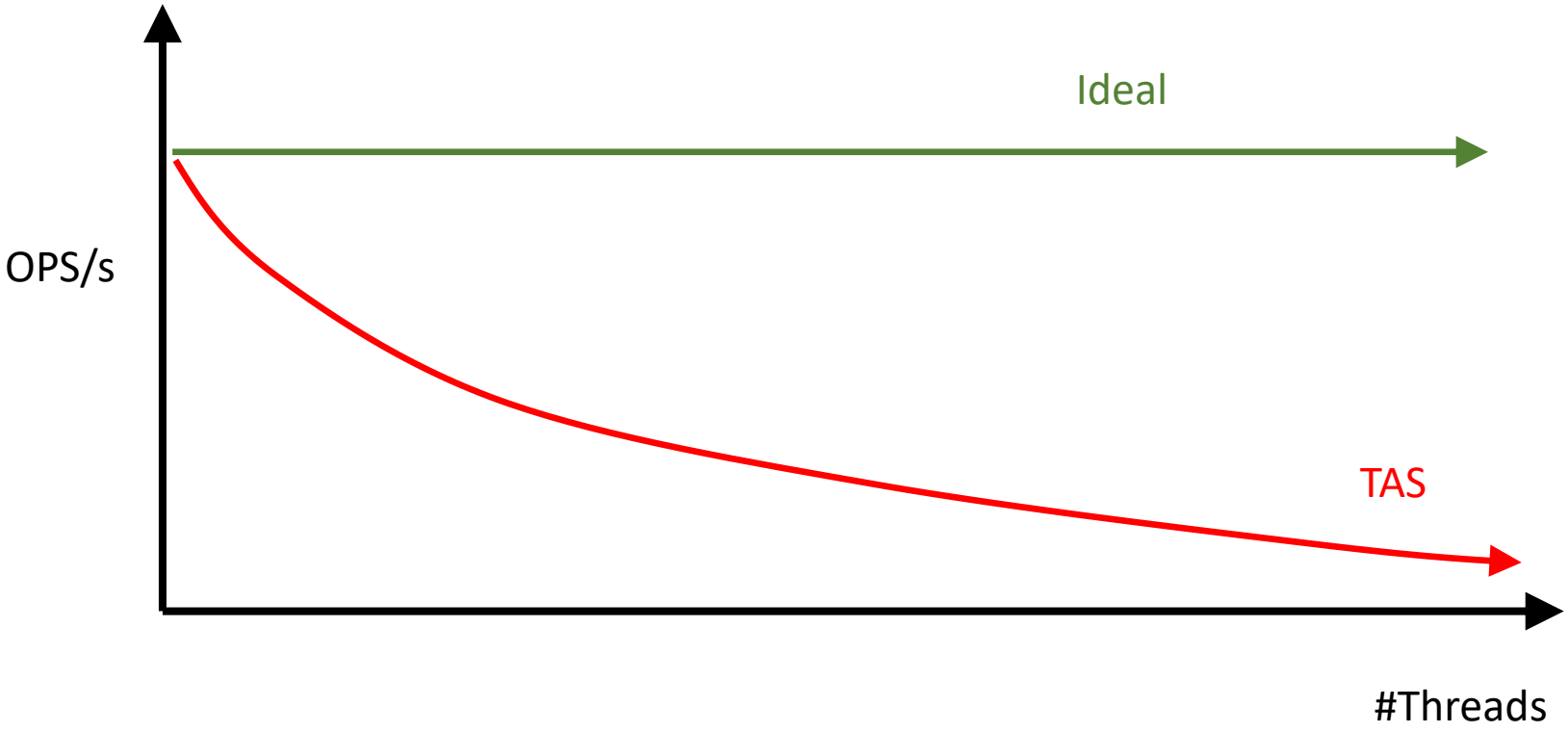
- We have an array of integers
- Each thread reverse the array

- This is done within a critical section

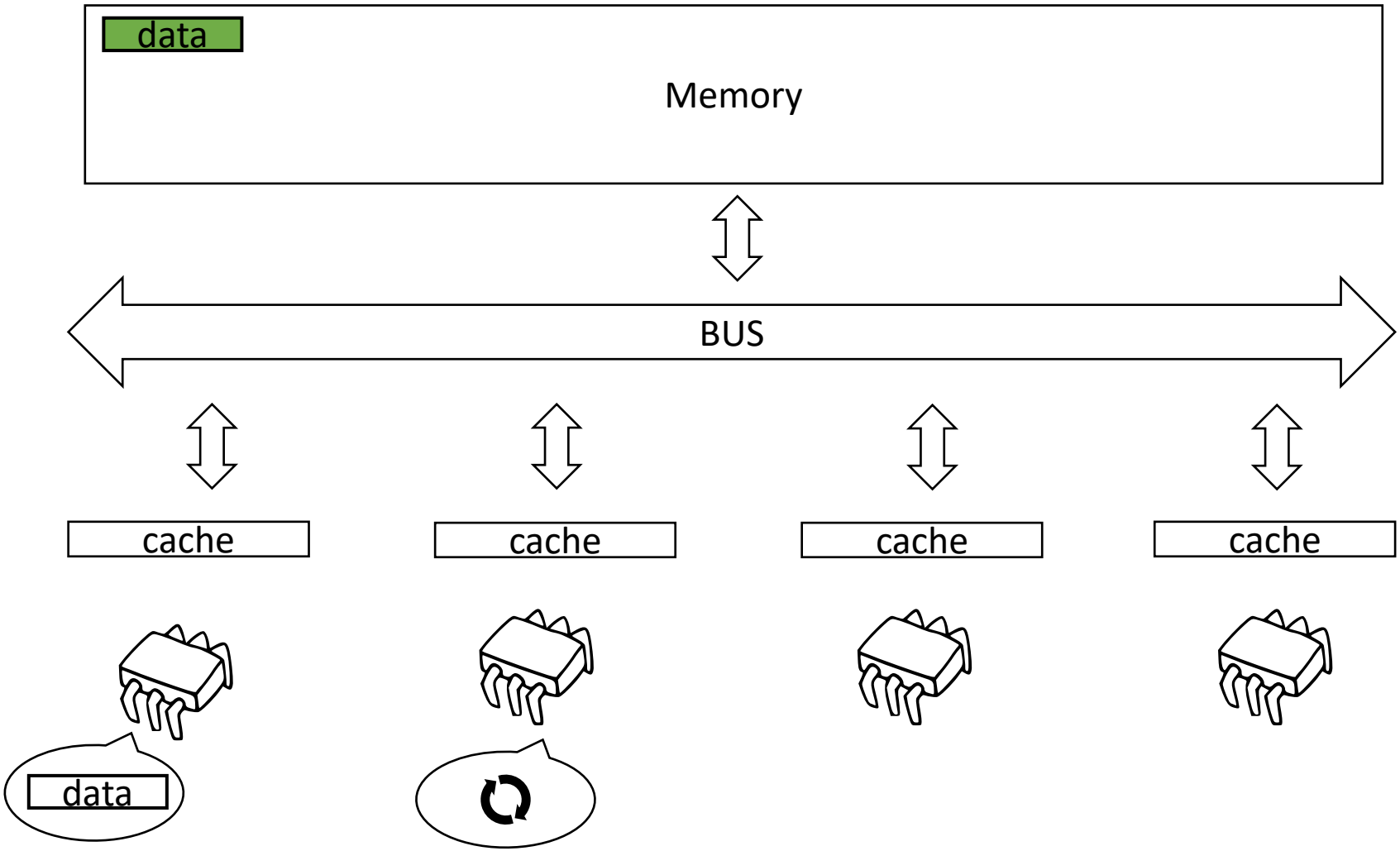
```
while(!stop){
   acquire(&lock);
   flip_array();
   release(&lock);
}
```

- Performance Metric:
  - Throughput = #Flips per second

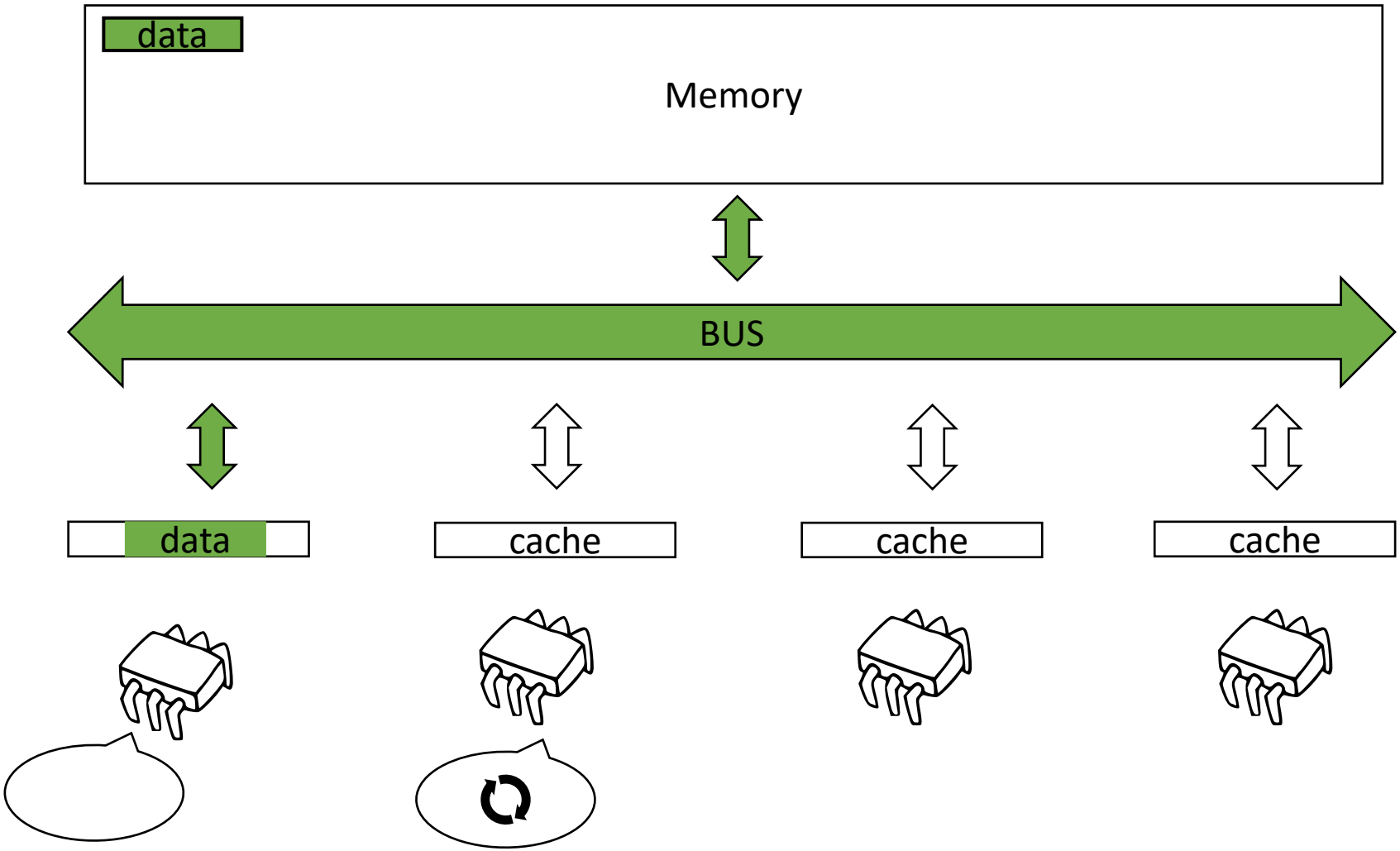
Results



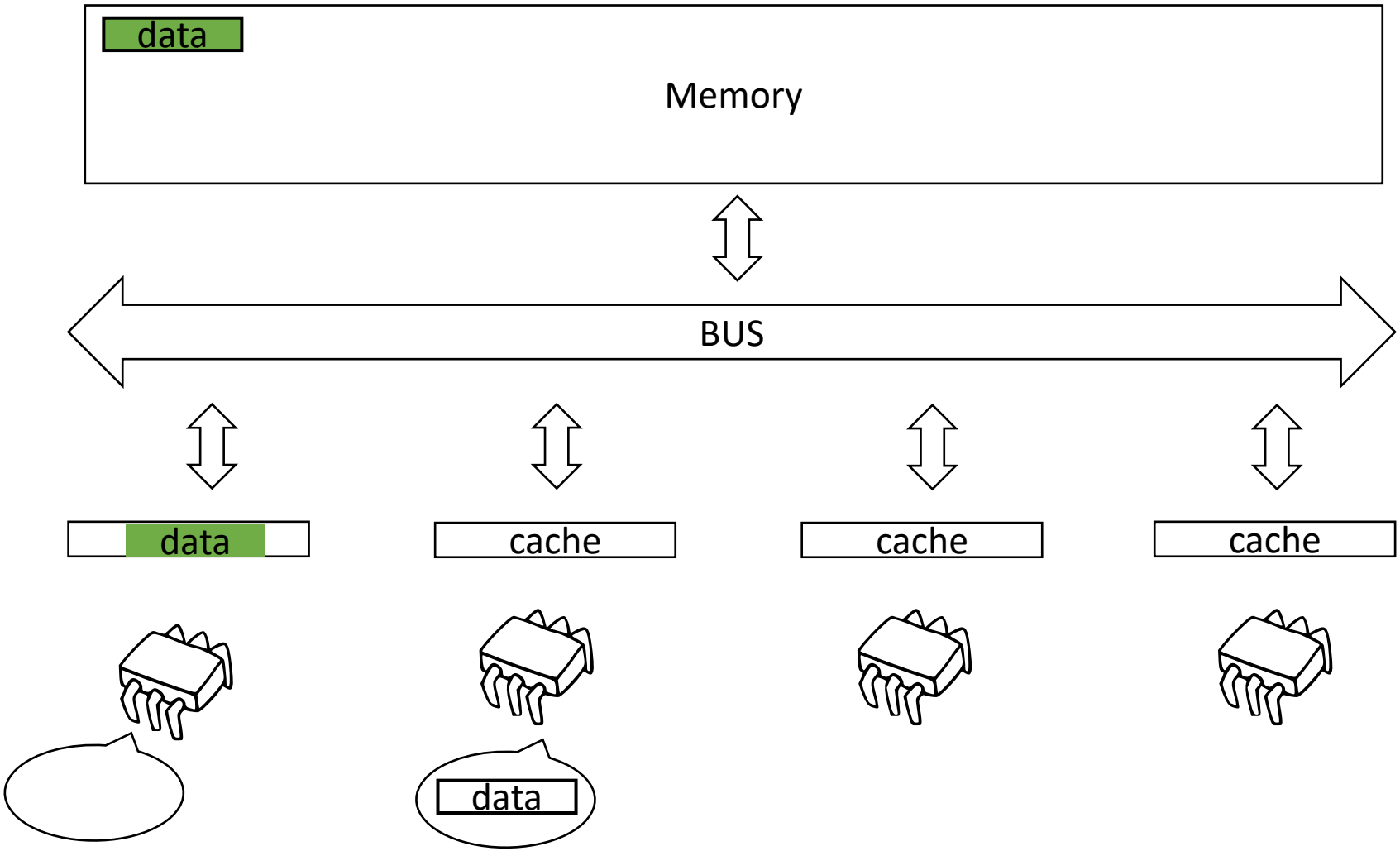
Memory Model



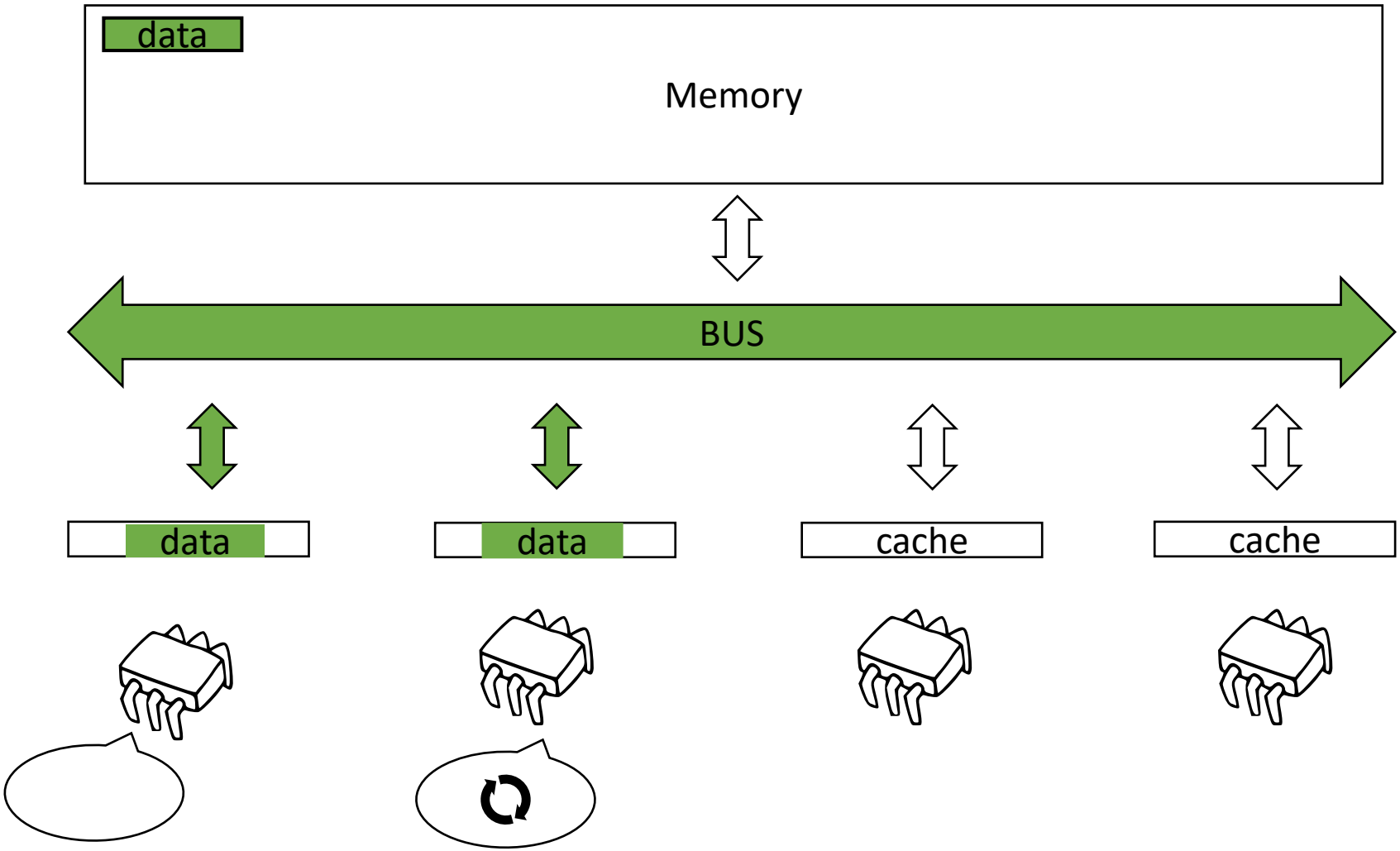
Memory Model



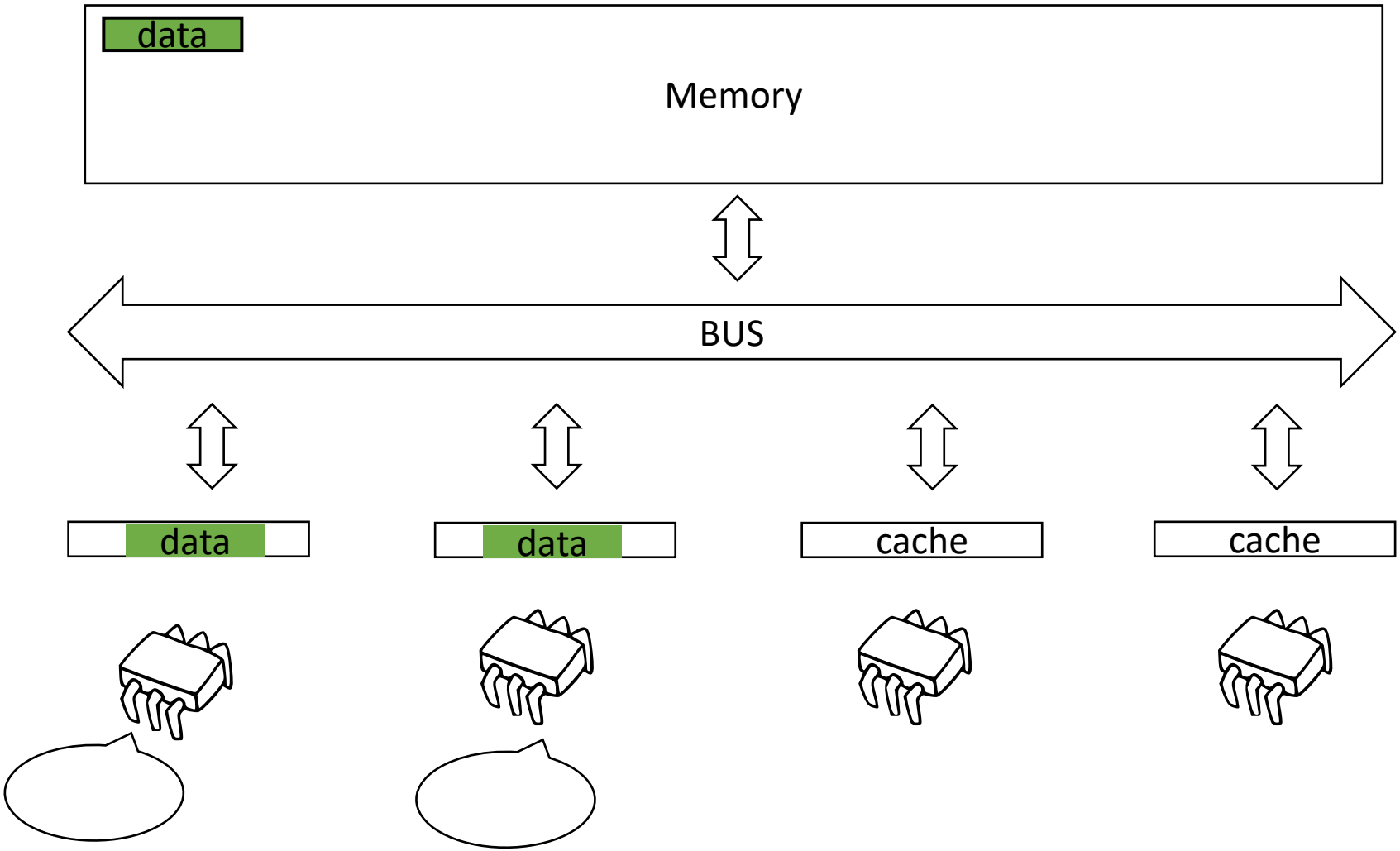
Memory Model



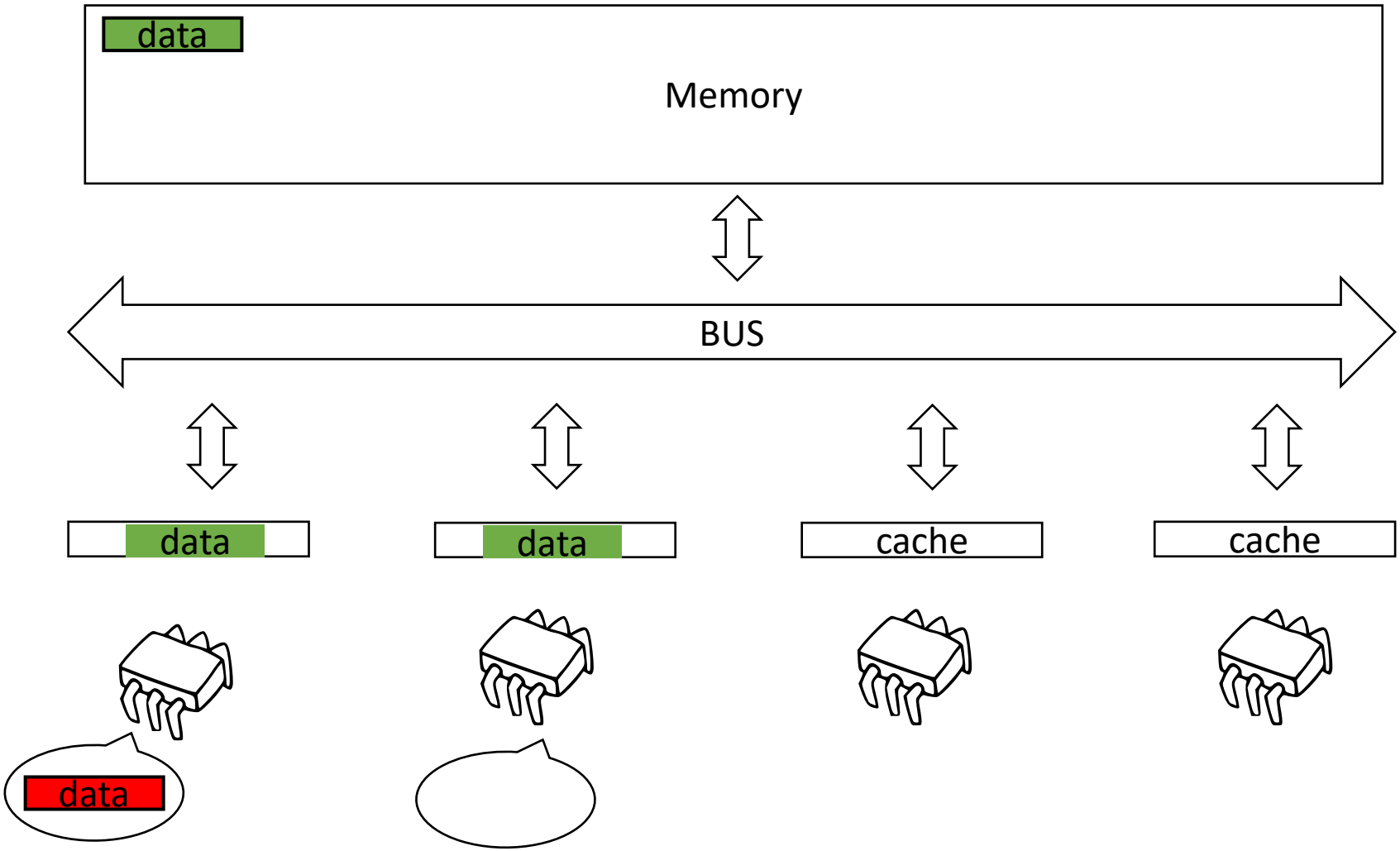
Memory Model



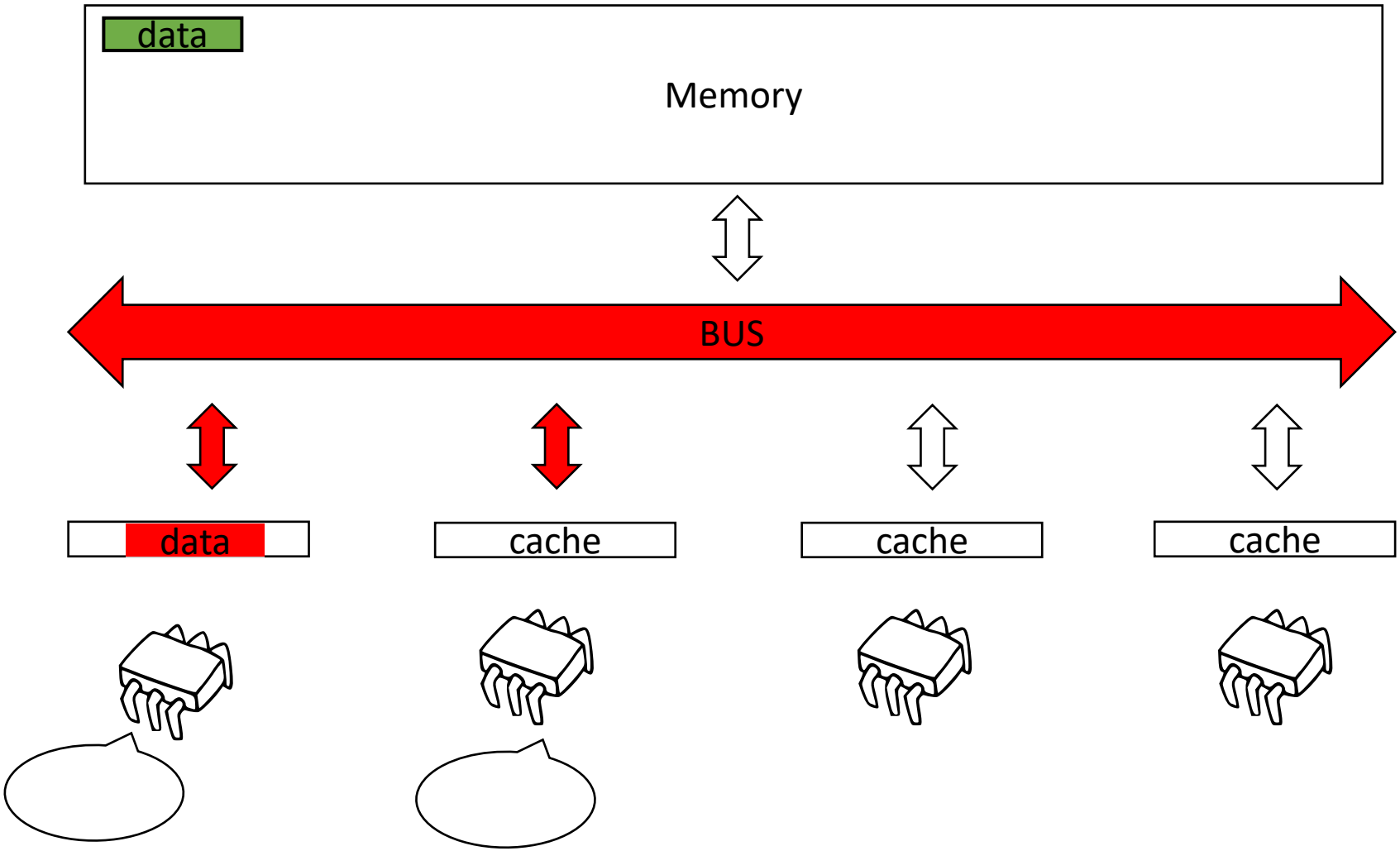
Memory Model



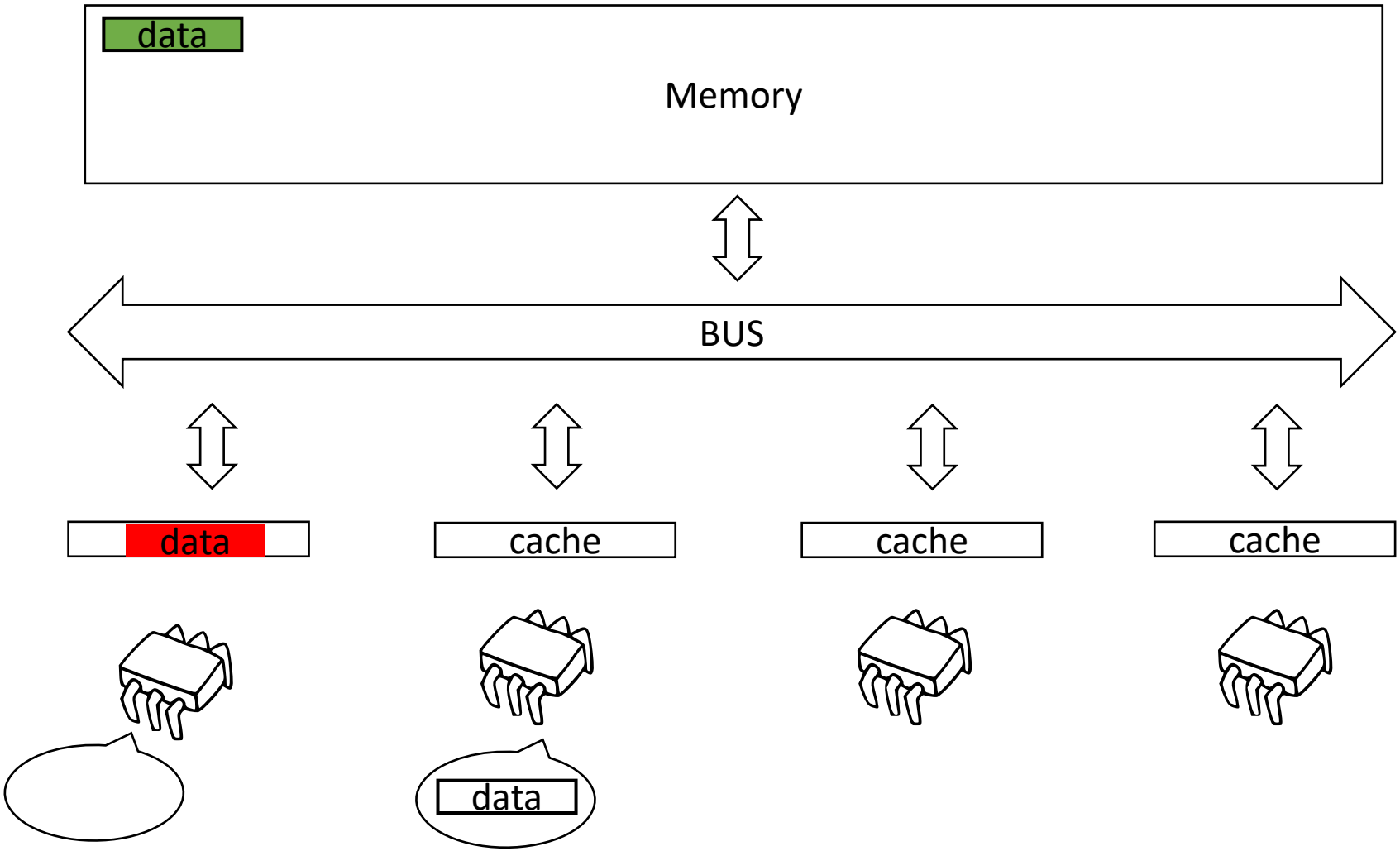
Memory Model



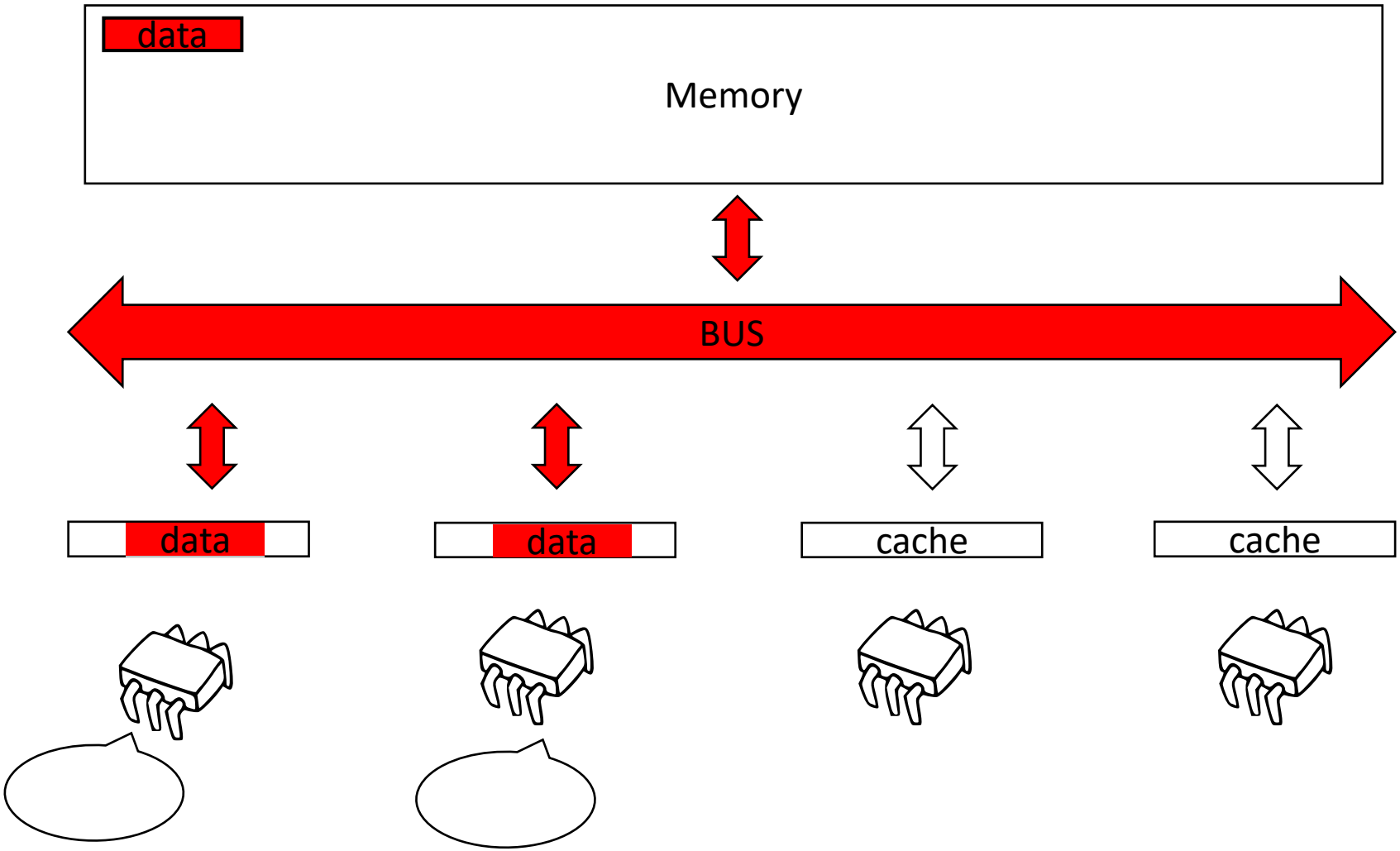
Memory Model



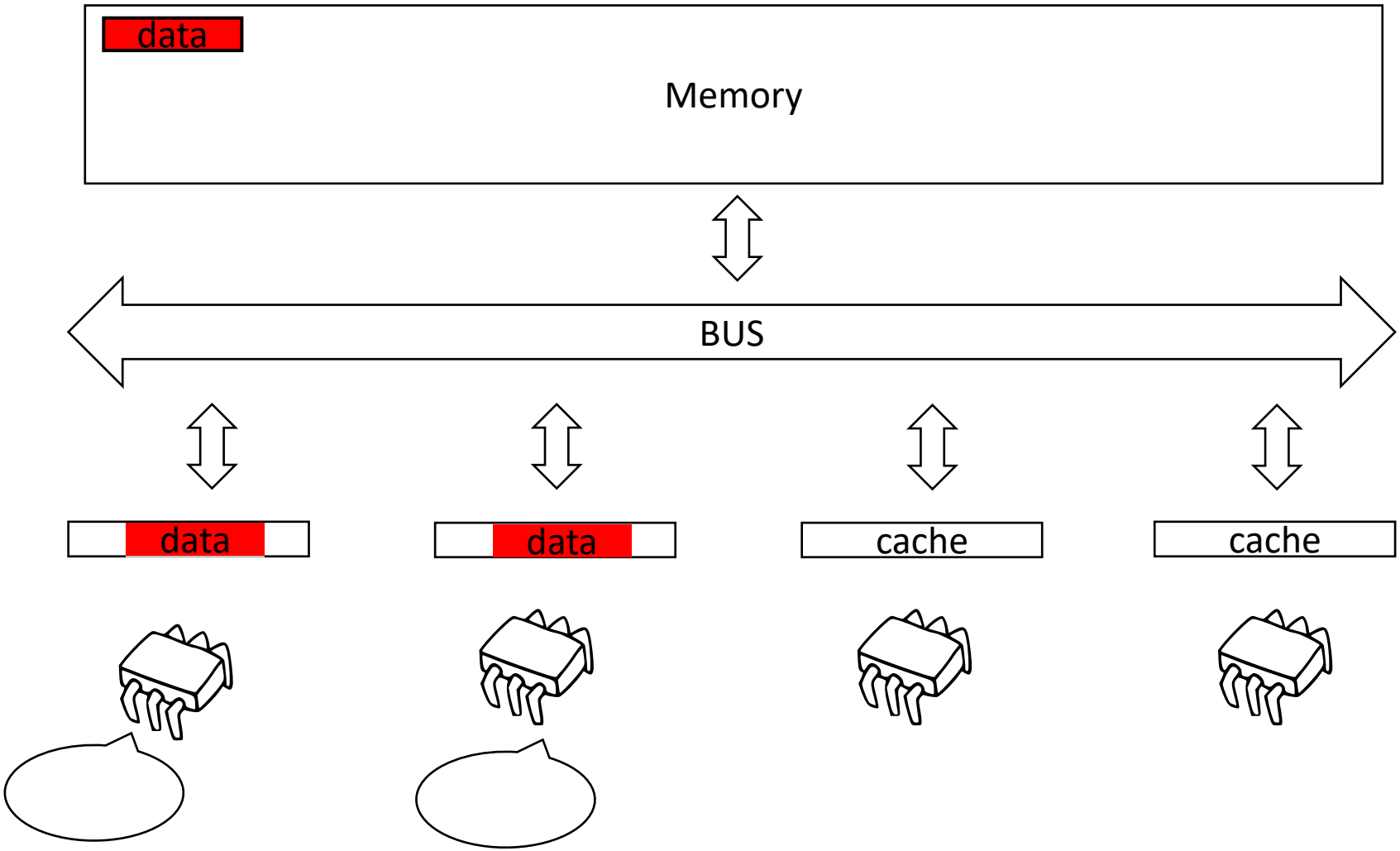
Memory Model



Memory Model



Memory Model



Lock implementations

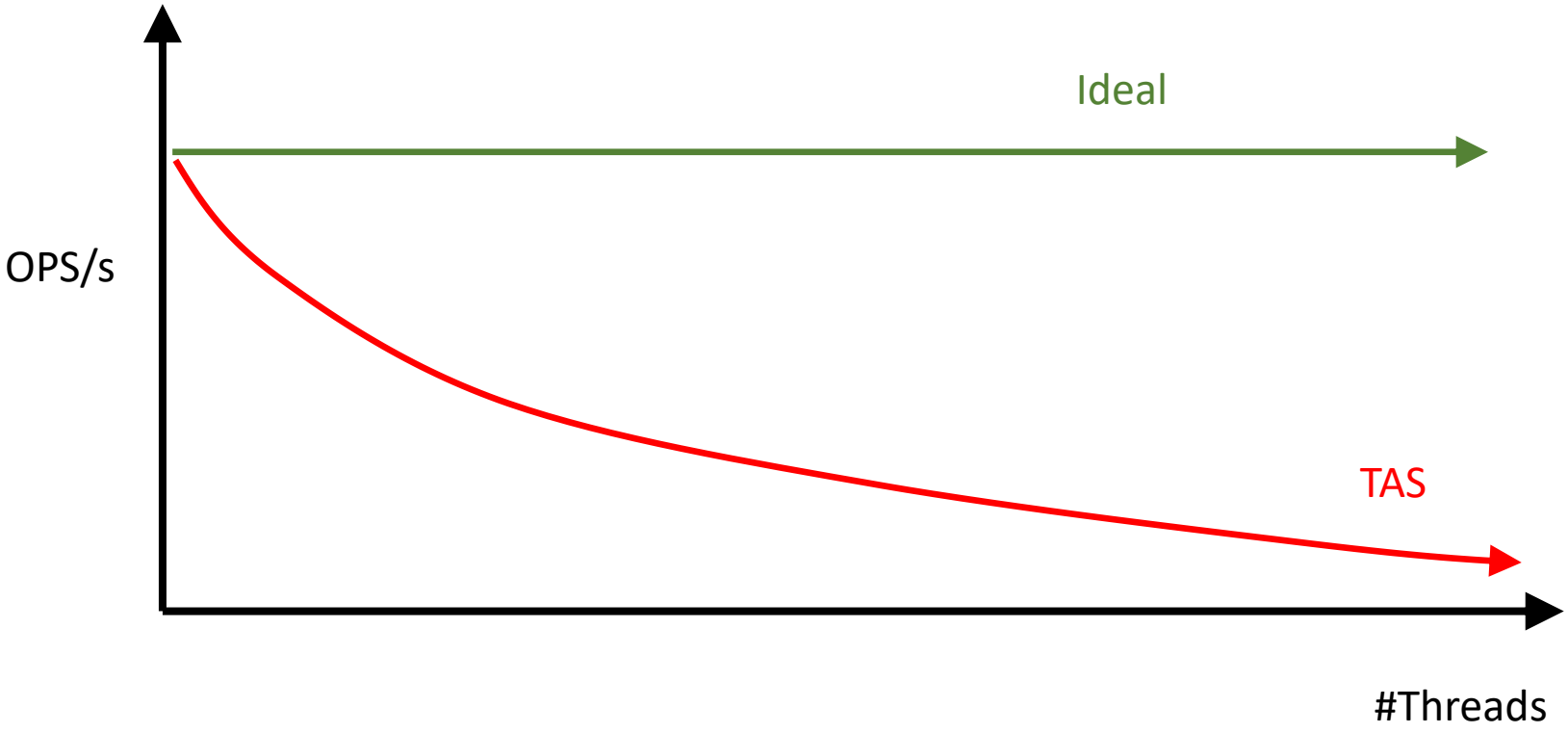
Test-and-set spin lock

- Test-and-set lock is the simplest spin lock
- Acquiring threads always try to set a variable via RMW

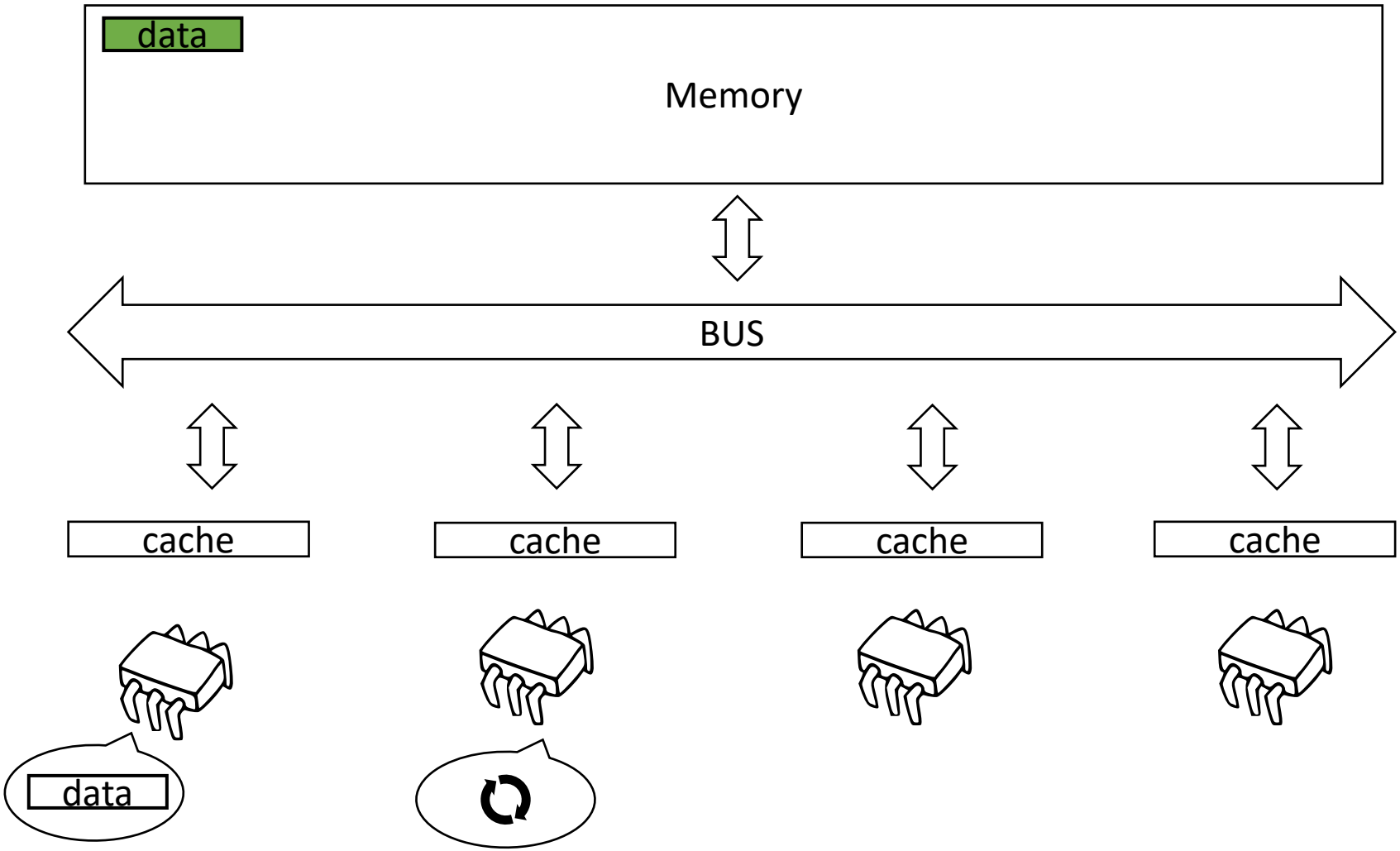
```
int lock = 0;
void acquire(int *lock){
  while(XCHG(lock, 1));
}
```

```
void release(int *lock){
  *lock = 0;
}
```

Results



Memory Model

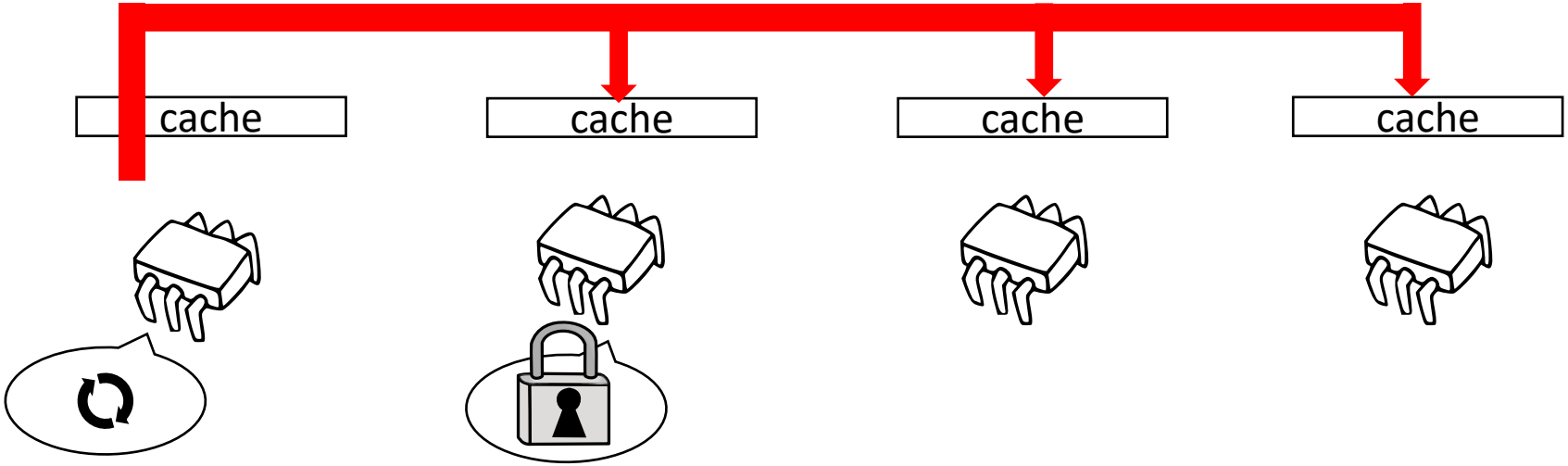


Test-and-set spin lock

- Test-and-set lock is the simplest spin lock
- Acquiring threads always try to set a variable via RMW

```
int lock = 0;
void acquire(int *lock){
  while(XCHG(lock, 1));
}
```

```
void release(int *lock){
  *lock = 0;
}
```

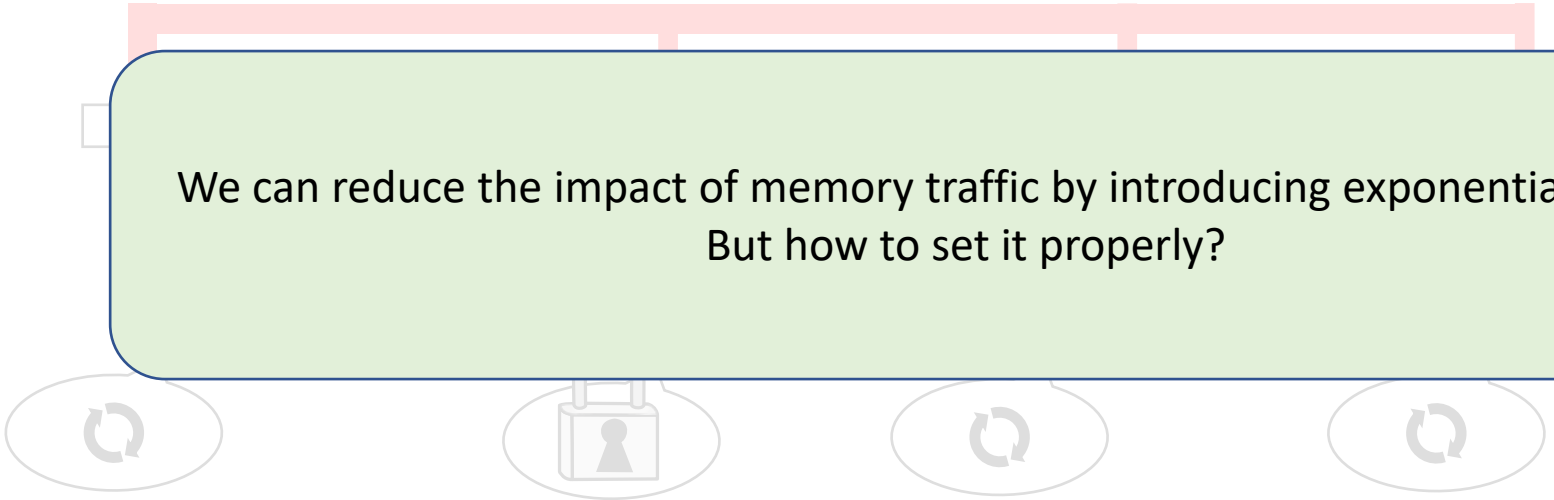


Test-and-set spin lock

- Test-and-set lock is the simplest spin lock
- Acquiring threads always try to set a variable via RMW

```
int lock = 0;
void acquire(int *lock){
  while(XCHG(lock, 1));
}
```

```
void release(int *lock){
  *lock = 0;
}
```



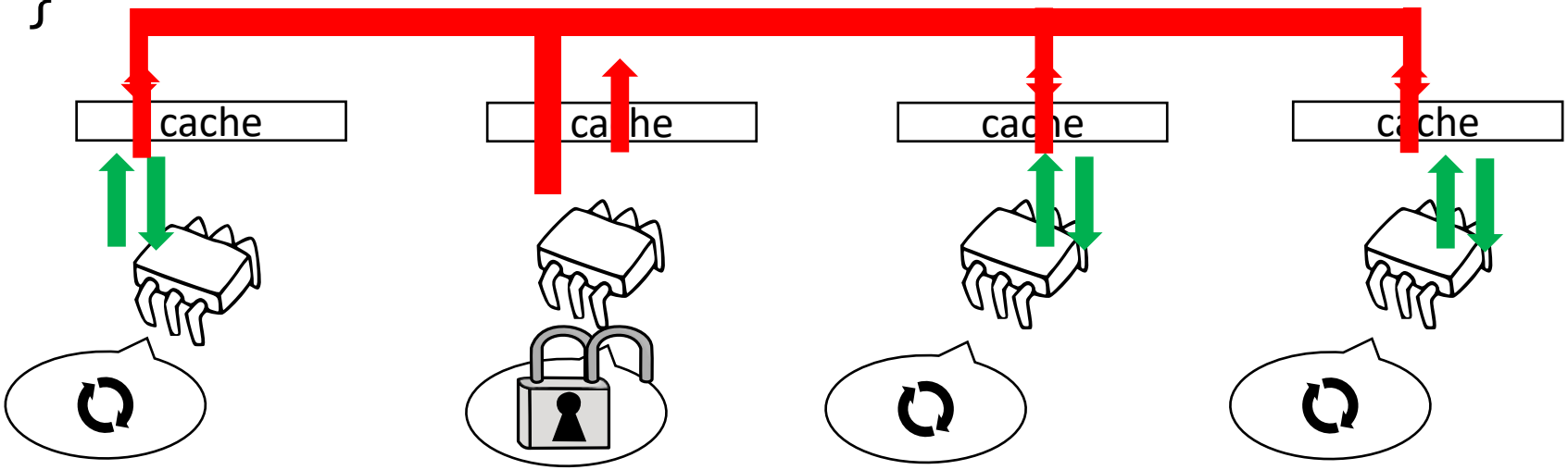
We can reduce the impact of memory traffic by introducing exponential back off!
But how to set it properly?

Test-and-test-and-set spin lock

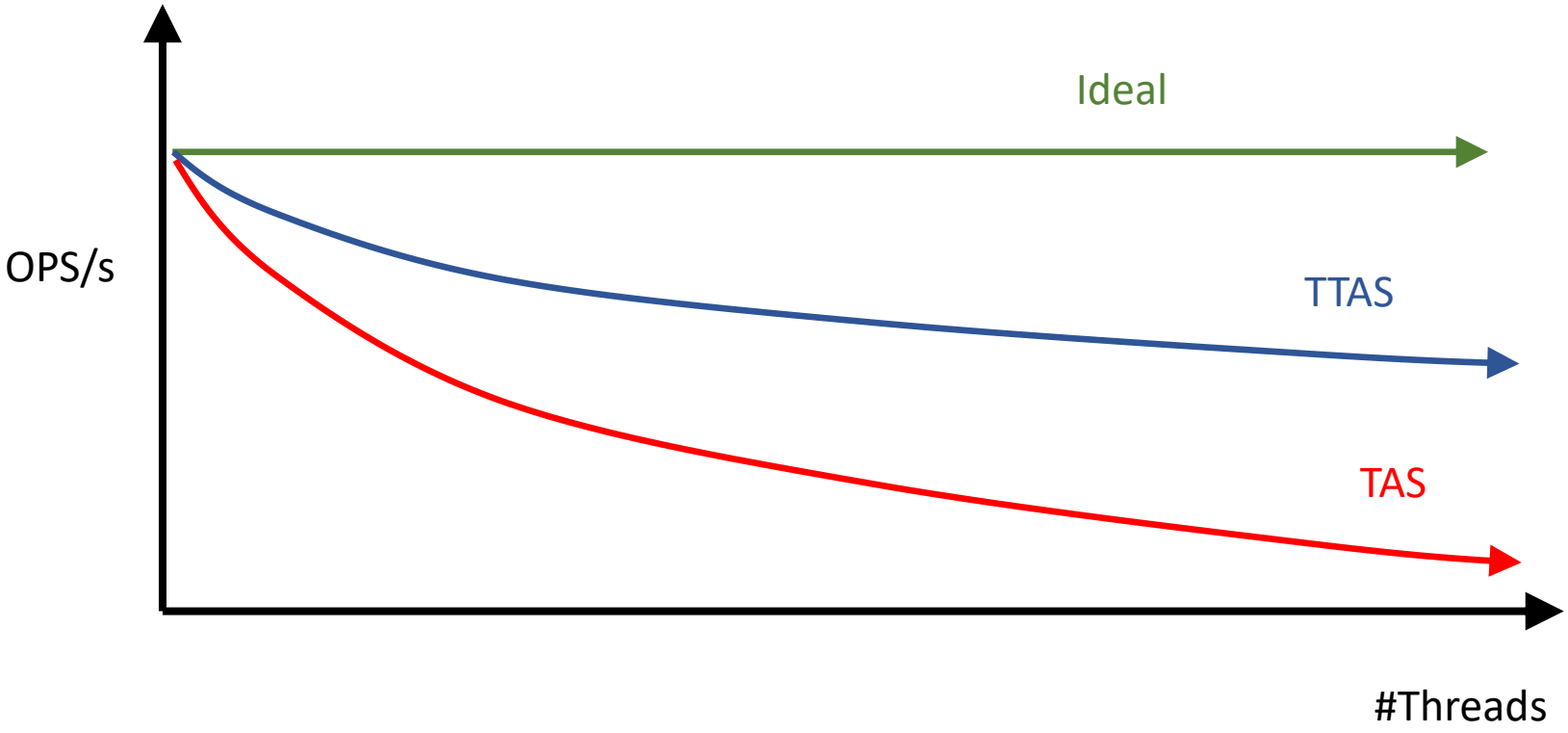
- Like test-and-set, but spins by reading the value of the lock
- Traffic is generated only upon lock handover

```
int lock = 0;
void acquire(int *lock){
  while(XCHG(lock, 1))
    while(*lock);
}
```

```
void release(int *lock){
  *lock = 0;
}
```



Results



Test-and-test-and-set spin lock

- Like test-and-set, but spins by reading the value of the lock
- Traffic is generated only upon lock handover

```
int lock = 0;
void acquire(int *lock){
  while(XCHG(lock, 1))
    while(*lock);
}
```

```
void release(int *lock){
  *lock = 0;
}
```

- Lock handover costs increase with the concurrency level
- Very lightweight for the uncontended case
- Is it feasible reducing handover costs?
- AND IMPROVING FAIRNESS?

FIFO locks

Ticket locks

- Similar to the bakery algorithm but it uses RMW instructions
- Two variables
  - The next available ticket
  - The served ticket

```
typedef struct _tck_lock{
  int ticket  = 0;
  int current = 0;
} tck_lock;
```

```
void acquire(tck_lock *lock){
   int cur_tck;
   int mytck = fetch&add(lock->ticket, 1);
   while(mytck != (cur_tck = lock->current) )
      delay((mytck-cur_tck)*BASE);
}
void release(tck_lock *lock){ lock->current += 1; }
```

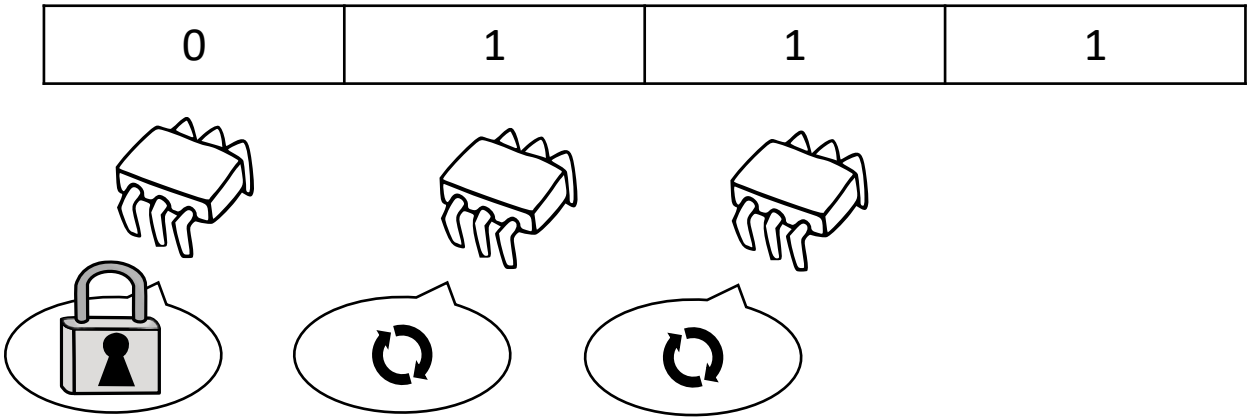
Ticket locks

- Ensure fairness
- Similar structure w.r.t. TTAS spinlock
  - One variable updated once at each acquisition (better than TTAS)
  - Write-1-Read-N variable updated at each release (same as TTAS)
- How?

Anderson queue lock

- Use similar to ticket lock
- Use the ticket to obtain an individual cache line

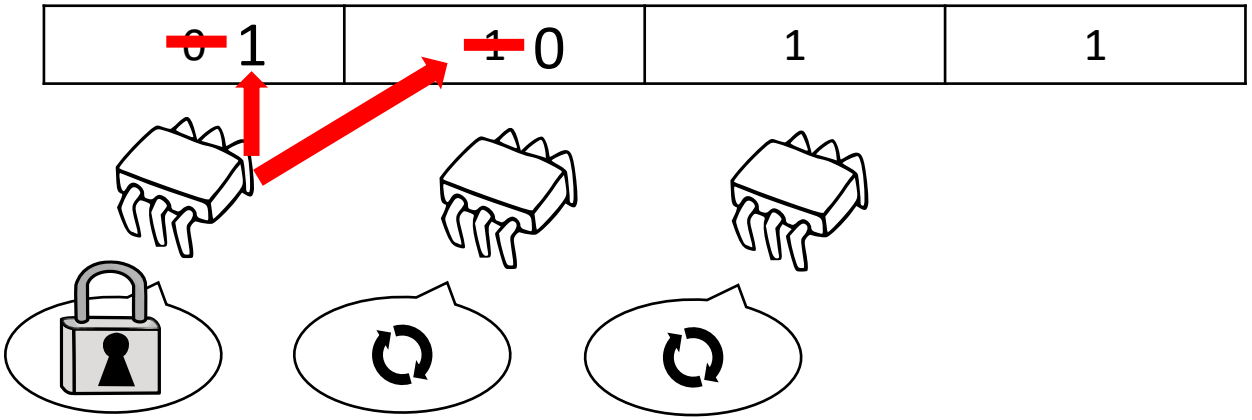
Ticket = ~~0 1 2~~ 3



Anderson queue lock

- Use similar to ticket lock
- Use the ticket to obtain an individual cache line

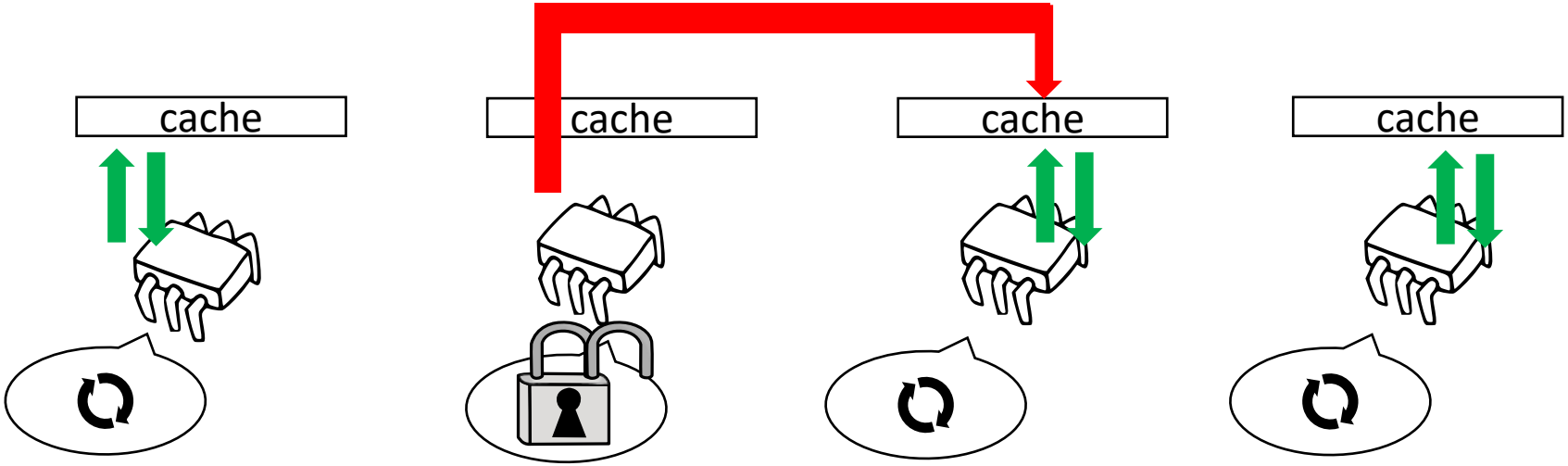
Ticket = ~~0 1 2~~ 3



Anderson queue lock

```
void acquire(anderson_lock *lock){
  mytck = fetch&add(lock->ticket, 1);
  while(lock->array[mytck]);
  lock->array[mytck] = 1;
}
```

```
void release(int *lock){
 lock->array[mytck+1] = 0;
}
```

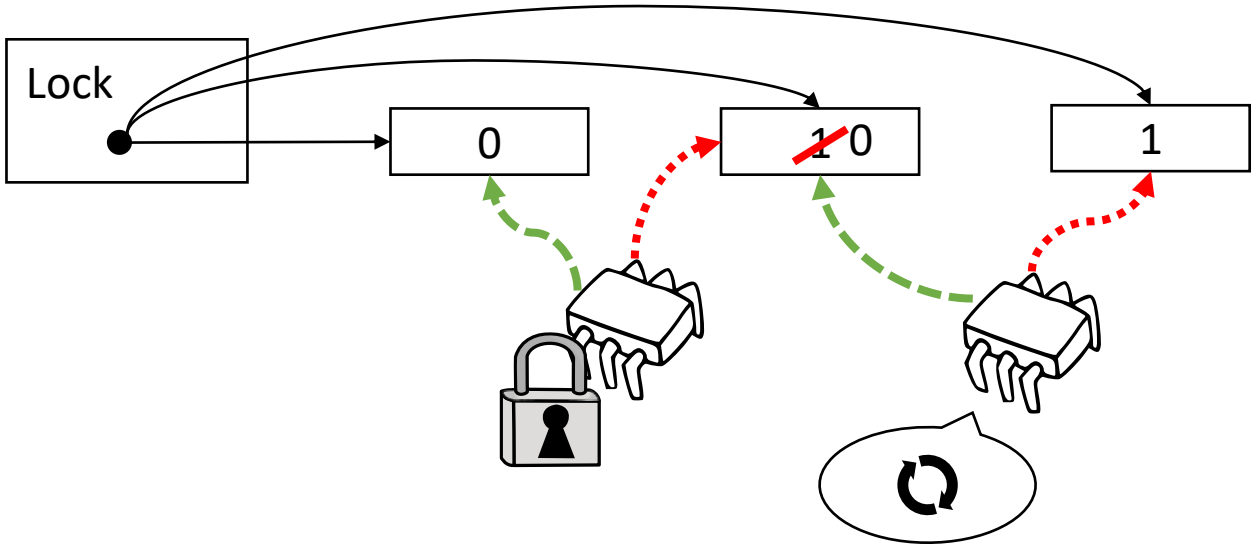


Anderson queue lock

- Pros:
  - One variable updated once at each acquisition (like Ticket lock)
  - Write-1-Read-1 variable updated once per release (better than (T)TAS and Ticket)
- Cons:
  - Increased memory footprint
  - Each lock needs to know the maximum number of threads
- Let:
  - T be the number of threads
  - L be the number of locks
- Space Usage
  - Anderson = O(LT)
  - TAS, TTAS, Ticket = O(L)

CLH lock

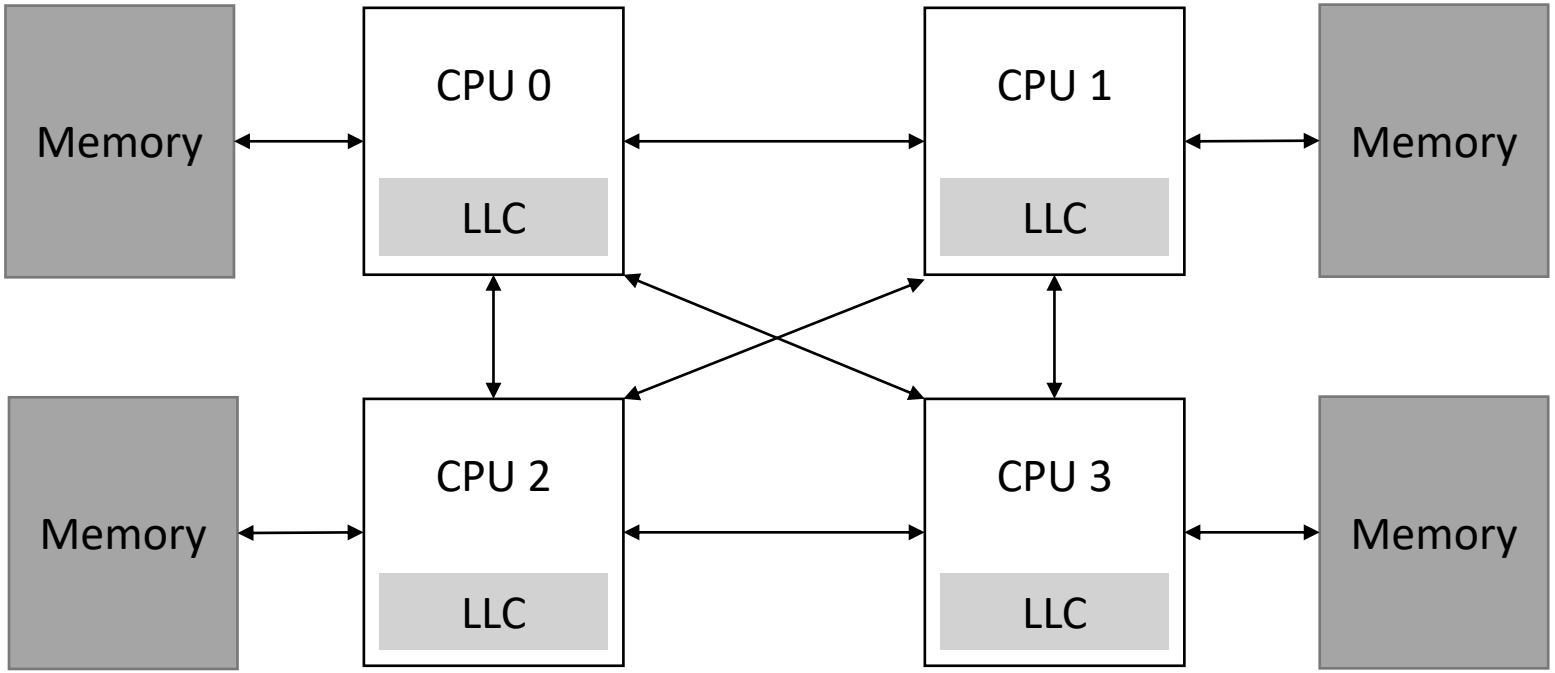
- An (implicit) linked list maintains the order between waiting threads
- An empty list represent an uncontended lock
- An arriving thread swaps the node with its private node
- Spin on the previous node
- Release on the new node



CLH queue lock

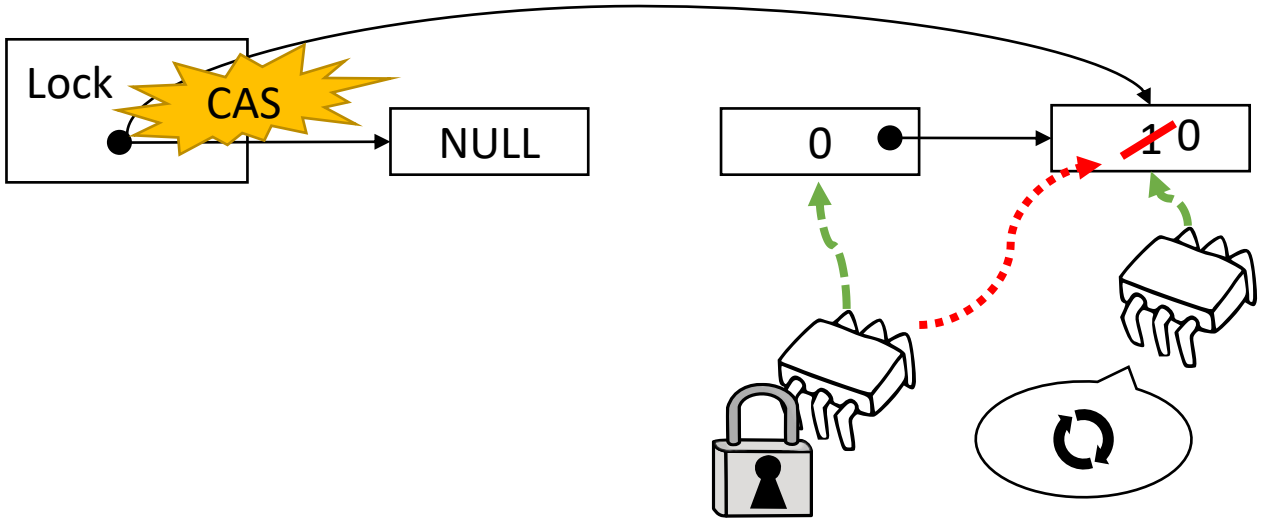
- Pros:
  - One variable updated once at each acquisition (like Ticket lock)
  - Write-1-Read-1 variable updated once per release (better than (T)TAS and Ticket)
- Cons:
  - Slightly increased memory footprint
- Let:
  - T be the number of threads
  - L be the number of locks
- Space Usage
  - CLH = O(L+T)
  - Anderson = O(LT)
  - TAS, TTAS, Ticket = O(L)

NUMA


Memory
CPU 0
LLC
CPU 1
LLC
Memory
Memory
CPU 2
LLC
CPU 3
LLC
Memory

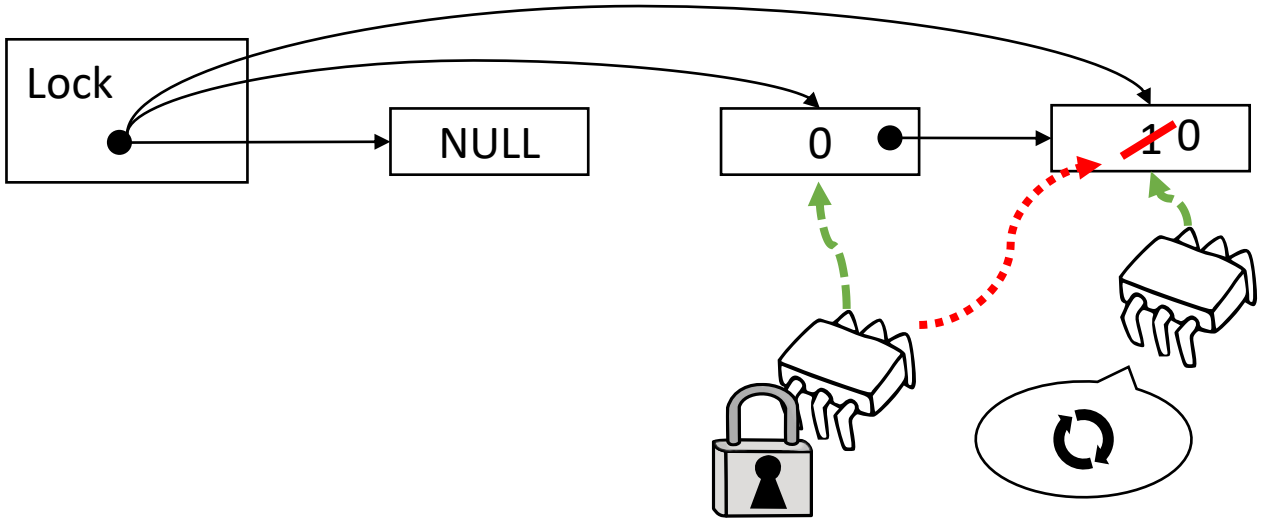
MCS lock

- An explicit linked list maintains the order between waiting threads
- An empty list represent an uncontended lock
- An arriving thread swaps the node with its private node
- Spin on the just inserted node
- Release on the new node



MCS lock

- An explicit linked list maintains the order between waiting threads
- An empty list represent an uncontended lock
- An arriving thread swaps the node with its private node
- Spin on the just inserted node
- Release on the new node

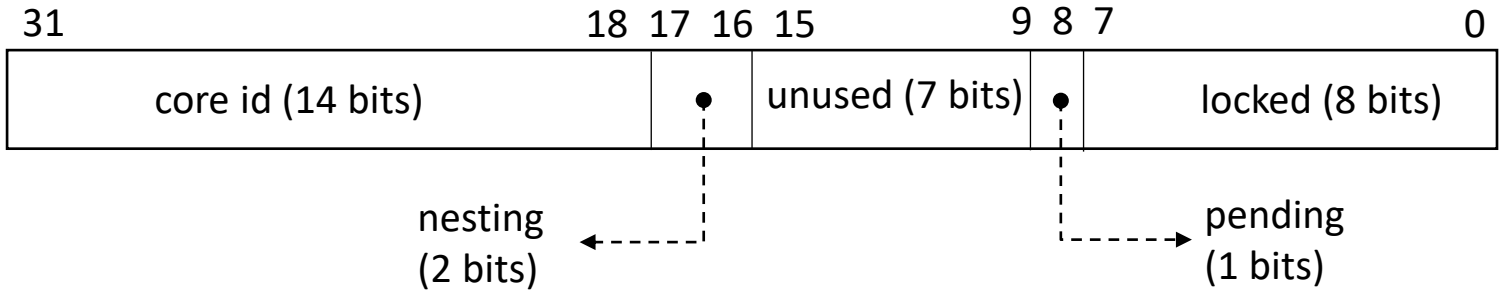


MCS queue lock

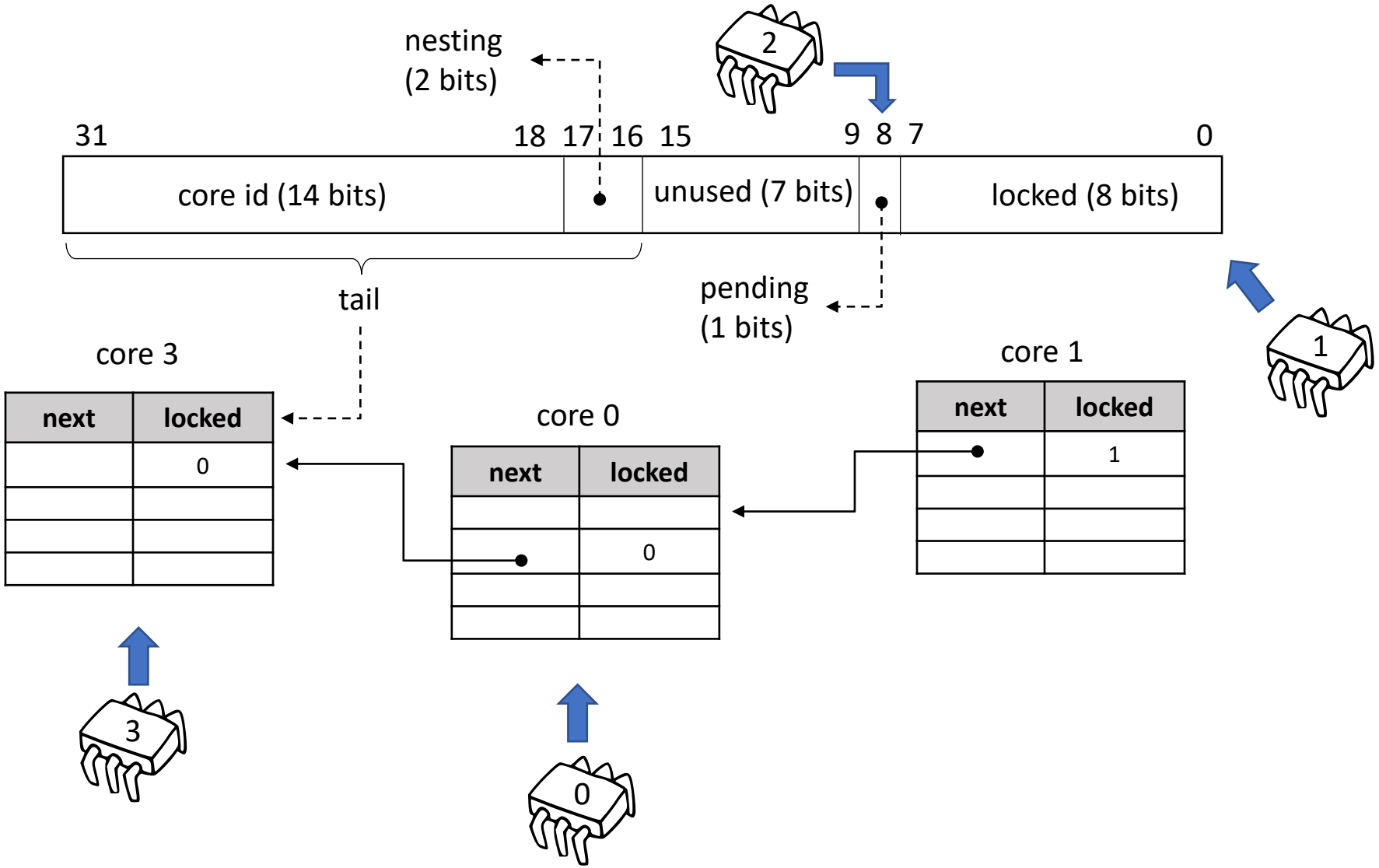
- Pros:
  - One variable updated once at each acquisition (like Ticket lock)
  - Write-1-Read-1 variable updated once per release (better than (T)TAS and Ticket)
  - No-remote spinning
- Cons:
  - Slightly increased memory footprint
- Let:
  - T be the number of threads
  - L be the number of locks
- Space Usage
  - MCS, CLH = O(L+T)
  - Anderson = O(LT)
  - TAS, TTAS, Ticket = O(L)

MCS in practice: the Linux kernel case

- The Linux kernel uses a particular implementation of a MCS lock: Qspinlock
- Additional challenge:
  - Maintain compatibility with classical 32-bit locks
  - MCS uses pointers (64-bit)
- Compact data:
  1. No recursion of same context in critical sections
  2. 4 different contexts (task, softirq, hardirq, nmi)
  3. Finite number of cores
- Use an additional bit for fast lock handover



MCS in practice: the Linux kernel case



A small benchmark

- We have an array of integers
- Each thread reverse the array

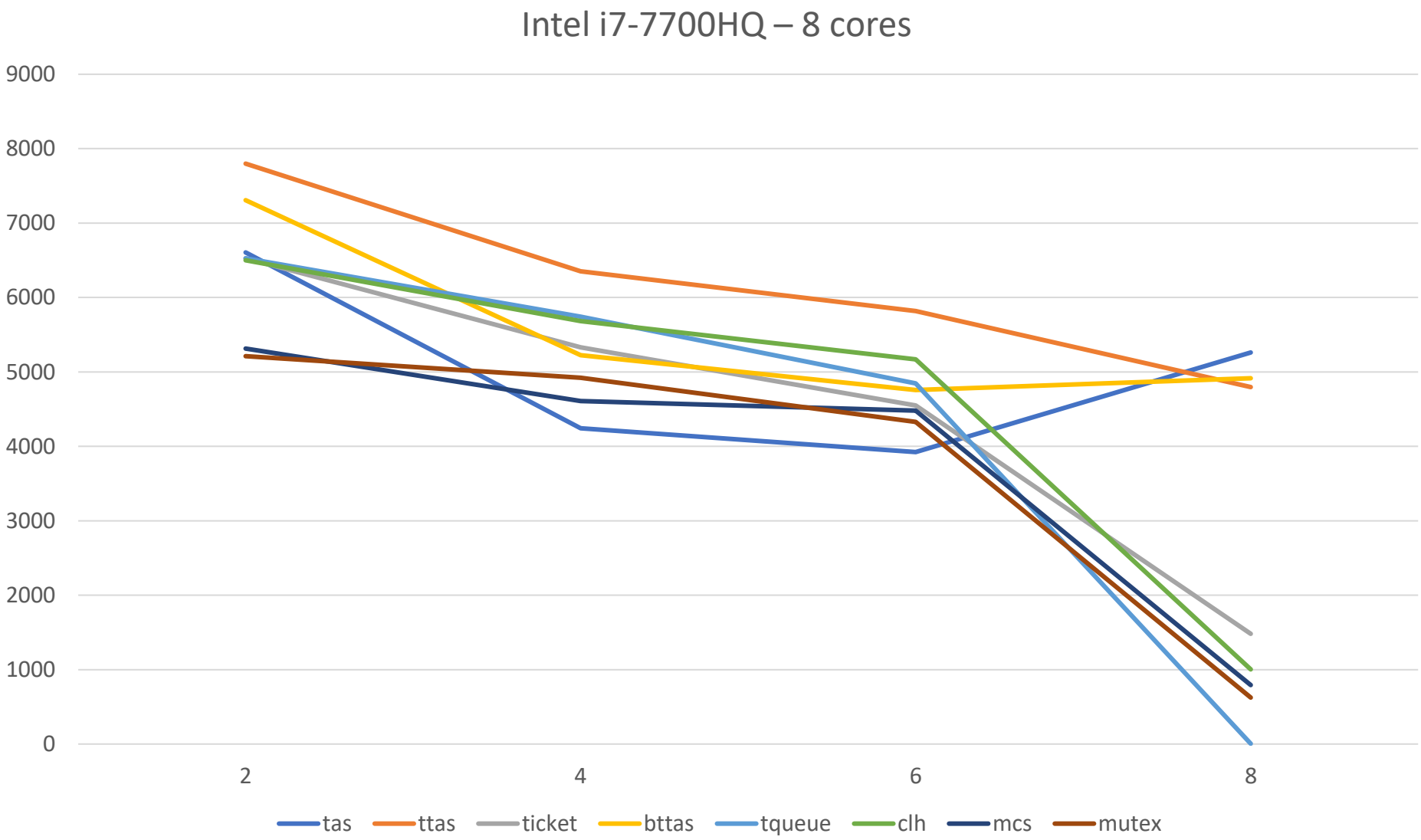
- This is done within a critical section

```
while(!stop){
   acquire(&lock);
   flip_array();
   release(&lock);
}
```

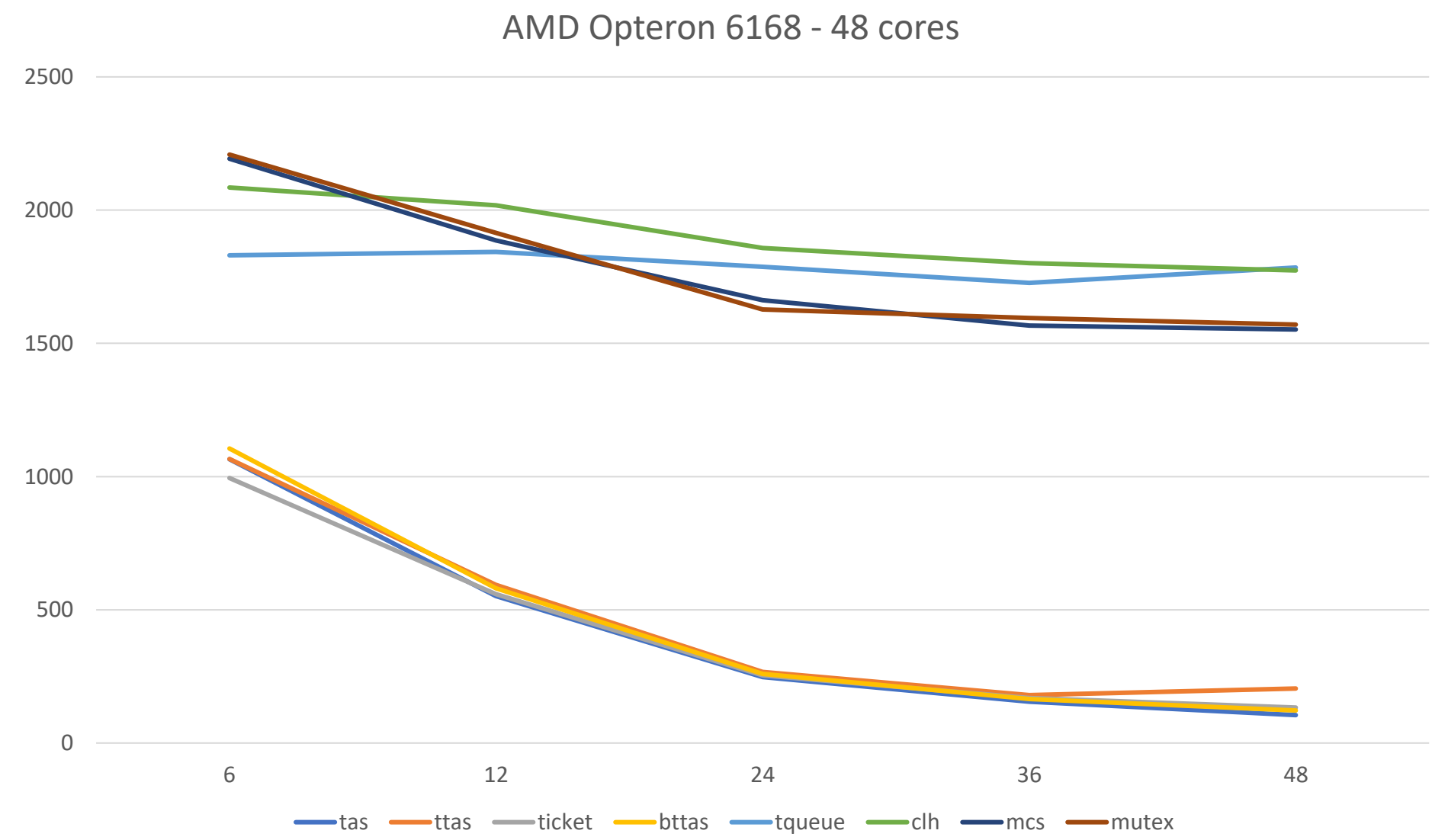
- Performance Metric:
  - Throughput = #Flips per second

One lock
to rule them all…

Performance

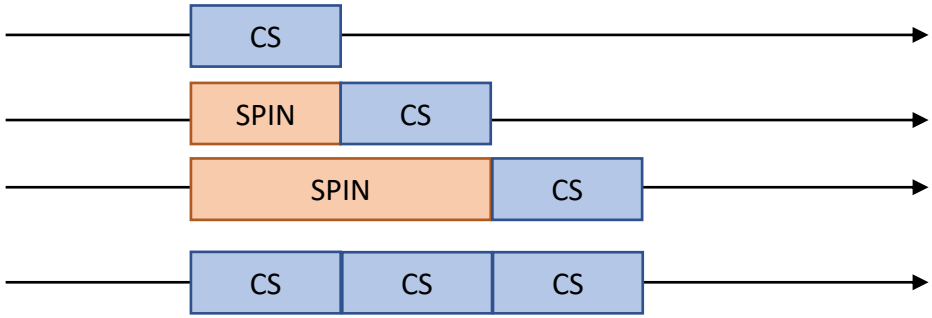


Performance



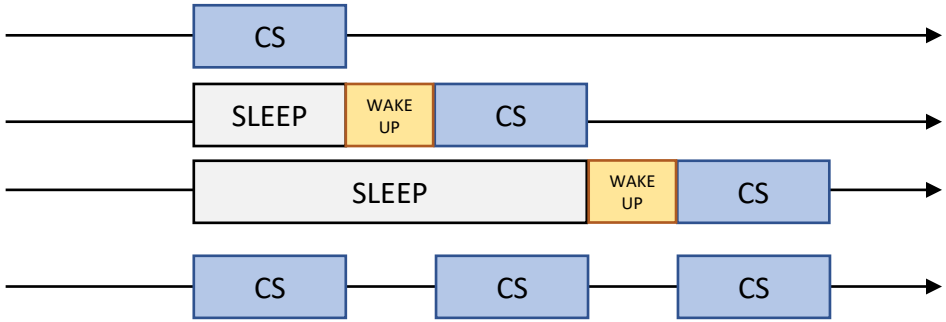
At the beginning was… Spin vs Sleep

	Waiting Policy	
Benefits	**Spinning**	**Sleeping**
Guaranteed low latency	✓	✗
Computing power savings	✗	✓



SPIN:
++Waste of CPU Cycles
--Latency

Sleep:
--Waste of CPU Cycles
++Latency



How to avoid costs for sleeping?

A general approach exists:

- Reducing the frequency of sleep/wake-up pairs
- How?

  ⟶ Trading Fairness in favor of Throughput
- Make some thread sleep longer than others
- If the lock is highly contented, some thread willing to access the critical section will arrive soon
- If the lock is scarcely contented, we pay lower latency as TTAS locks

An example - MutexEE

- MutexEE is a pthread_mutex optimized for throughput and energy efficiency

lock()

MUTEX	MUTEXEE
For up to 100 attempts	
spin with pause	
if still busy, sleep	

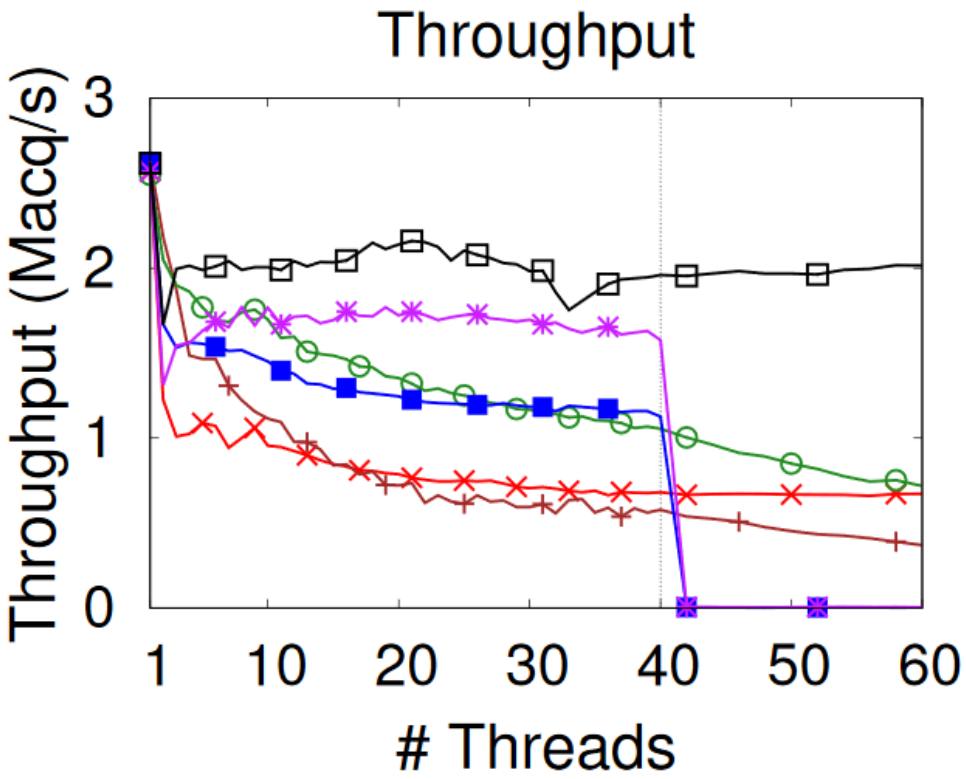
unlock()

MUTEX	MUTEXEE
release in user space (lock->locked = 0)	
wake up a thread	

Credits: Falsafi et all. “Unlocking energy”

An example - MutexEE

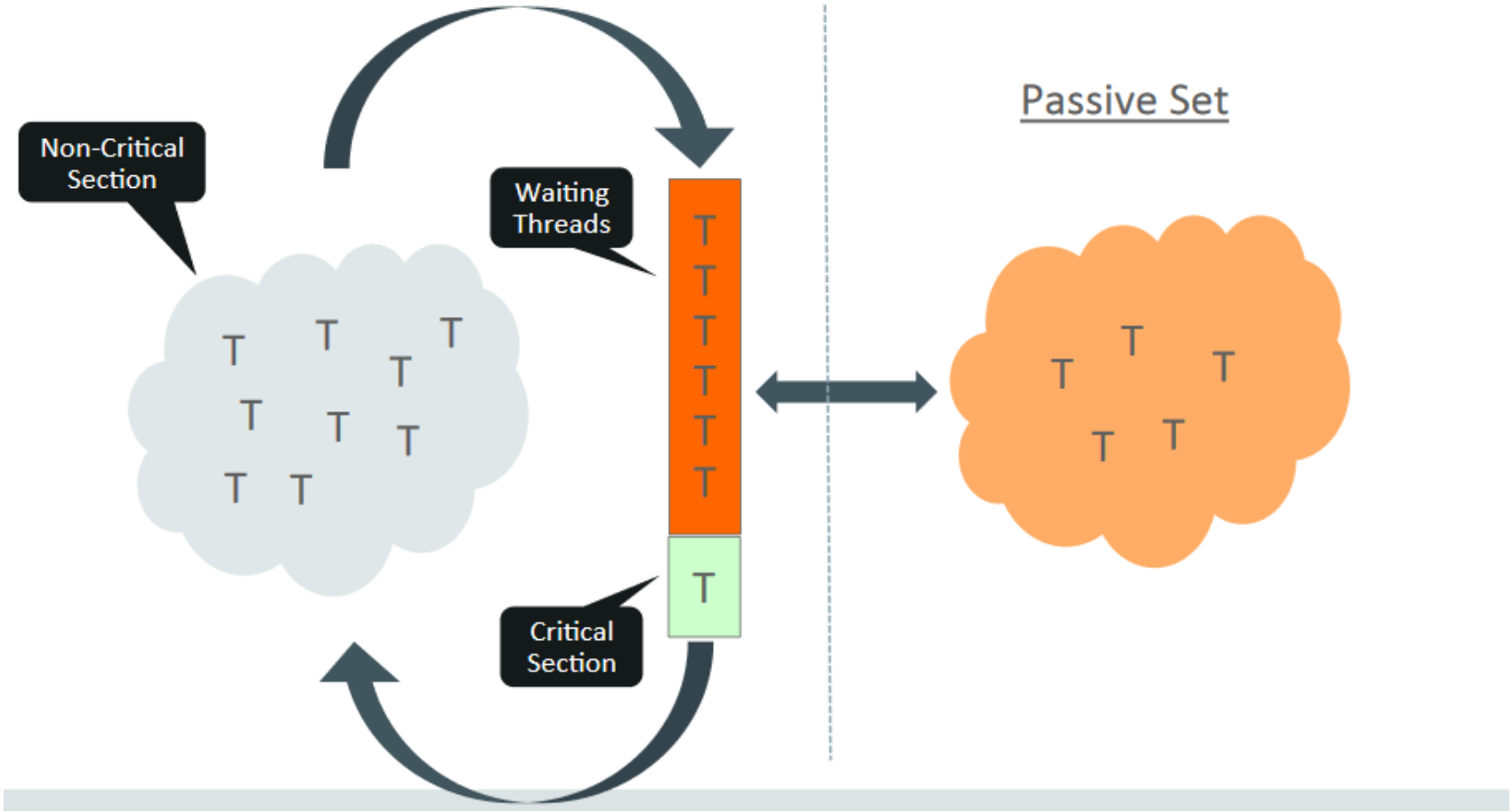
- MutexEE is a pthread_mutex optimized for throughput and energy efficiency



- Global lock
- 1000 cycles CS
- 40 cores

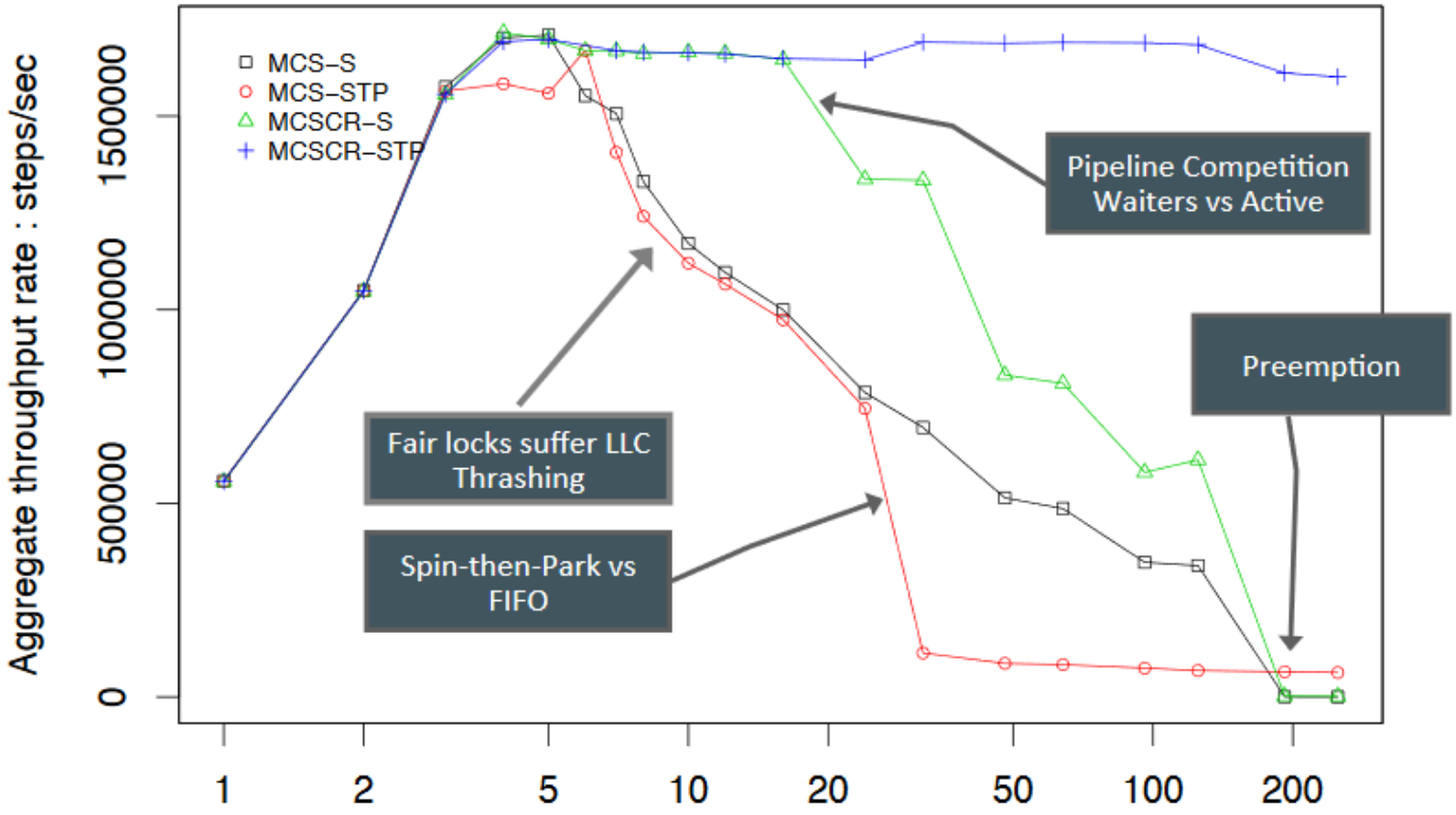
Credits: Falsafi et all. “Unlocking energy”

An example 2 – Malthusian locks



Credits: Dave Dice “Malthusian locks”

An example 2 – Malthusian locks



Credits: Dave Dice “Malthusian locks”

Hierarchical locks

HPC wants maximum usage of CPU power

- Sleeping might be required for better management of I/O
- Large number of cores per machine

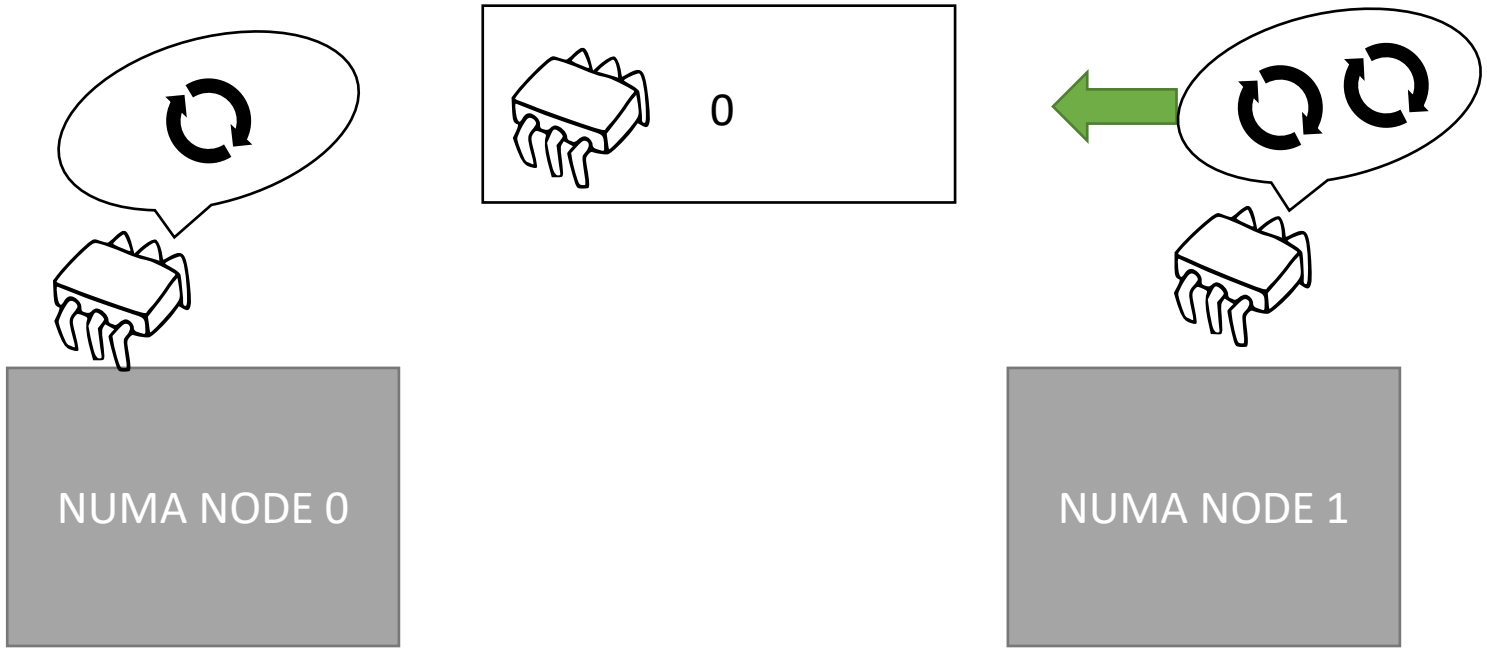
$\Rightarrow$NUMA (again)

- FIFO locks cannot avoid transfer to remote NUMA nodes

Again, we can trade fairness in favor of throughput

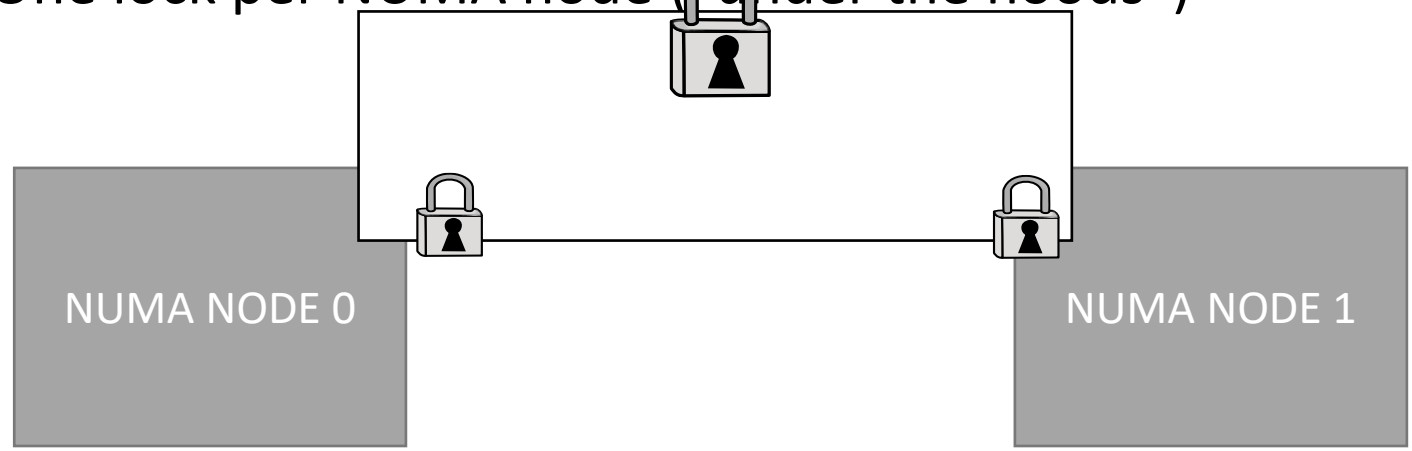
Hierarchical locks

- Transfer the lock to threads that reside on the same NUMA node
- Hierarchical TTAS
  - Shorter backoff for local threads, longer for remote ones



Hierarchical locks

- Transfer the lock to threads that reside on the same NUMA node
- Hierarchical TTAS
  - Shorter backoff for local threads, longer for remote ones
- Hierarchical QUEUE LOCKS (lock cohorting)
  - One global lock (the application one)
  - One lock per NUMA node (“under the hoods”)



Optimizing Critical Section Execution

Optimizing the waiting phase

We have seen several approaches to optimize the lock acquisition phase:

- Back-off scheme
- Cache-awareness TTAS, FIFO locks
- Non-trivial combinations of both sleep and spin phases

What can we do to improve the execution of threads running the critical section?

- Improve locality and cache usage

How?

Observation:

- A lock (typically) protects data (instead of code)

Idea!

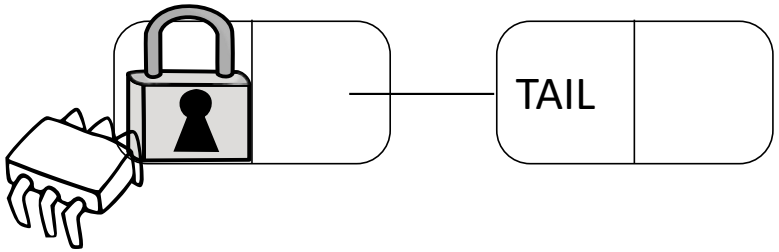
- There is a good chance that threads willing to acquire a lock want to access “similar” sets of data

$\Rightarrow$Allow thread holding the lock to execute the critical section for waiting threads

- Reduces lock handover costs
- Increases locality

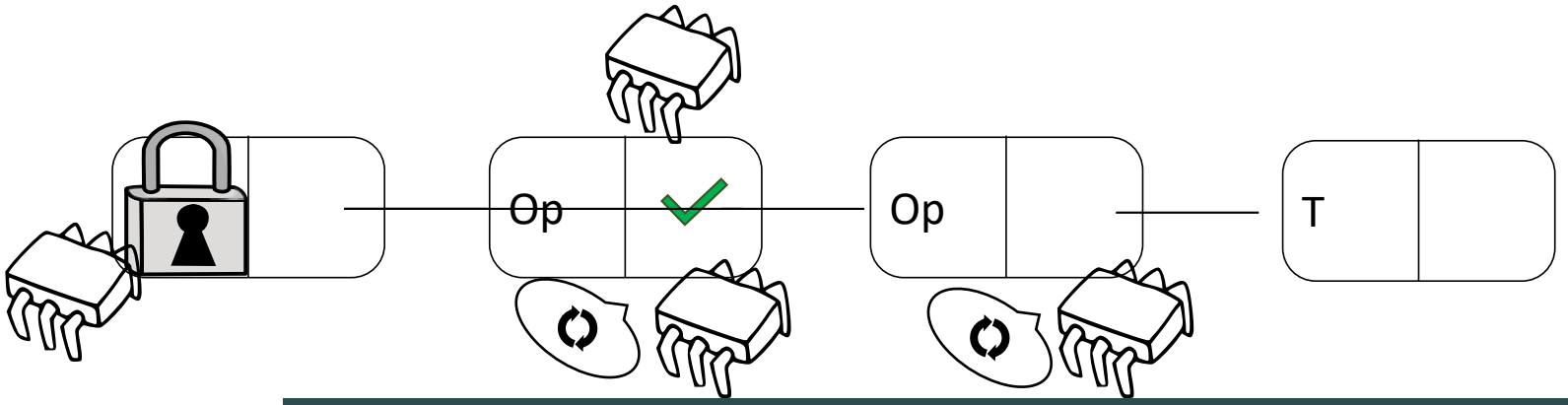
Flat Combining

- Use a linked list for holding waiting threads
- Each node maintains:
  - The waiting thread ID
  - The critical section descriptor
- Thread check waiting queue before releasing the lock
  - If empty exit



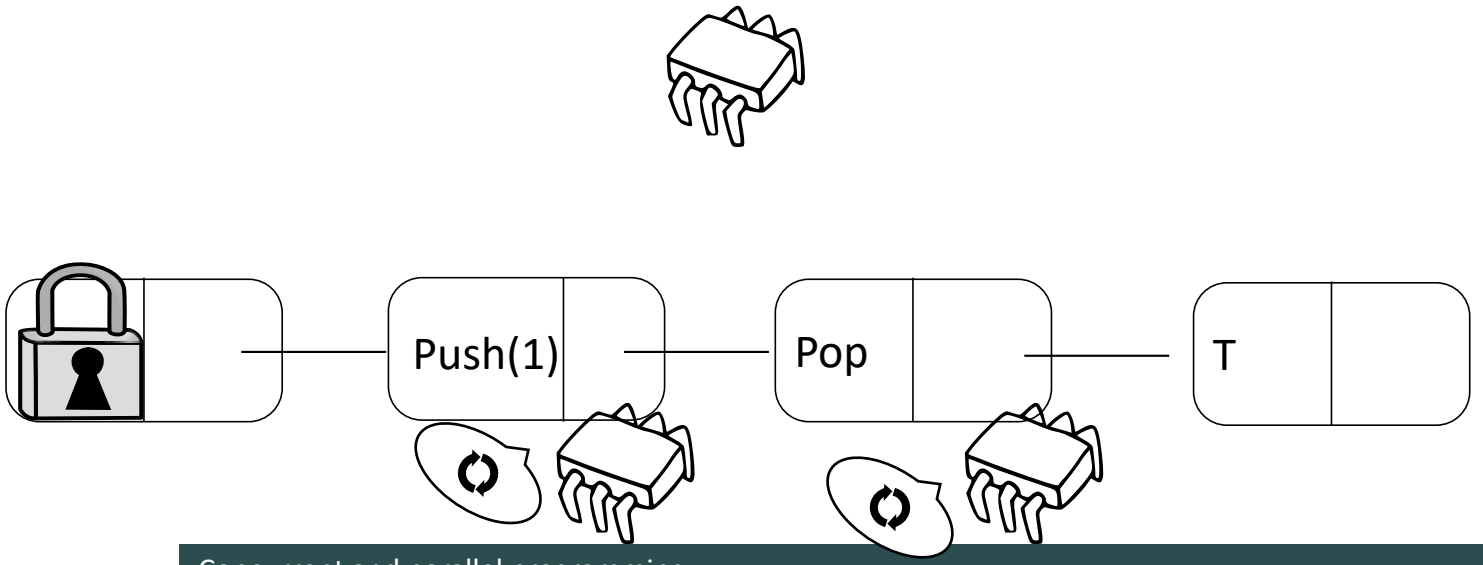
Flat Combining

- Use a linked list for holding waiting threads
- Each node maintains:
  - The waiting thread ID
  - The critical section descriptor
- Thread check waiting queue before releasing the lock
  - If empty exit
  - Otherwise take a node from the waiting queue and execute the critical section for the waiting thread



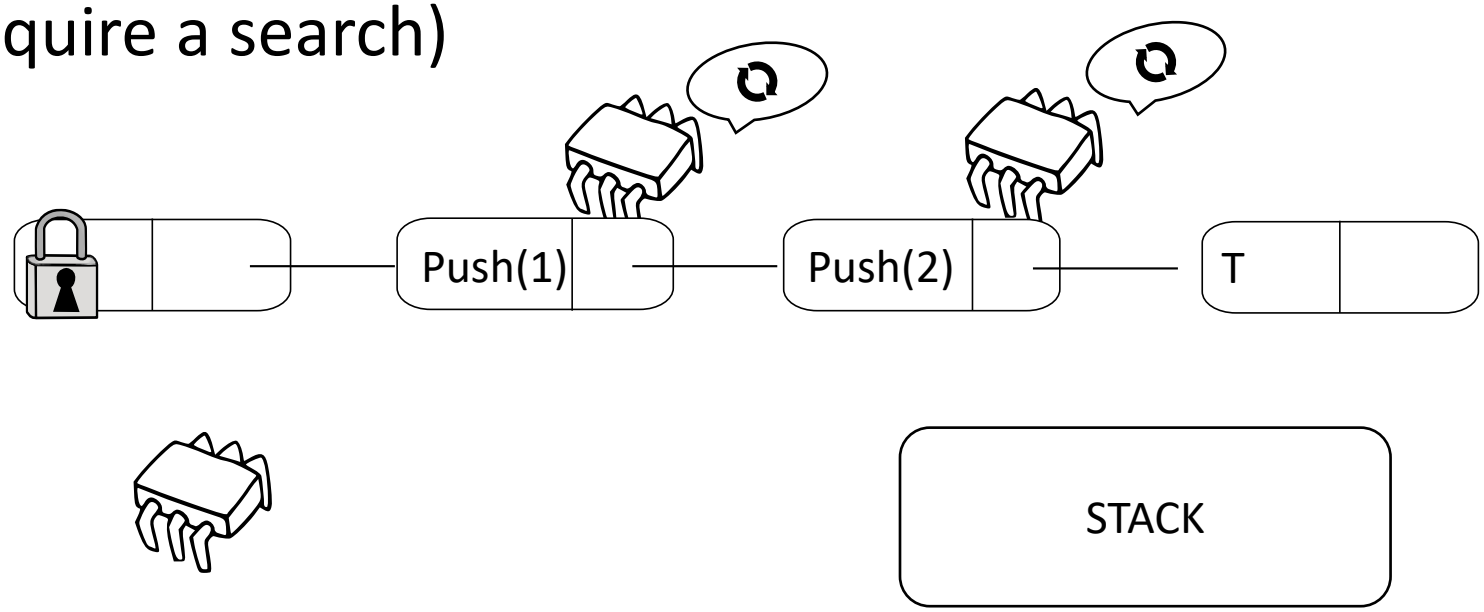
Flat Combining

- It might allow further (asymptotic) optimizations (e.g., data structures)
- Operations can be combined to each other BEFORE interacting with protected data

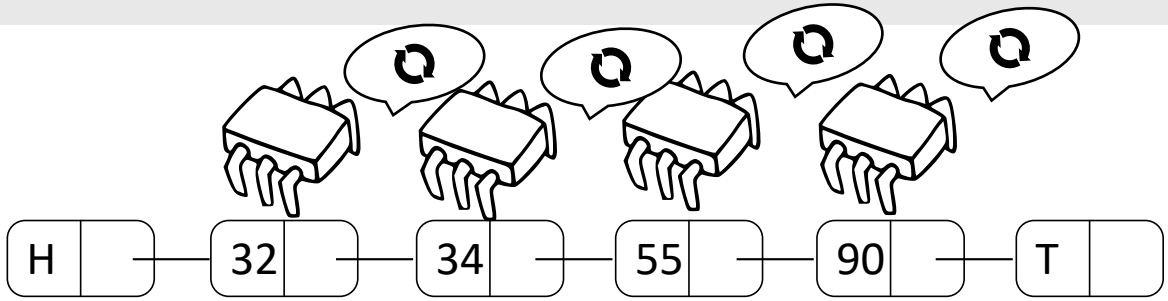


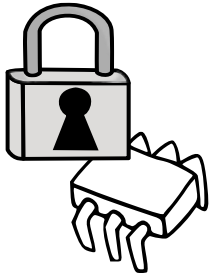
Flat Combining

- It might allow further (asymptotic) optimizations (e.g., data structures)
- Operations can be combined to each other BEFORE interacting with protected data
- Operations can be applied in batch (relevant for accesses that require a search)

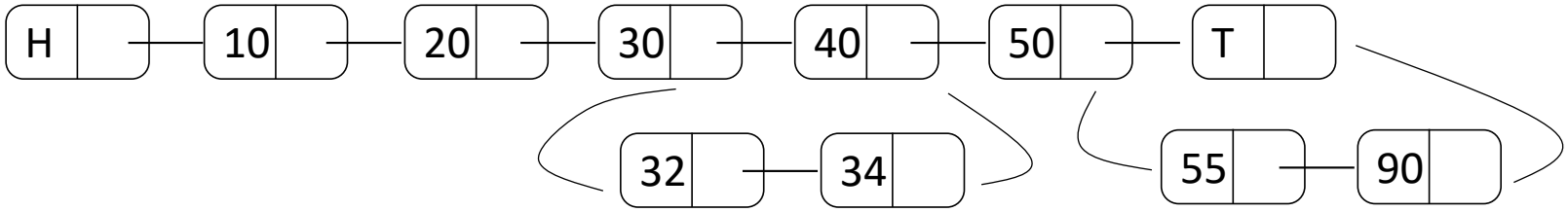


Flat Combining





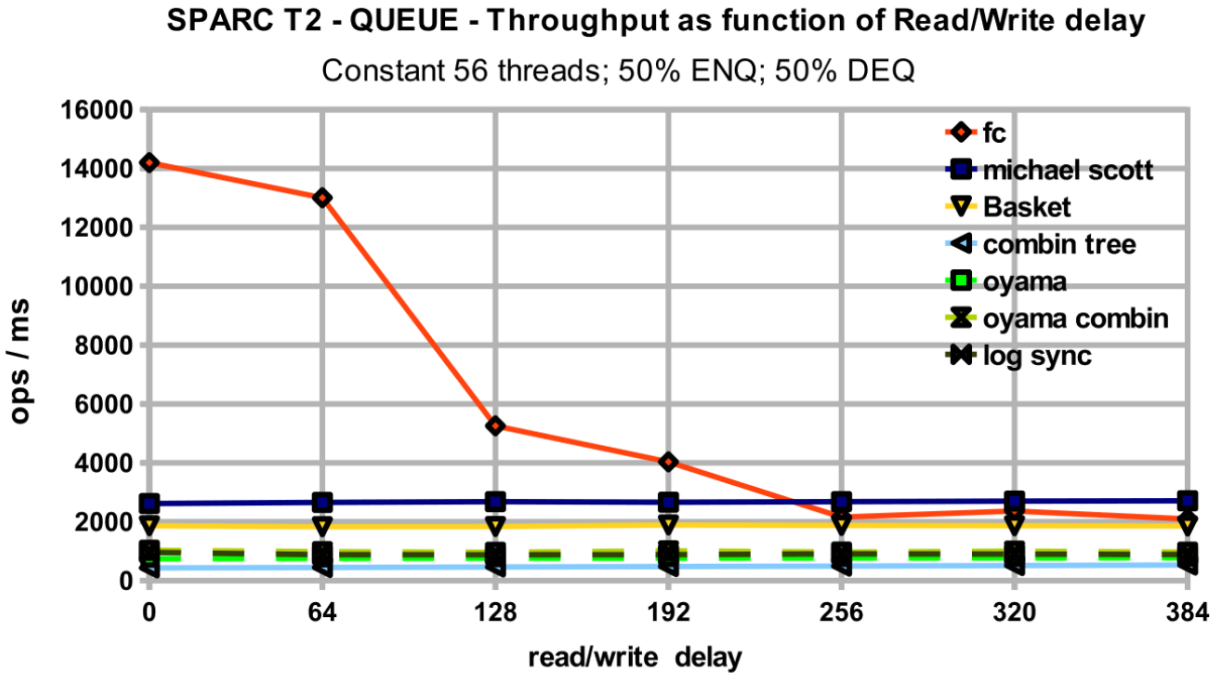
No need for restarting the search from scratch!



Flat Combining

Is it a silver bullet? Can replace complex lock-free algorithms?

NO!



HIGH CONTENTION LOW

Credits: Hendler et all. “Flat combining and the synchronization-parallelism tradeoff”

Flat Combining

Is it a silver bullet? Can replace complex lock-free algorithms?

- No, performance depends on the actual contention!
- Combining requires hand-written code!

How to improve for NUMA?

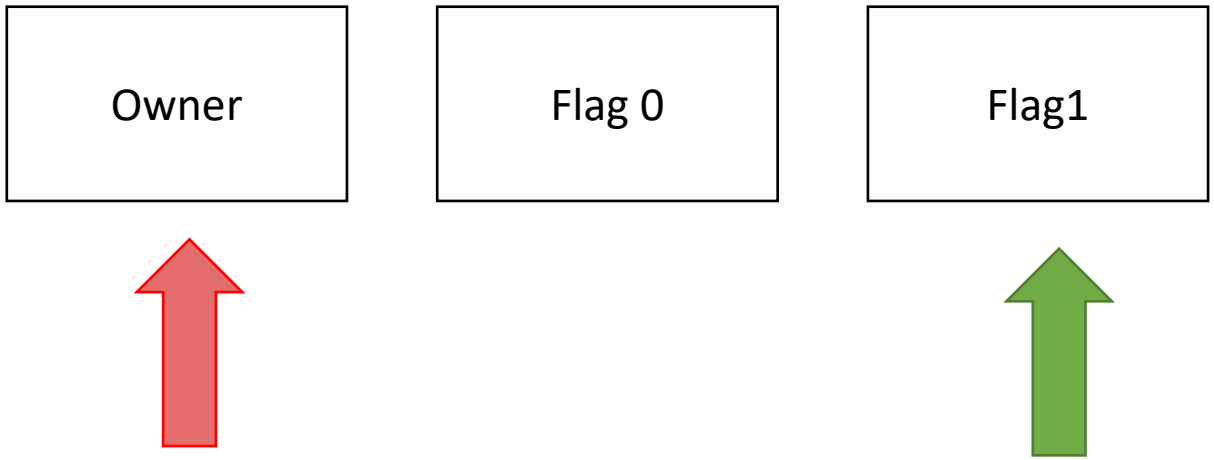
- Hierarchical Flat Combining

Approaches targeting peculiar workloads

- Read-Write locks
  - Threads that do not want to perform updates can acquire the lock with other “readers”
  - Threads willing to perform updates (“writers”) take exclusive lock
- Easy to implement:
  - Lock < 0 : acquired by a writer
  - Lock = 0 : available
  - Lock = N > 0 : locked by N readers
- RW locks work well in read-mostly workloads, but:
  - It has a greater impact to readers (exclusive accesses to the lock variable)
  - Can be optimized by splitting the read counter

RW locks

Multiple RW locks (each one has its own cache line)

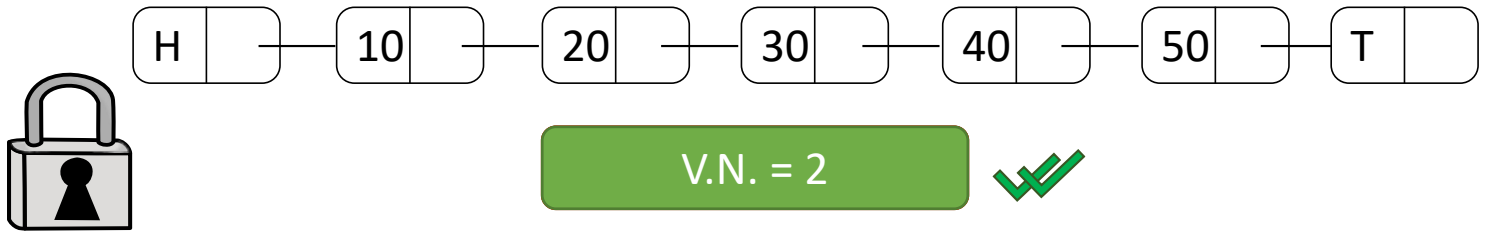


Writer acquire Owner and spin until all flags are 0

If Owner is free
Readers acquire their assigned Flag (e.g. the one of their numa node)
Then, check again Owner

Approaches targeting peculiar workloads (2)

- When read-only accesses are predominant, we can make reader DO NOT use any lock
- Version Numbers
- Writer:
  - Acquire a (writer) lock
  - Increase Version Number
  - Apply Update
  - Increase Version Number
  - Release Lock
- Reader:
  - Wait even Version Number
  - Do job
  - If Version Number is unchanged OK else retry

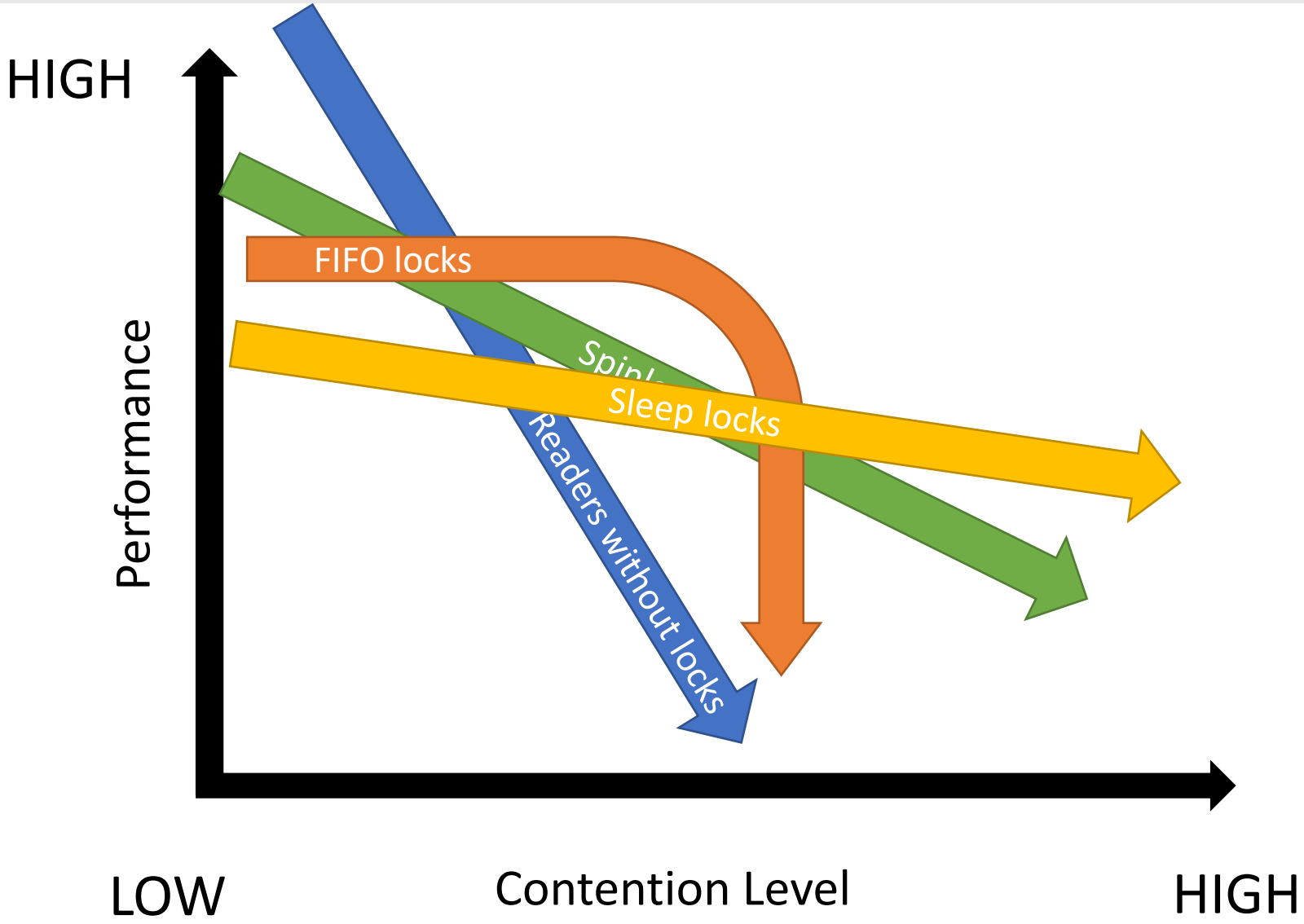


Approaches targeting peculiar workloads (3)

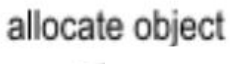
- When read-only accesses are predominant, we can make reader DO NOT use any lock
- Version Numbers
- Read-Copy-Update
  - Single shared-data entry point
- V

These solutions NEEDS memory management as non-blocking algorithms!!!

Final Picture

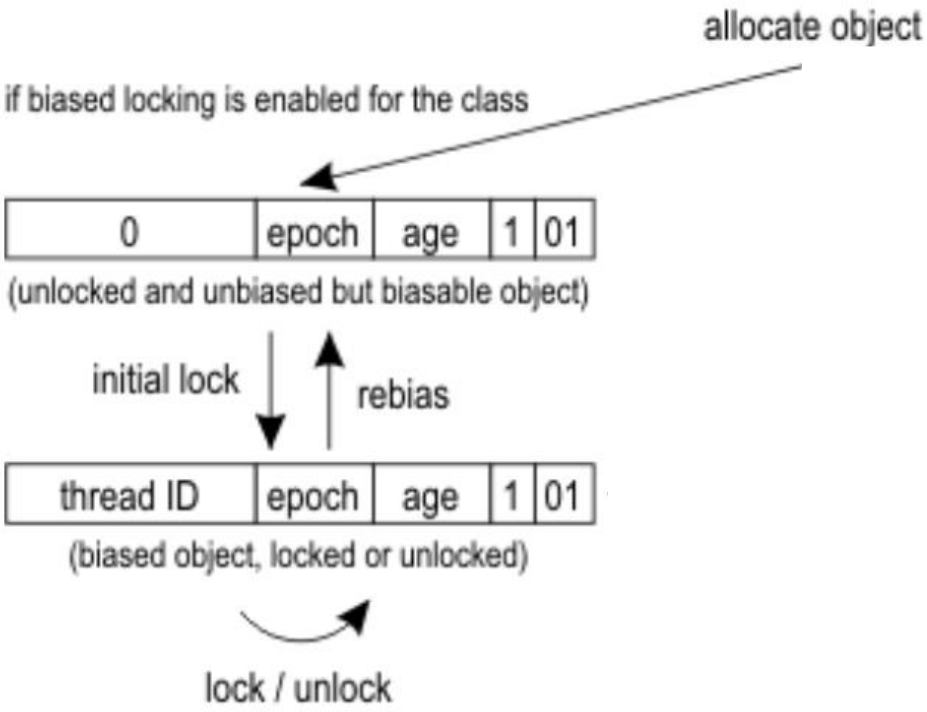


The Java Case



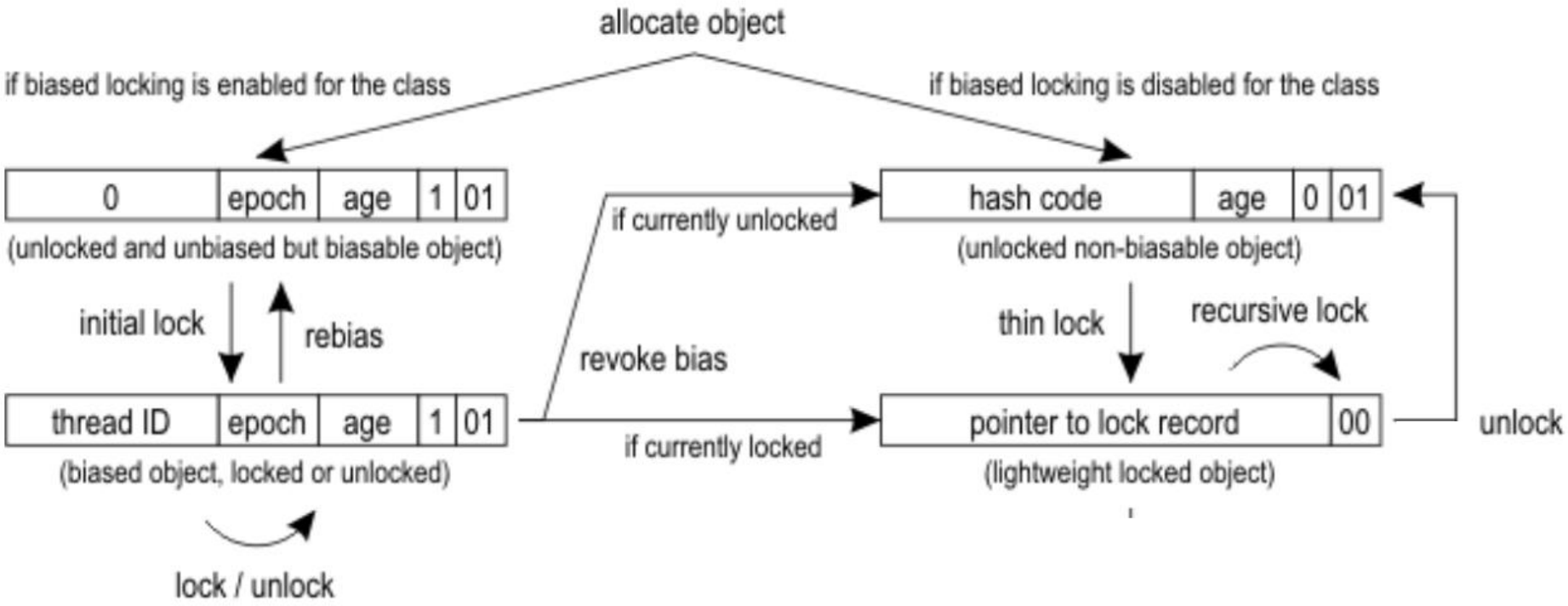
Credits: Kotzmann et all. “Synchronization and Object Locking”

The Java Case



Credits: Kotzmann et all. “Synchronization and Object Locking”

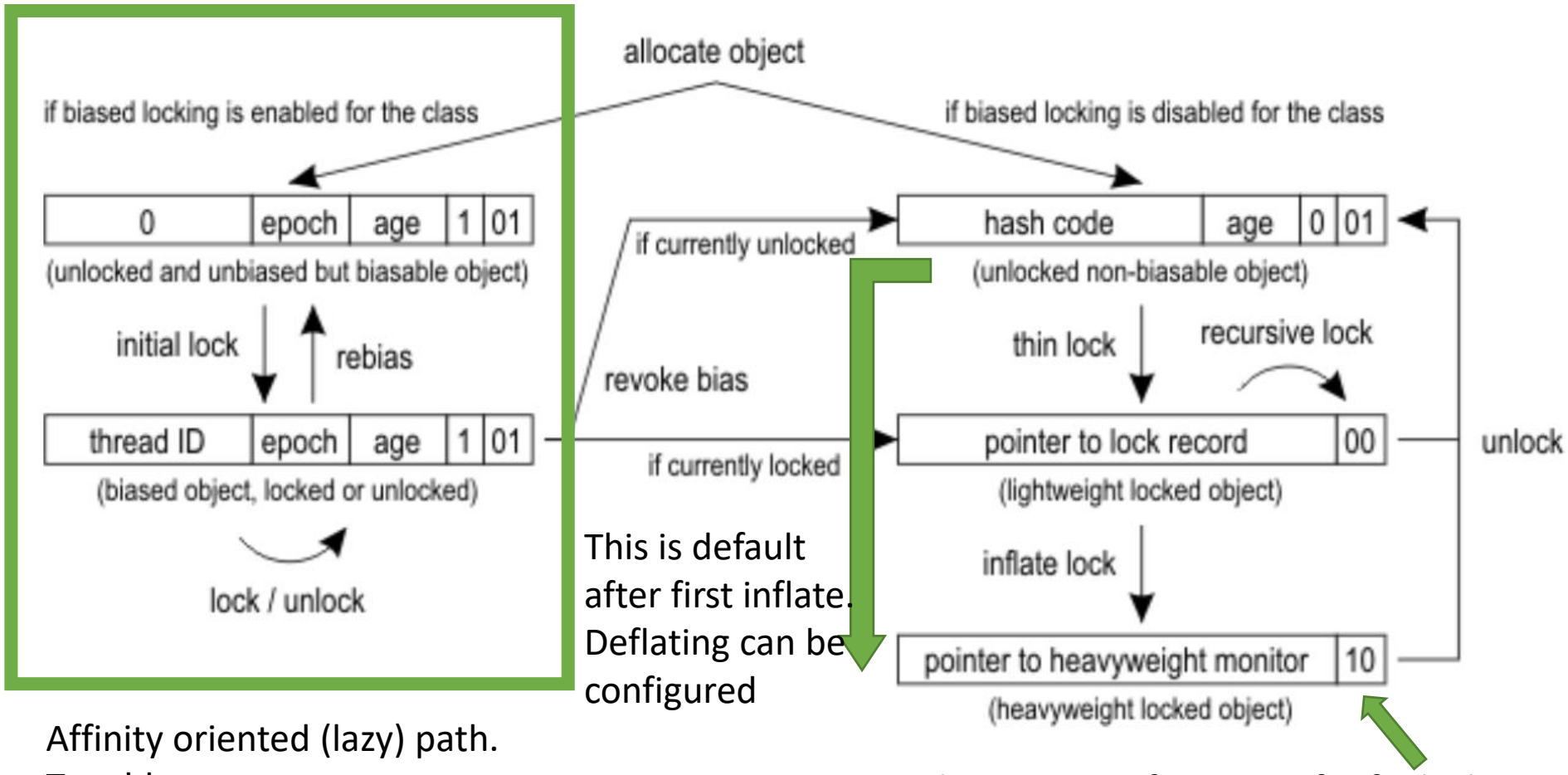
The Java Case



Credits: Kotzmann et all. “Synchronization and Object Locking”

The Java Case

Synchronization and Object Locking: https://wiki.openjdk.java.net/display/HotSpot/Synchronization



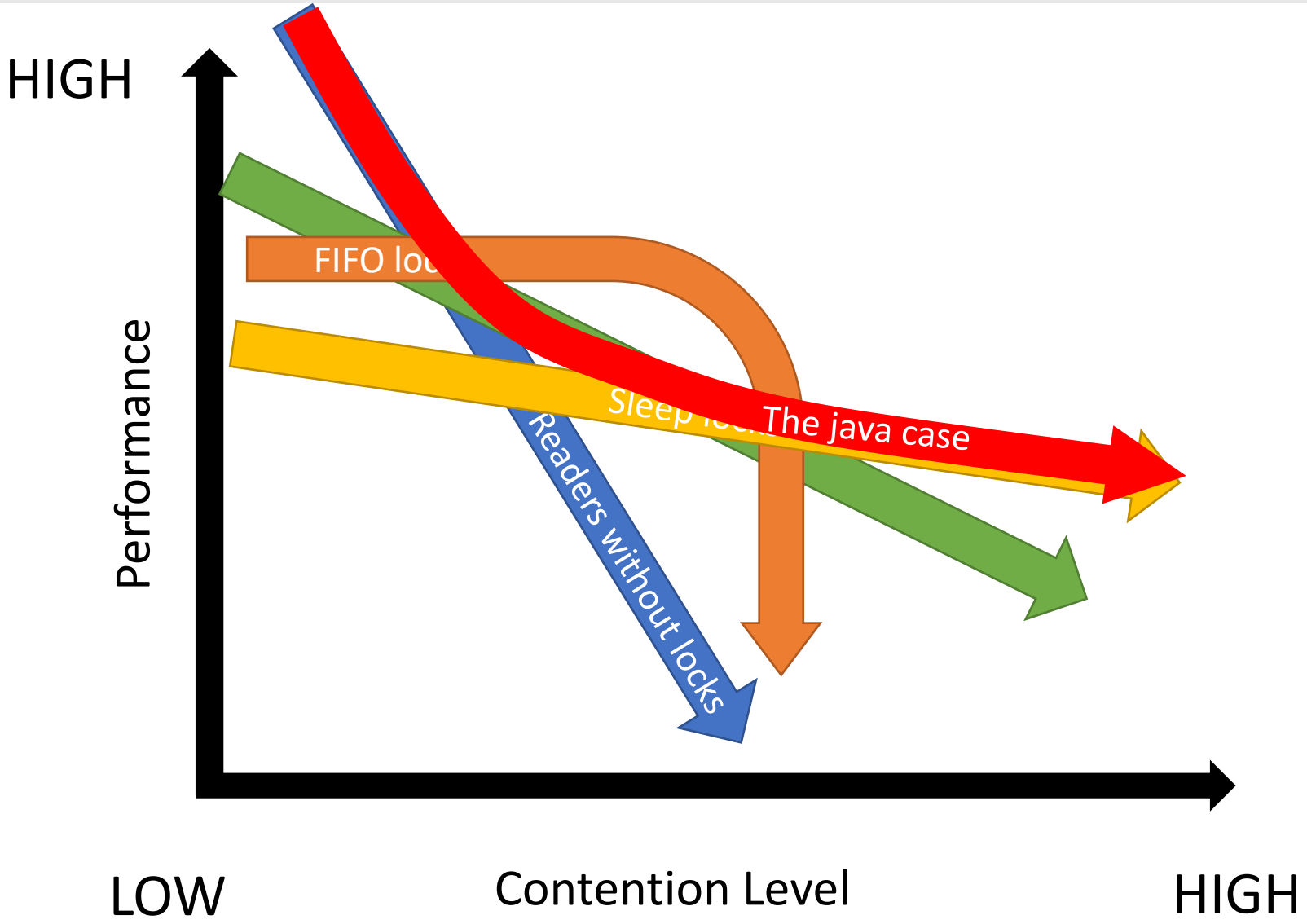
This is default after first inflate. Deflating can be configured

Affinity oriented (lazy) path. Tunable

The amount of spinning for fat locks can be configured (no spin vs adaptive)

Credits: Kotzmann et all. "Synchronization and Object Locking"

Final Picture



Synchronization approaches:

- Non-blocking data structures
- Locks
- Transactional Memory

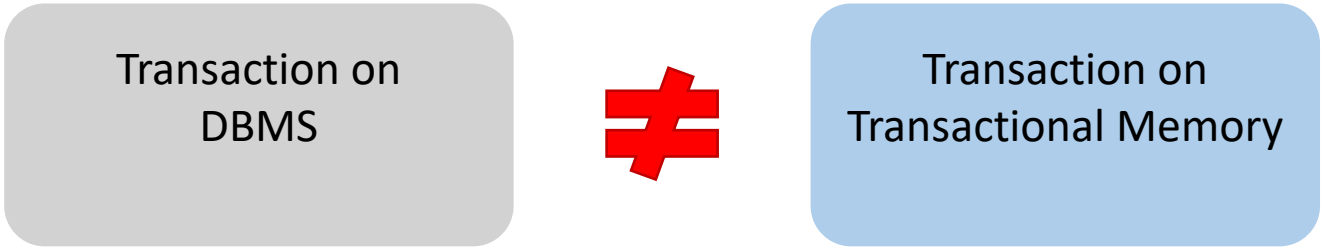
Transactional Memory

- Why?
  - Fine grain locking (or non-blocking synchronization) can scale but it is hard
  - Locks do not scale in general, but they are hard too:
    - Deadlocks
    - Races (forgotten locks)
    - Do not compose
- Transactions:
  - They compose (e.g. nested transactions)
  - Simpler to reason about

```
Begin_transaction
    x.op()
    y.op2(k)
    z.op(j)
End_transaction
```

Transactions

- Well known in the context of databases
- Conceived integration of transaction in hardware (1993)
- Software implementations (1995-2005)
- Commercial hardware support (2013)

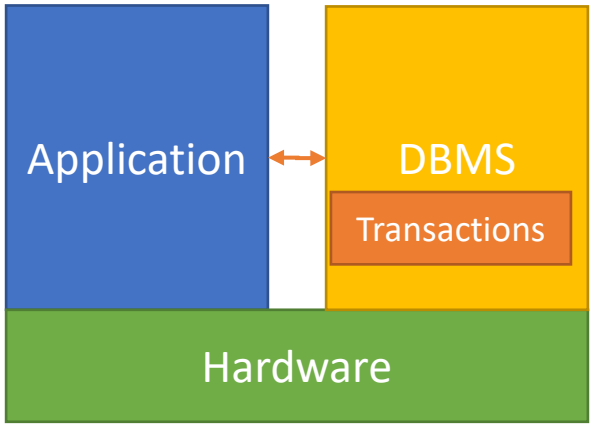


Transactions

Transaction on top of DBMS

≠

Transaction on top of Transactional Memory



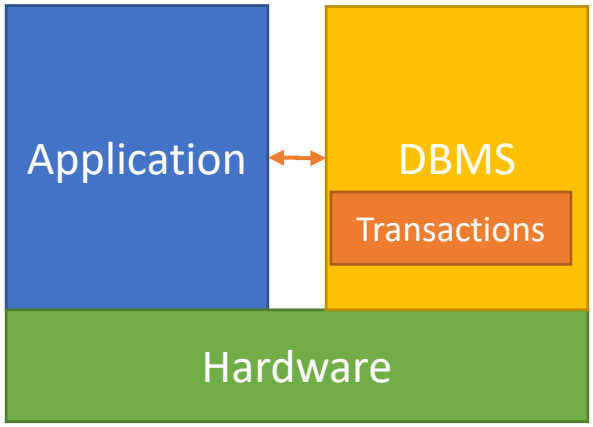

```
x = 2; y = 1
Begin:
    d = x
                                Begin:
                                    y++
                                    x++
                                Commit
    n = y
    write(z, 1/(n-d))
Abort
```

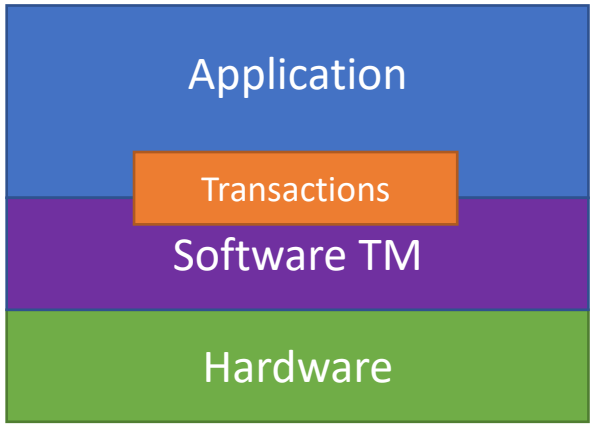
Transactions

Transaction on top of DBMS

≠

Transaction on top of Transactional Memory






```
x = 2; y = 1
B
                    Begin:
                      y++
                      x++
                    Commit
    n = y
    write(z, 1/(n-d))
Abort
```

Managed by DBMS instead of developers

Float Exceptions are not transparent to developers

Transactions

Transaction on top of DBMS

≠

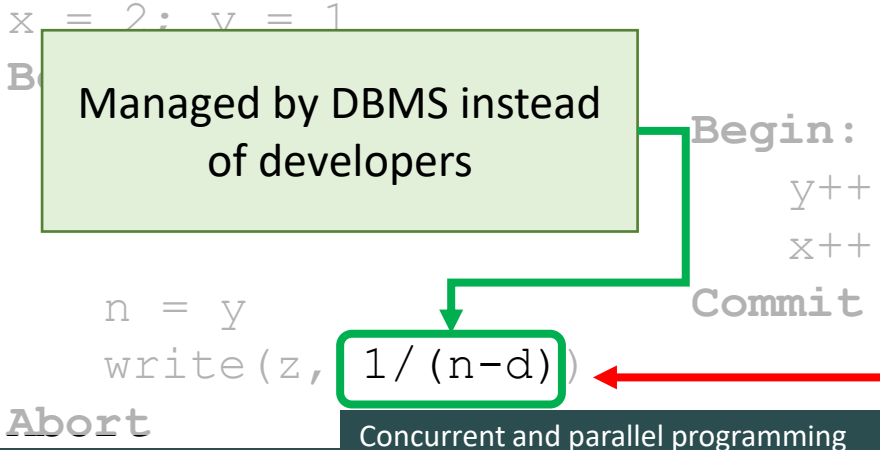
Transaction on top of Transactional Memory

(view) serializability:
Committed transactions see consistent values

Opacity:
Both committed and aborted transactions see consistent values

Managed by DBMS instead of developers

Float Exceptions are not transparent to developers



Transactions

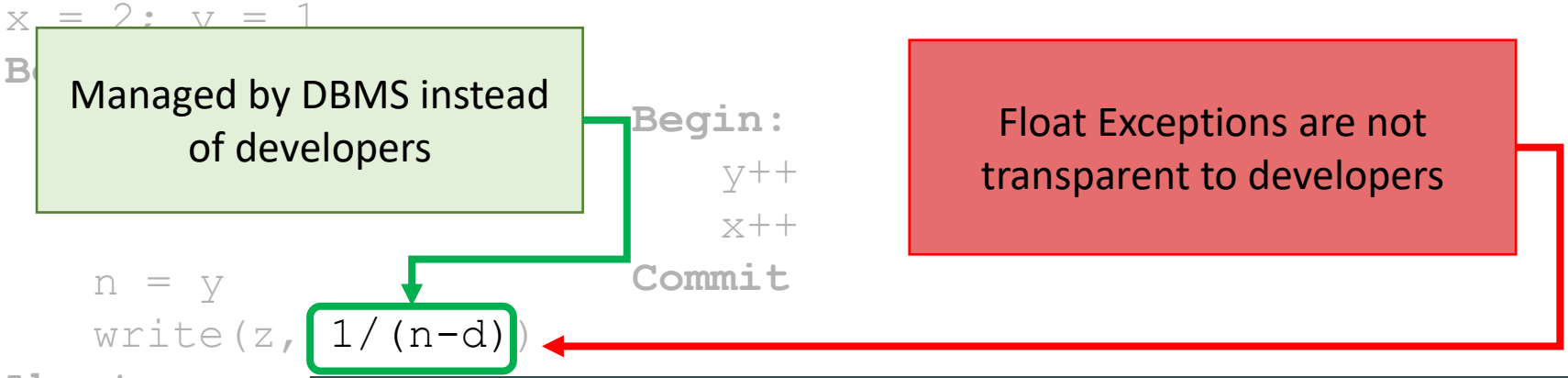
Deadlock or starvation freedom

≠

Obstruction of lock freedom

(view) serializability:
Committed transactions see consistent values

Opacity:
Both committed and aborted transactions see consistent values

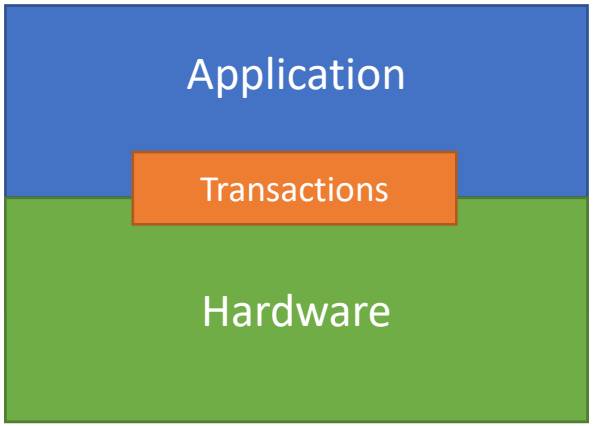


Managed by DBMS instead of developers

Float Exceptions are not transparent to developers

Hardware Transactional Memory

Intel TSX
BlueGene
RockProcessor
Arm Transactional Extension
IBM POWER8 and 9



Memory:

- Exploit cache coherency protocols
- Modified
- Exclusive
- Shared
- Invalid
- Tracked for speculative execution of transaction
- Losing track of a cache line leads to an abort

CPU:

- Ability to restore the processor state as the one before the beginning

Hardware transaction and abort

- Why can a hardware transaction abort?
  - Whenever, we lose track of a cache line….
- Any reason that could lead to an invalidation of a tracked cache line:
  - Another core wants it exclusive (conflict)
  - Change of execution mode (syscall, interrupts, page fault)
  - Working set too large

Intel Transactional Synchronization eXtensions (TSX)

RTE

- XBEGIN:
  - Start a hardware transaction (keep track of accessed cache lines)
- XEND:
  - Try to commit a hardware transaction (untrack cache lines)
- XABORT:
  - Make a hardware transaction abort programmatically

Are HTM so simple?

```
int committed_count;
void transaction(){
char *buf = malloc(4096*1024); // 4MB
start_tsx:
    if(_XBEGIN() == _XBEGIN_STARTED){
        committed_count++;
        do_job(buf,...)
        _XEND();
        return;
    }
    else goto start_tsx;
}
```

All threads try to updated the same cache line!!!

Huge memory init!!!

This is not a good fallback path!

Are HTM so simple?

```
int committed_count;
void transaction(){
char *buf = malloc(4096*1024); // 4MB
start_tsx:
    if(_XBEGIN() == _XBEGIN_STARTED){
        do_job(buf,...)
        _XEND();
        FAD(&committed_count,1);
        return;
    }
    else goto start_tsx;
}
```

Huge memory init!!!

This is not a good fallback path!

Are HTM so simple?

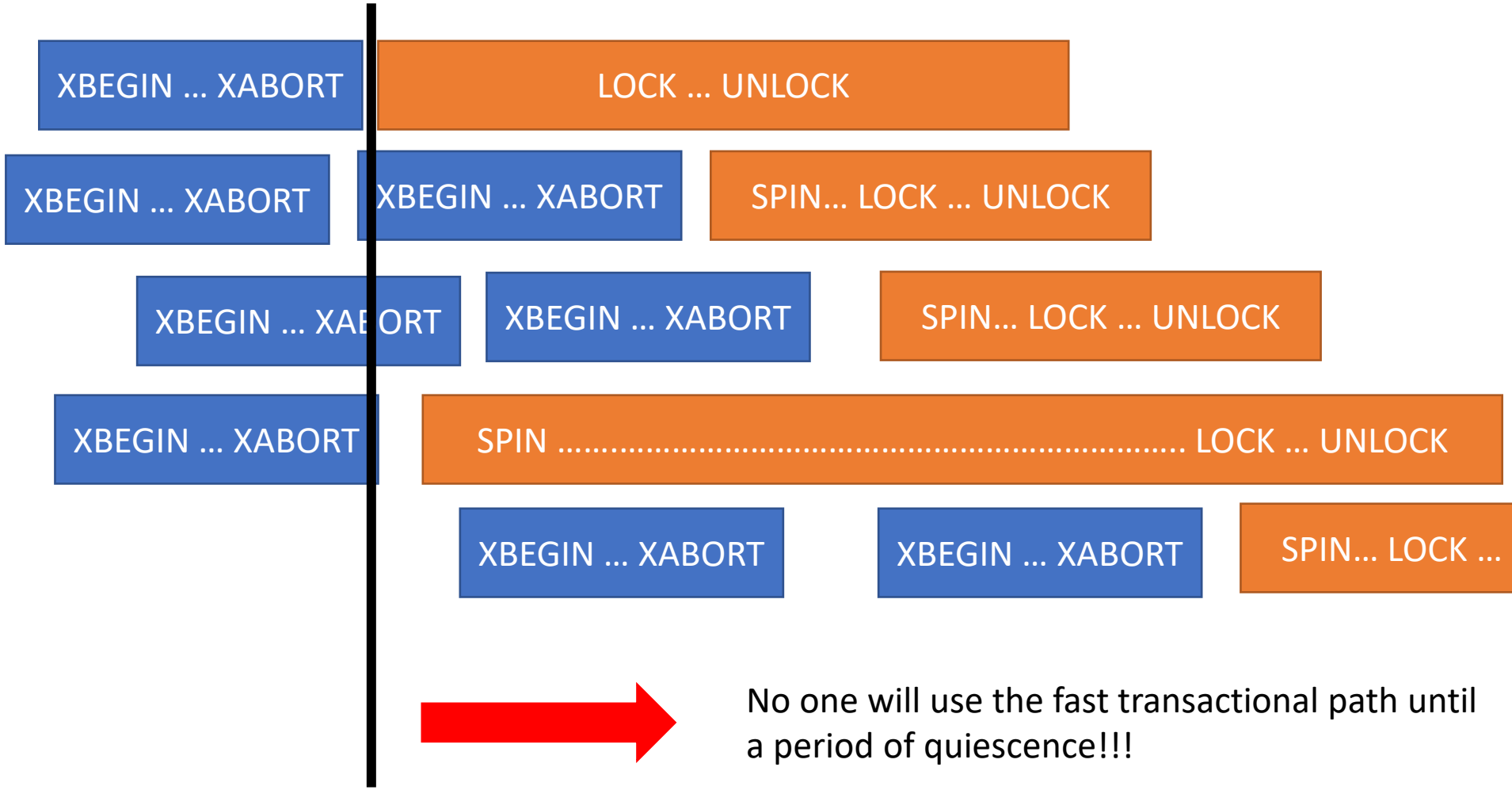
```
int committed_count;
void transaction(){
char *buf = malloc(4096*1024); // 4MB
init(buf);
start_tsx:
      if(_XBEGIN() == _XBEGIN_STARTED){
            do_job(buf,...)
            _XEND();
            FAD(&committed_count,1);
            return;
      }
      else goto start_tsx;
}
```

This is not a good fallback path!

Are HTM so simple? NO!

```
int committed_count; volatile int lock = UNLOCKED;
void transaction(){
char *buf = malloc(4096*1024); // 4MB
init(buf); bool fb = false; int retry =0;
start_tsx:
     if(fb || _XBEGIN() == _XBEGIN_STARTED){
          if(lock==LOCKED) _XABORT();
          if(fb) TTAS(&lock, LOCKED);
          do_job(buf,...)
          if(fb) lock = UNLOCKED;
          _XEND();
          FAD(&committed_count,1);
          return;
     }
     else {fb=++retry>MAX_RETRY; goto start_tsx;}
}
```

Are HTM so simple? NO!



No one will use the fast transactional path until a period of quiescence!!!

Are HTM so simple? NO!

```
int committed_count; volatile int lock = UNLOCKED;
void transaction(){
char *buf = malloc(4096*1024); // 4MB
init(buf); bool fb = false; int retry =0;
start_tsx:
      if(fb || _XBEGIN() == _XBEGIN_STARTED){
            if(lock==LOCKED){while(lock==LOCKED);
                  _XABORT();}
            if(fb) TTAS(&lock, LOCKED);
            do_job(buf,...)
            if(fb) lock = UNLOCKED;
            _XEND();
            FAD(&committed_count,1);
            return;
      }
      else {fb=++retry<MAX_RETRY; goto start_tsx;}
}
```

Are HTM so simple? NO!

- We cannot replace lock with HTM as is due to performance aspects
- Naïve code might abort frequently due to:
  - Statistics
  - Memory allocations
  - Fallback path policy make the fast past rarely used
  - False cache-sharing
  - NUMA
  - NVRAM

Intel Transactional Synchronization eXtensions (TSX)

RTE

- XBEGIN:
  - Start a hardware transaction (keep track of accessed cache lines)
- XEND:
  - Try to commit a hardware transaction (untrack cache lines)
- XABORT:
  - Make a hardware transaction abort programmatically
- Needs a fallback path (e.g., by using locks)

HLE

- XACQUIRE:
  - Start a hardware transaction
  - execute a RMW without the LOCK prefix (XACQUIRE LOCK XCHG mutex, 1)
- XRELEASE:
  - Execute a mov to release the lock (XRELEASE mov mutex, 0)
  - Try to commit
- No need for an additional fallback path (just drop xacquire/xrelease and restart)

Is it worth investing in optimizing our code for HTM?

- VERY HARD TO SAY

HTM has been around for a while (2014), BUT:

- IBM BlueGene/Q It is a high-end processor, not an off-the-shelf
- RockProcessor Canceled in 2009
- IBM POWER8 and 9 (Power ISA v.2.07 to 3.0) Not present in 10 (3.1)
- Intel TSX
  - First releases were bugged => disabled by firmware update
  - As other speculative components of Intel processors, they are vulnerable (leak info, see *TSX* Asynchronous Abort (TAA) / CVE-2019-11135) => disabled by firmware update
  - Not supported in last generation Cometlake cpu (finger crossed for the next one)
- Arm Transactional Extension introduced in the last generation Armv9 (Mar 30 2021)

What about Software Transactional Memory

From a programmer perspective:

- It is less efficient than hardware implementation
- It generally provides stronger progress
- No need for a fallback path
- Processor independent
- Stick with the support of the community/organization developing it

What about Software Transactional Memory

- Hot topic in 2005
- A pletora of implementations for several programming languages
- C/C++
  - TinySTM
  - From G++ v4.7 (still expertimental)
- C#
  - SXM by Microsoft (discontinued)
- Haskell
  - STM is part of the Haskell platform
- Scala
  - Akka framework
- Java, python