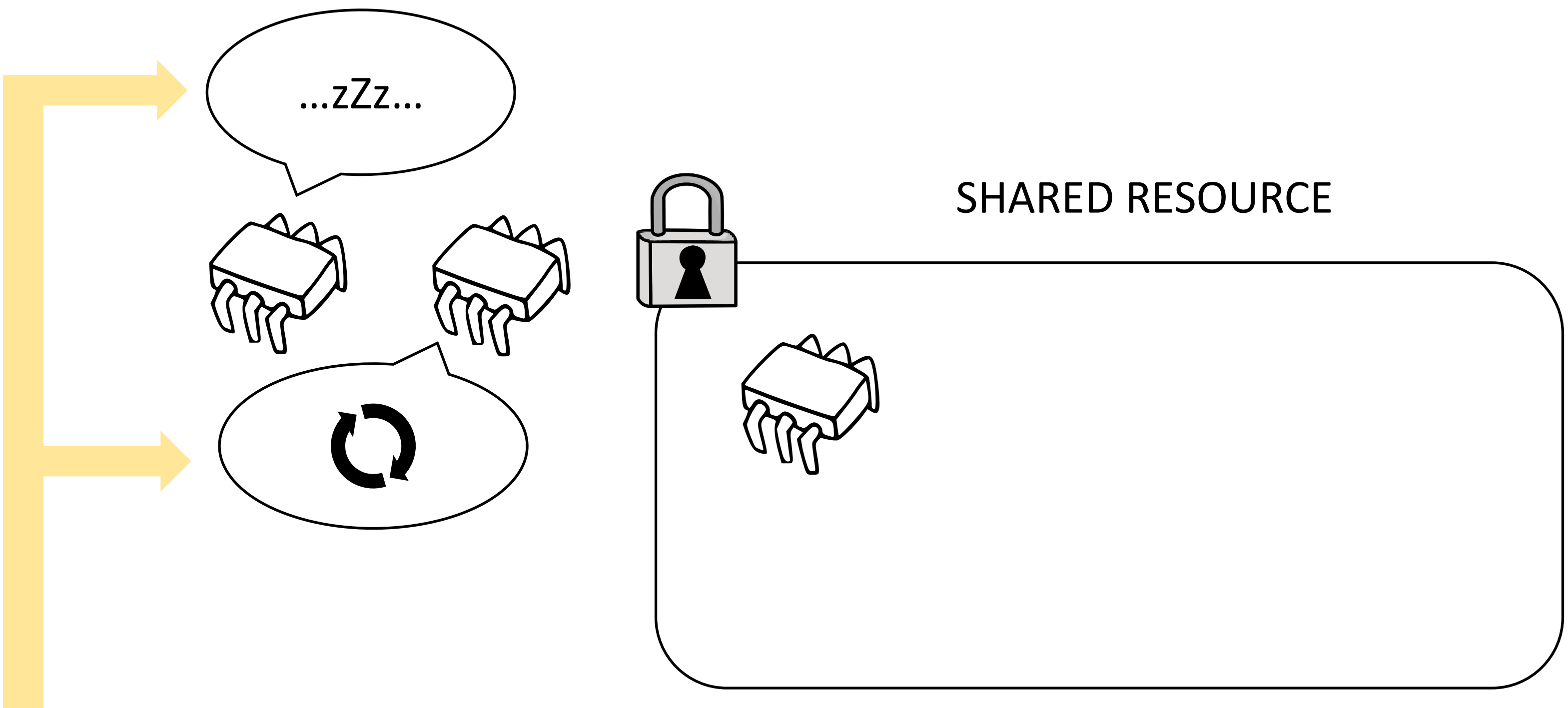
Programmazione concorrente

Laurea Magistrale in Ingegneria Informatica
Università Tor Vergata
Docente: Romolo Marotta

Locks

1. Spin locks
2. FIFO Spin locks
3. Hybrid (Spin+Sleep) locks

Blocking coordination

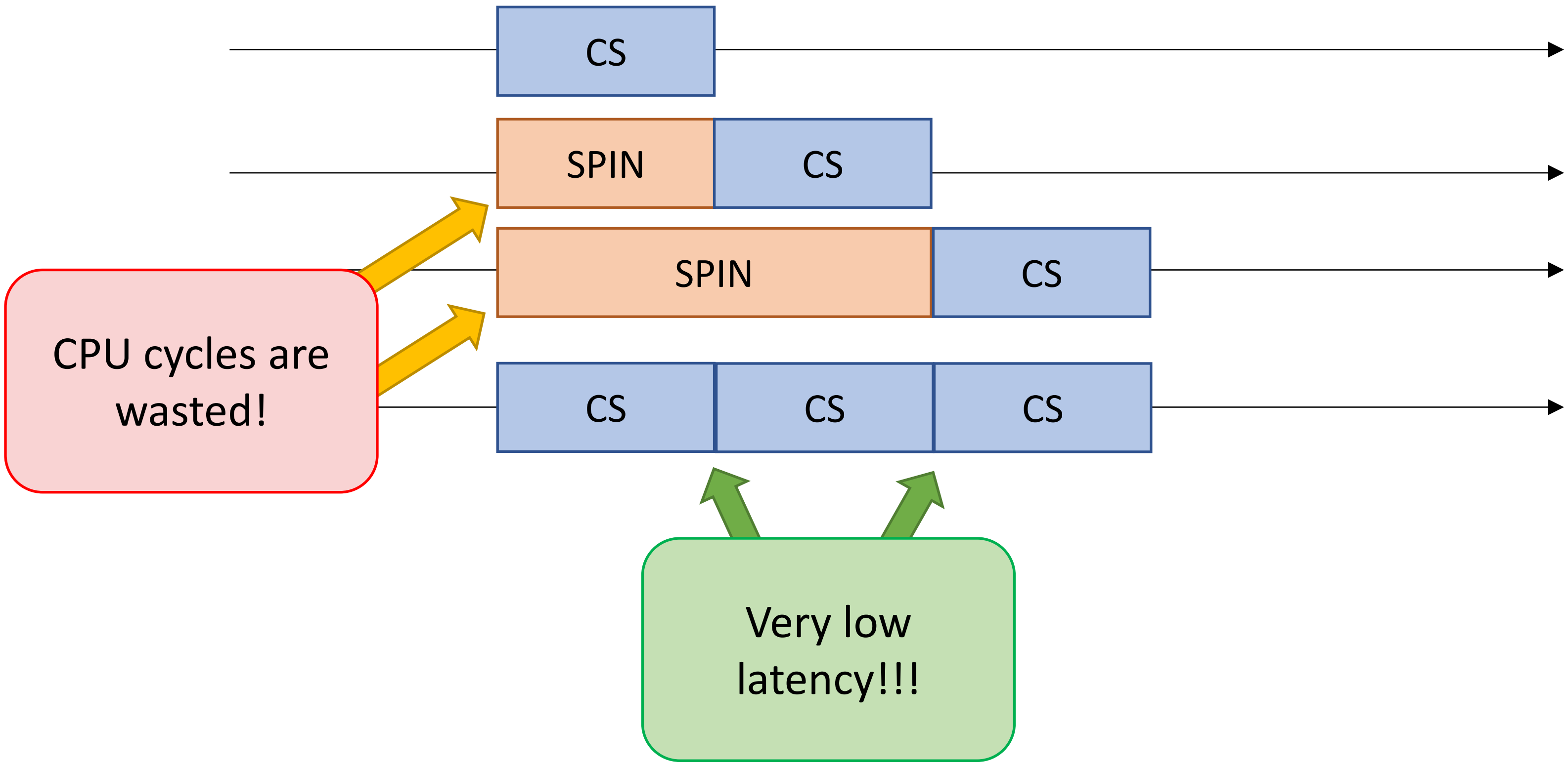


Choosing between them is delegated to developers!

This is a very hard task: multiple trade offs!

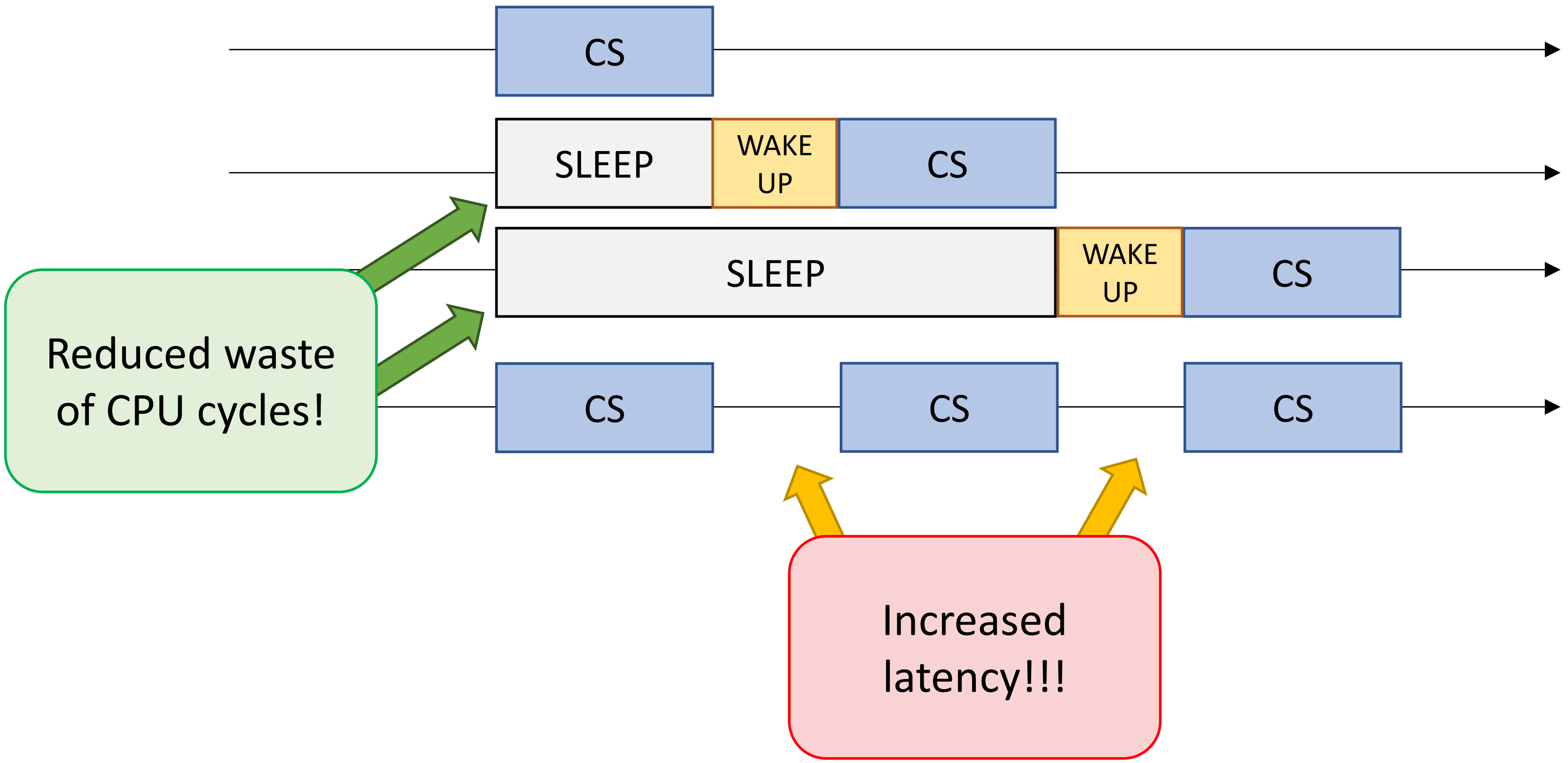
Spinning vs Sleeping

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

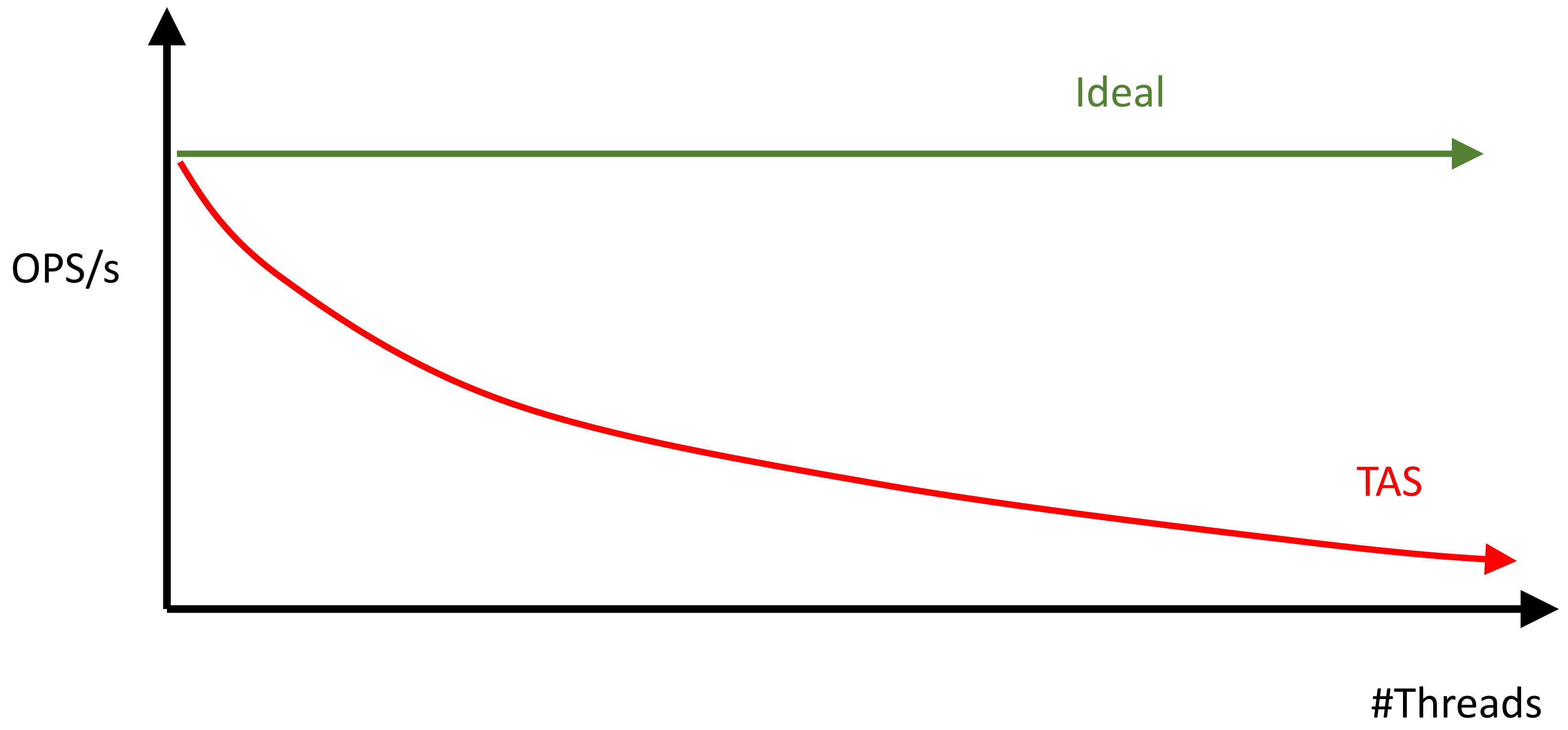
1	2	3	4	5	6	→	6	5	4	3	2	1

- This is done within a critical section

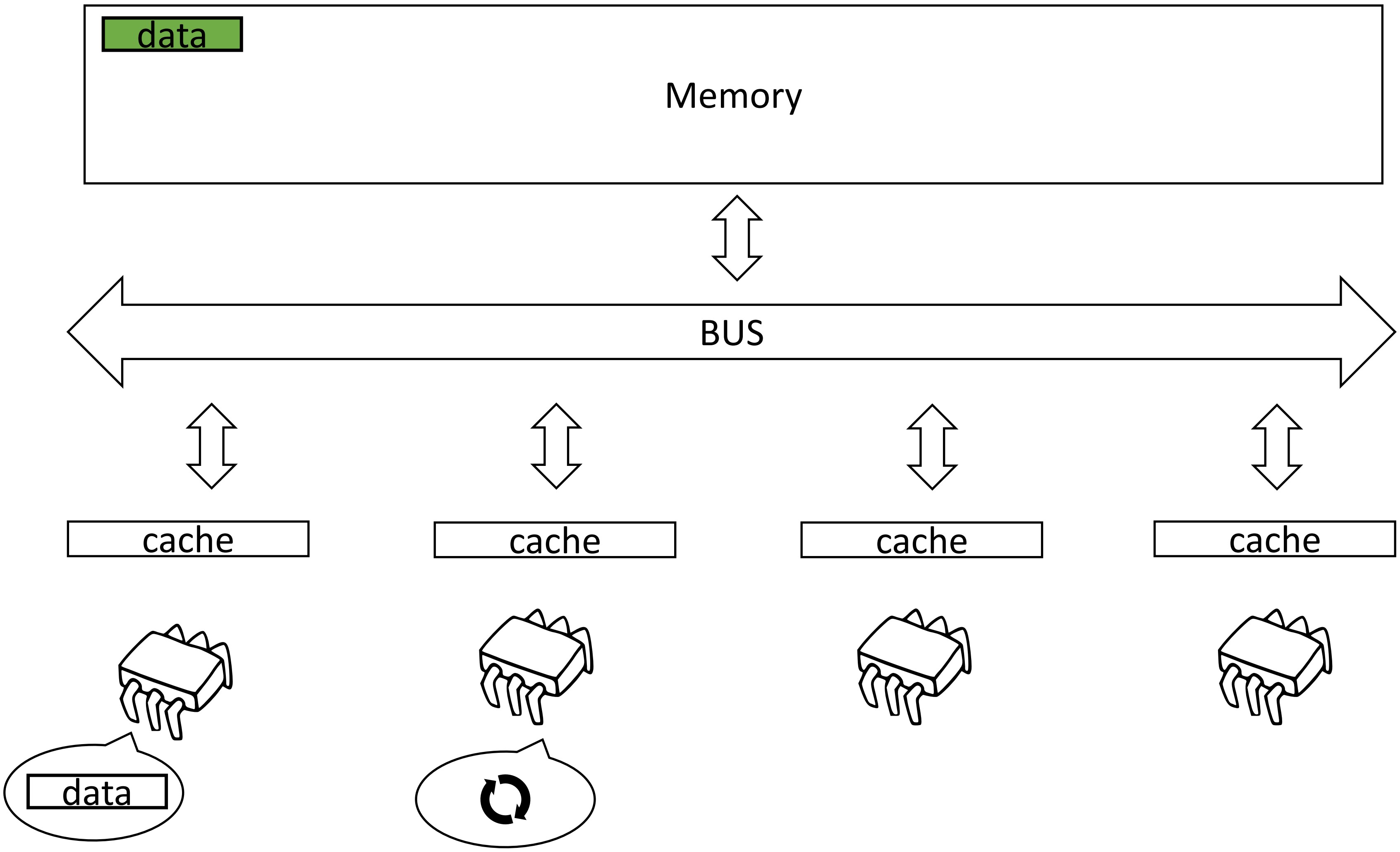
```
while(!stop){
  acquire(&lock);
  flip_array();
  release(&lock);
}
```

- Performance Metric:
  - Throughput = #Flips per second

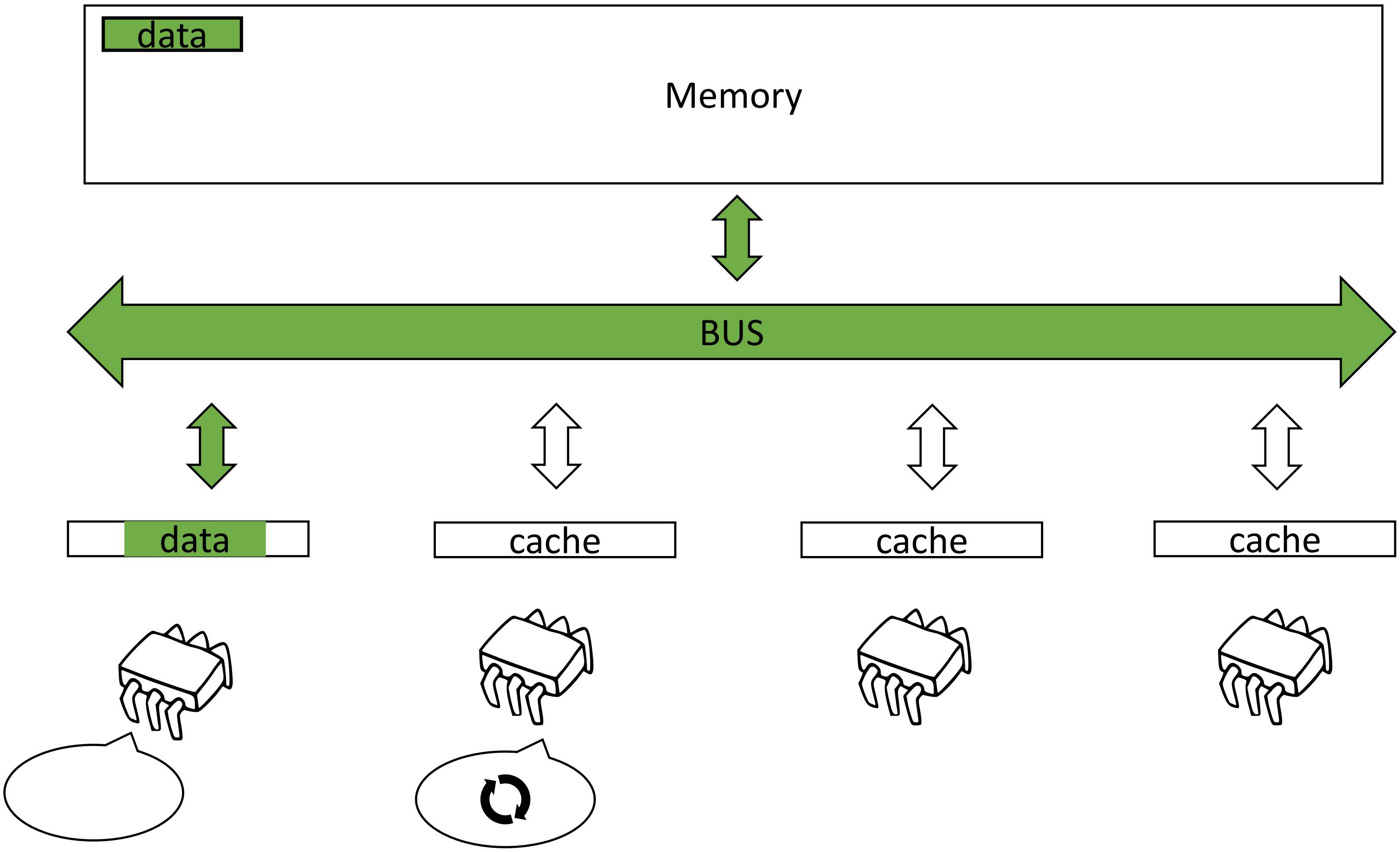
Results



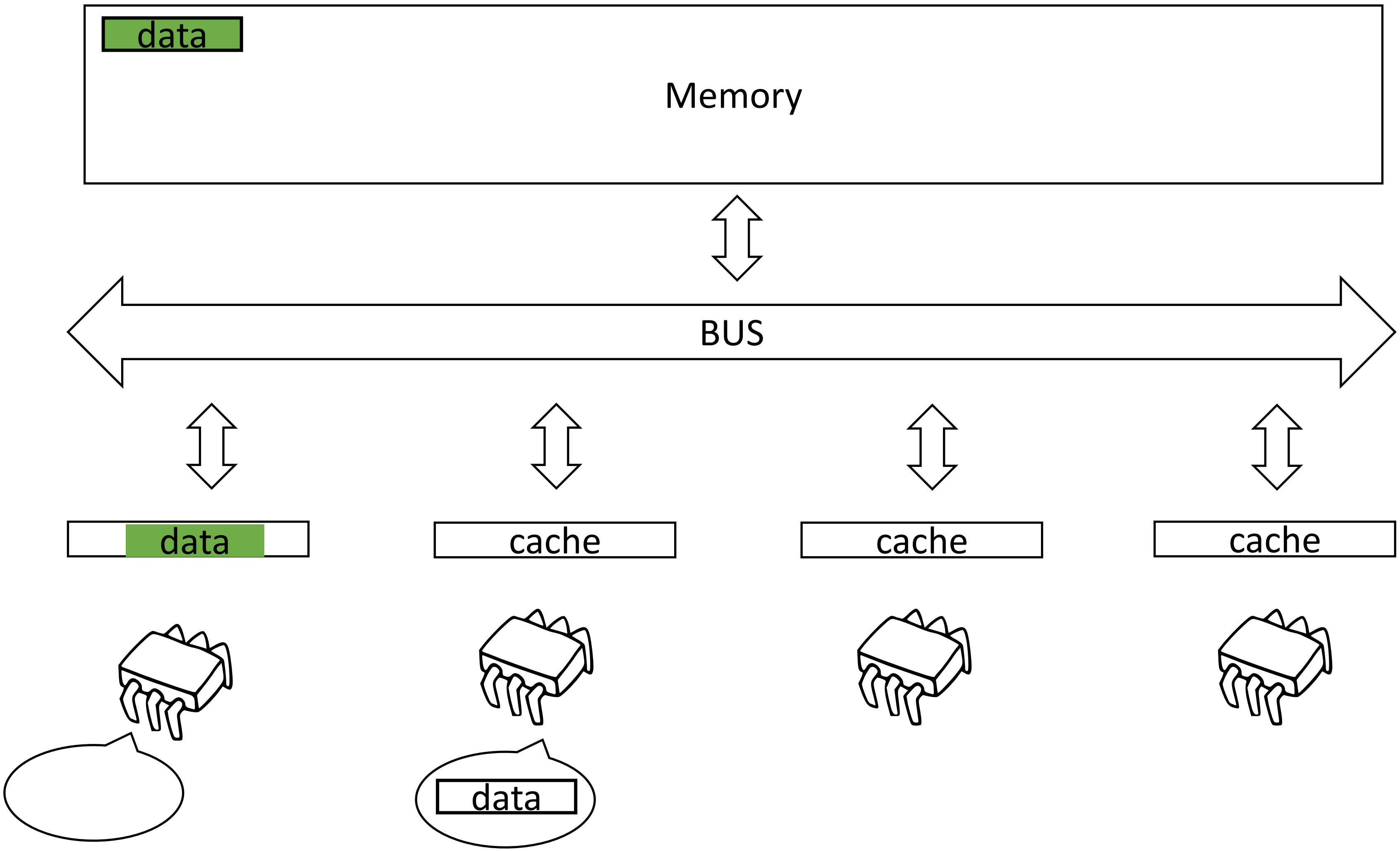
Memory Model



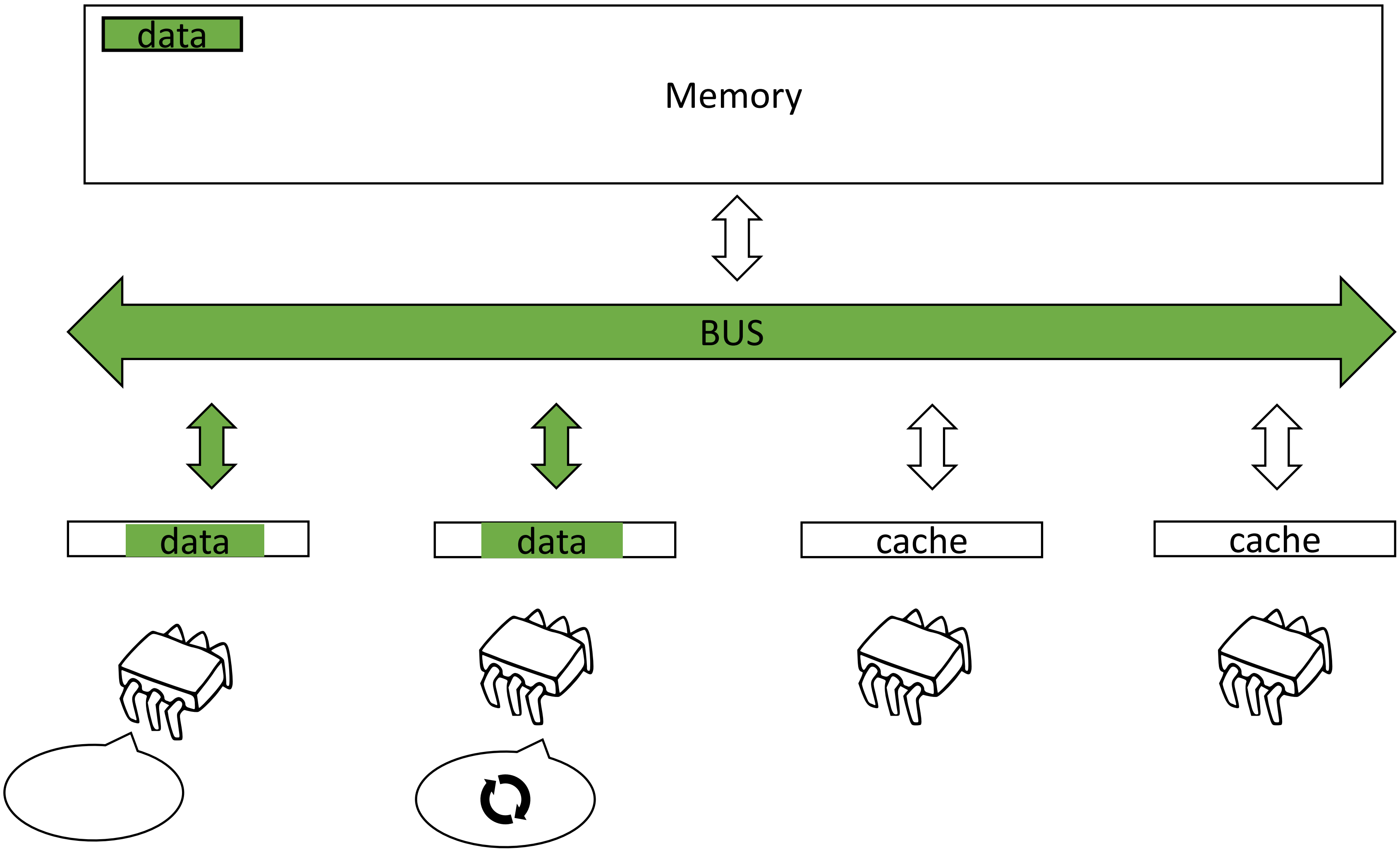
Memory Model



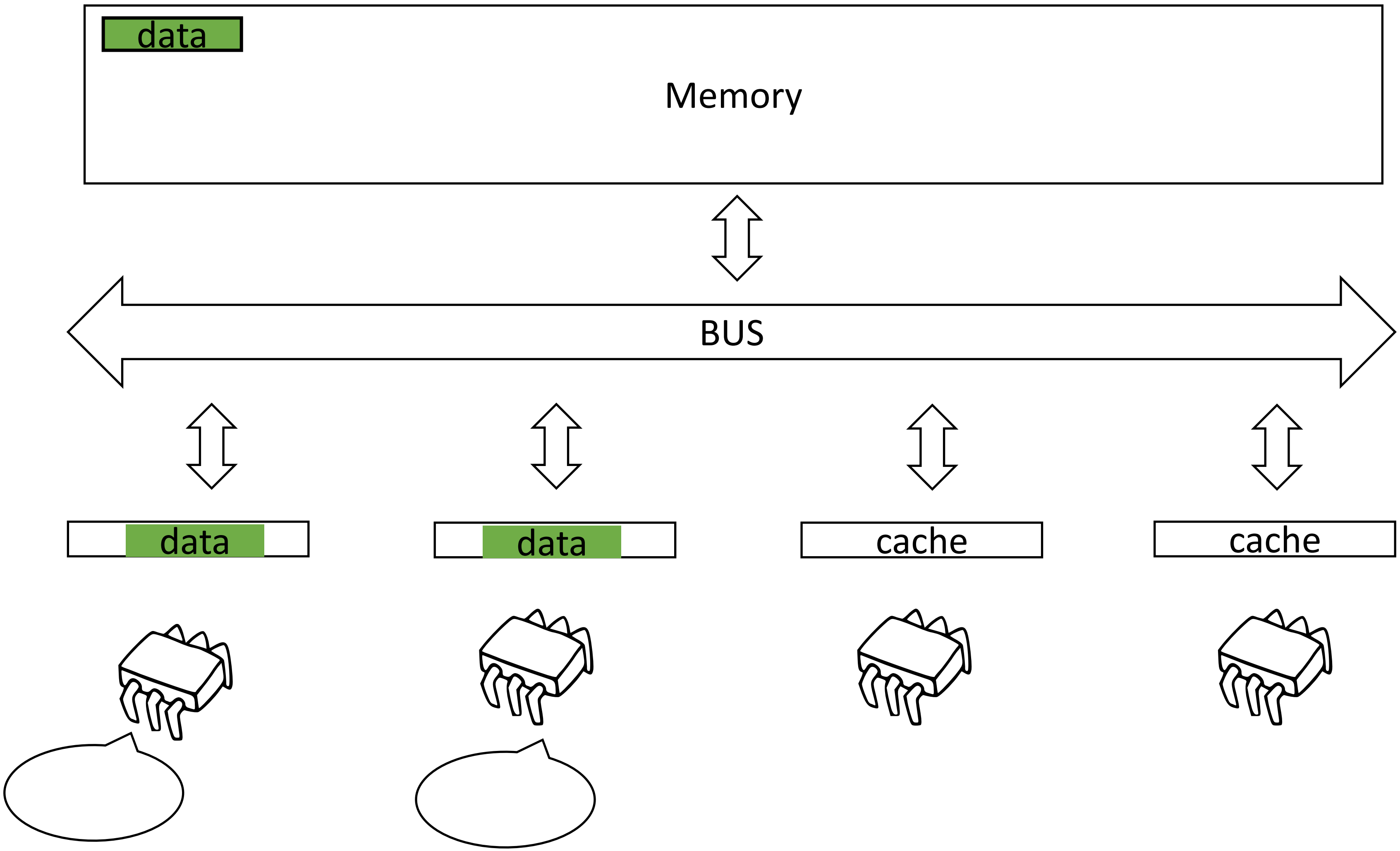
Memory Model



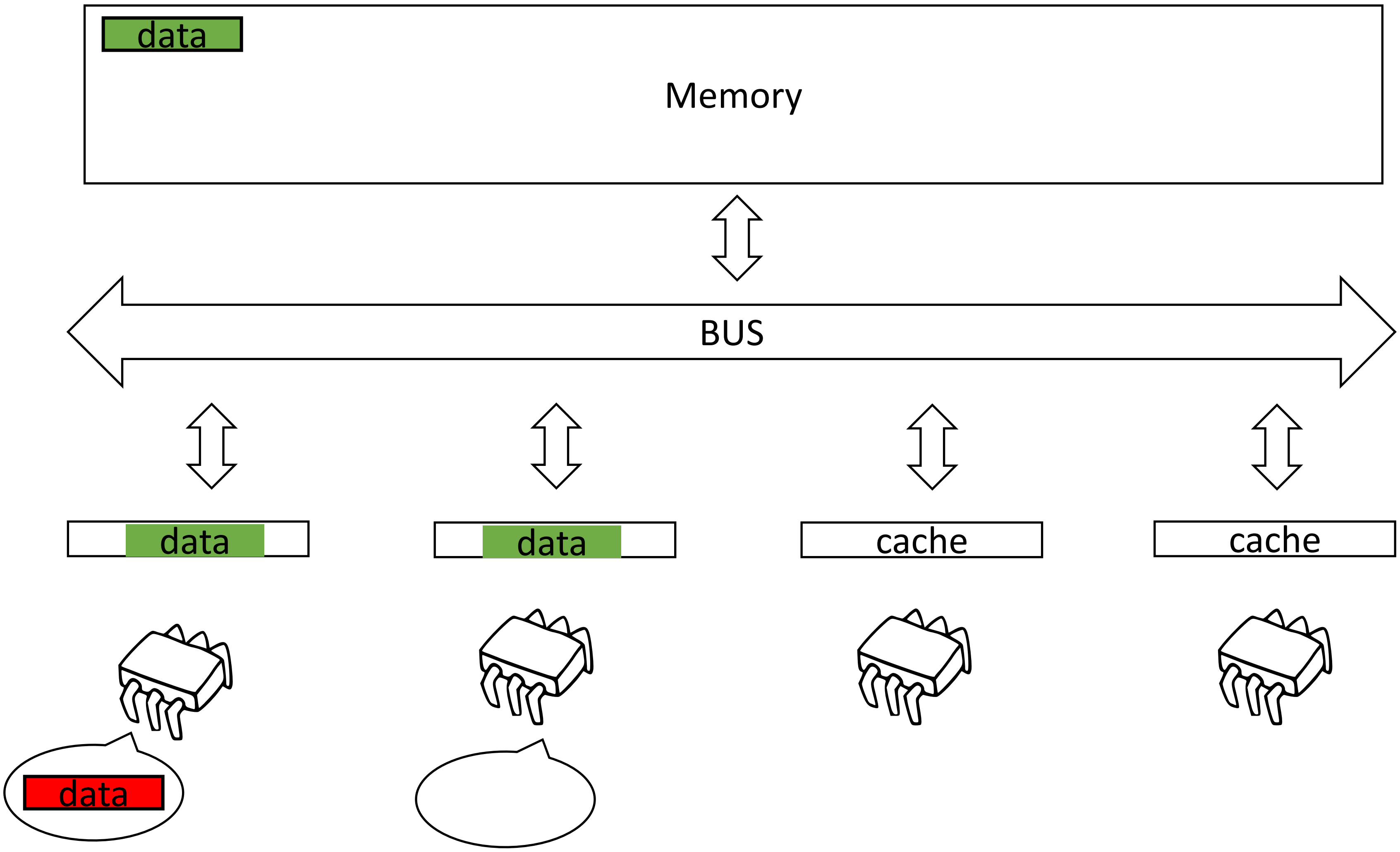
Memory Model



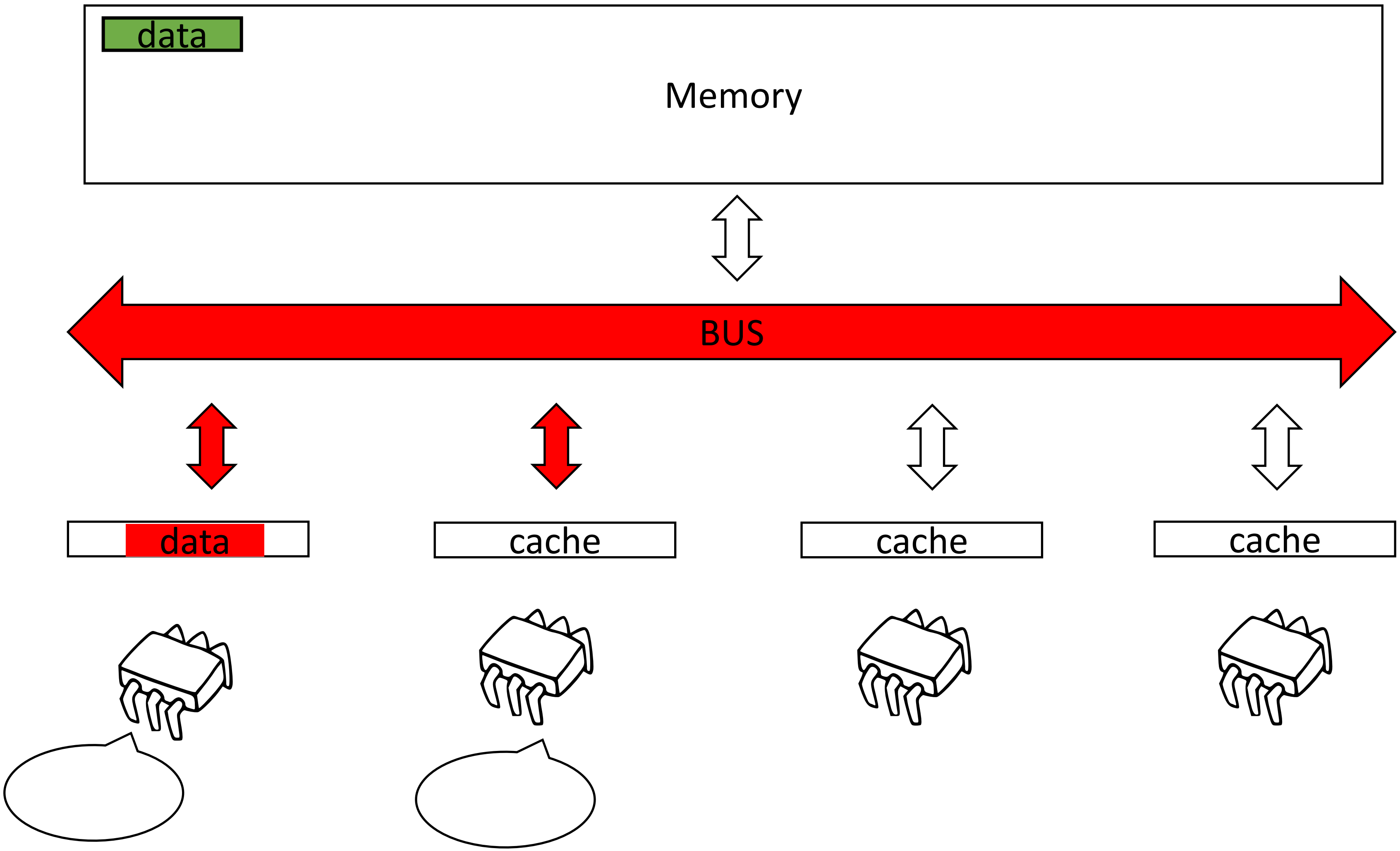
Memory Model



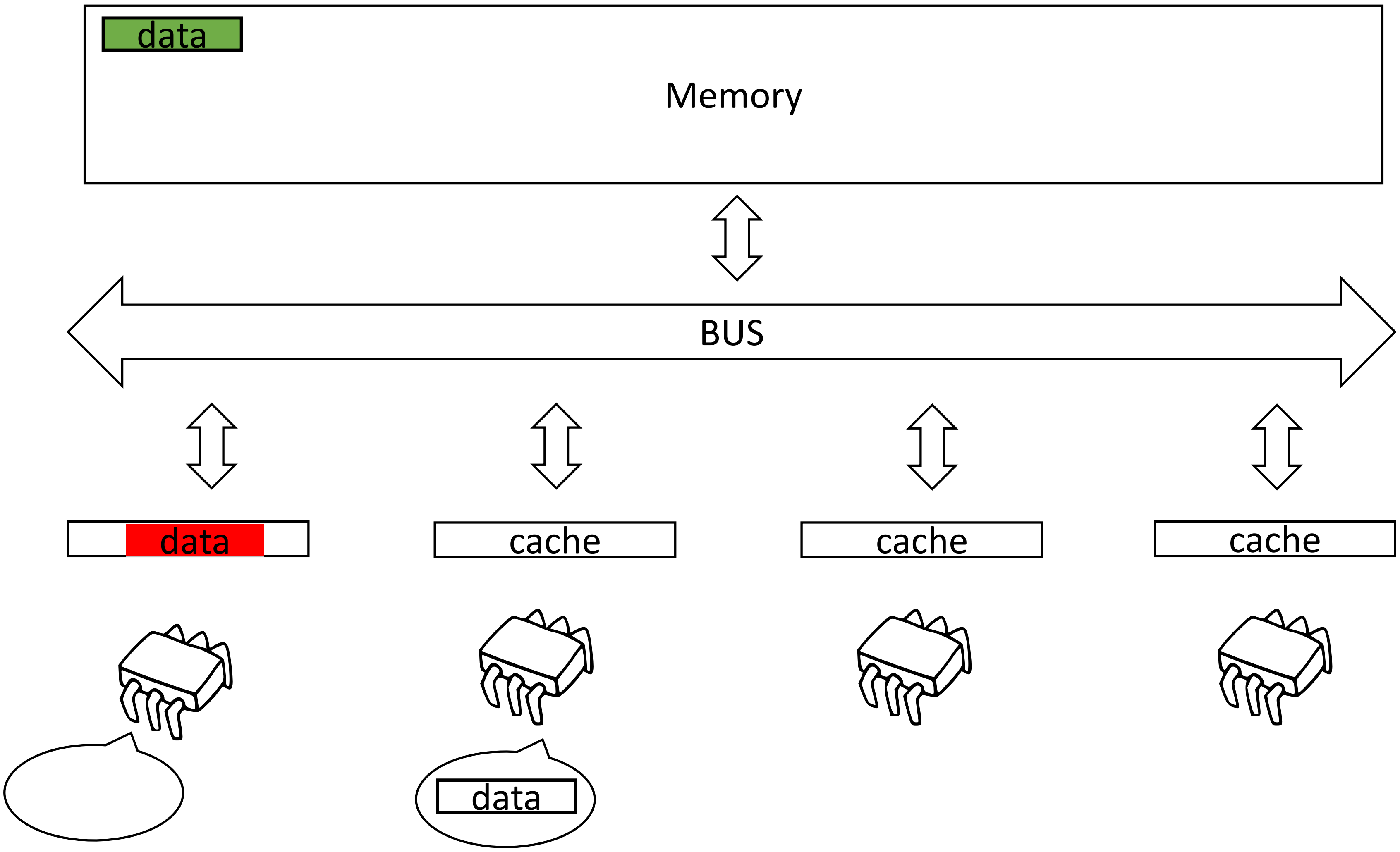
Memory Model



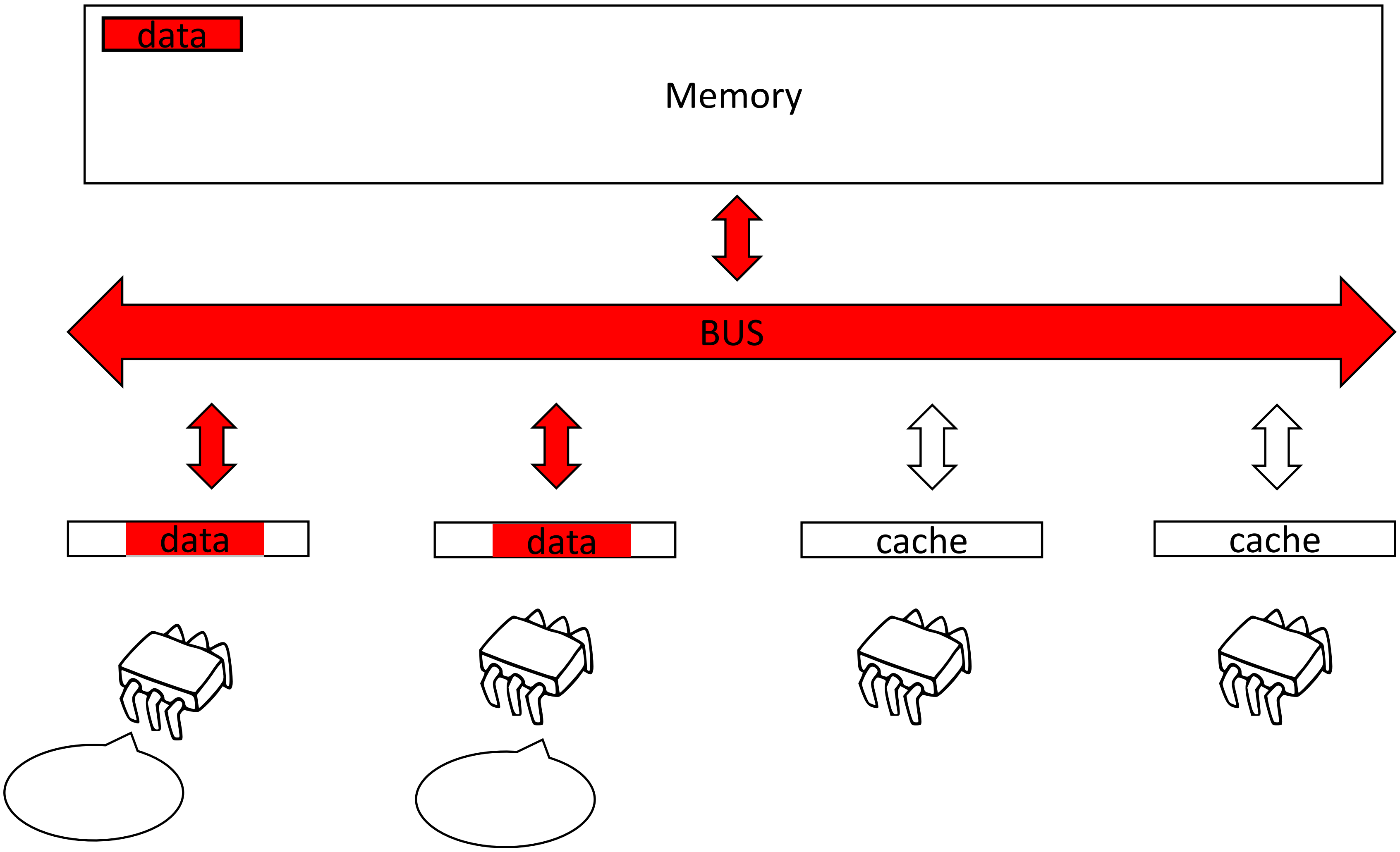
Memory Model



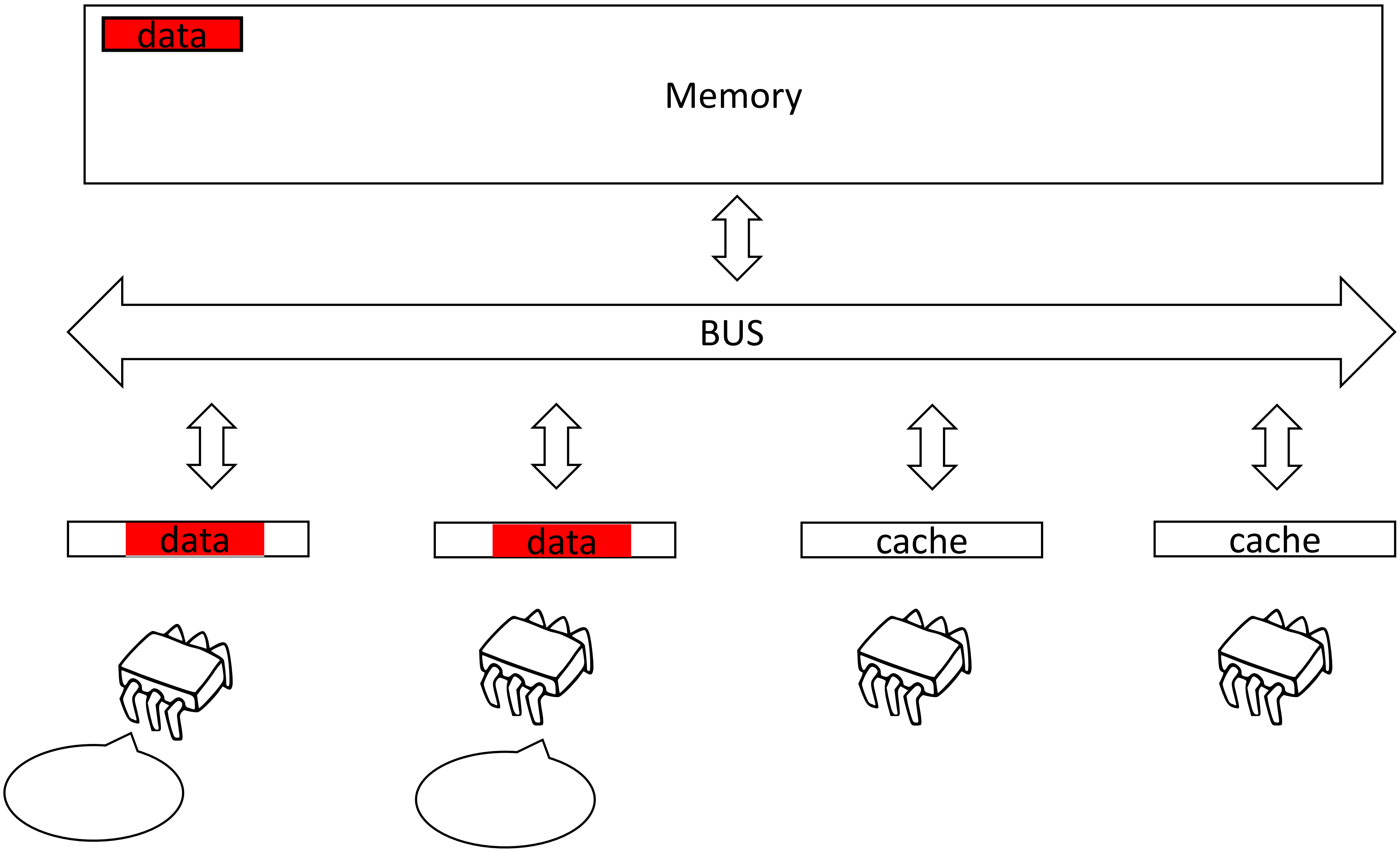
Memory Model



Memory Model



Memory Model



Lock implementations

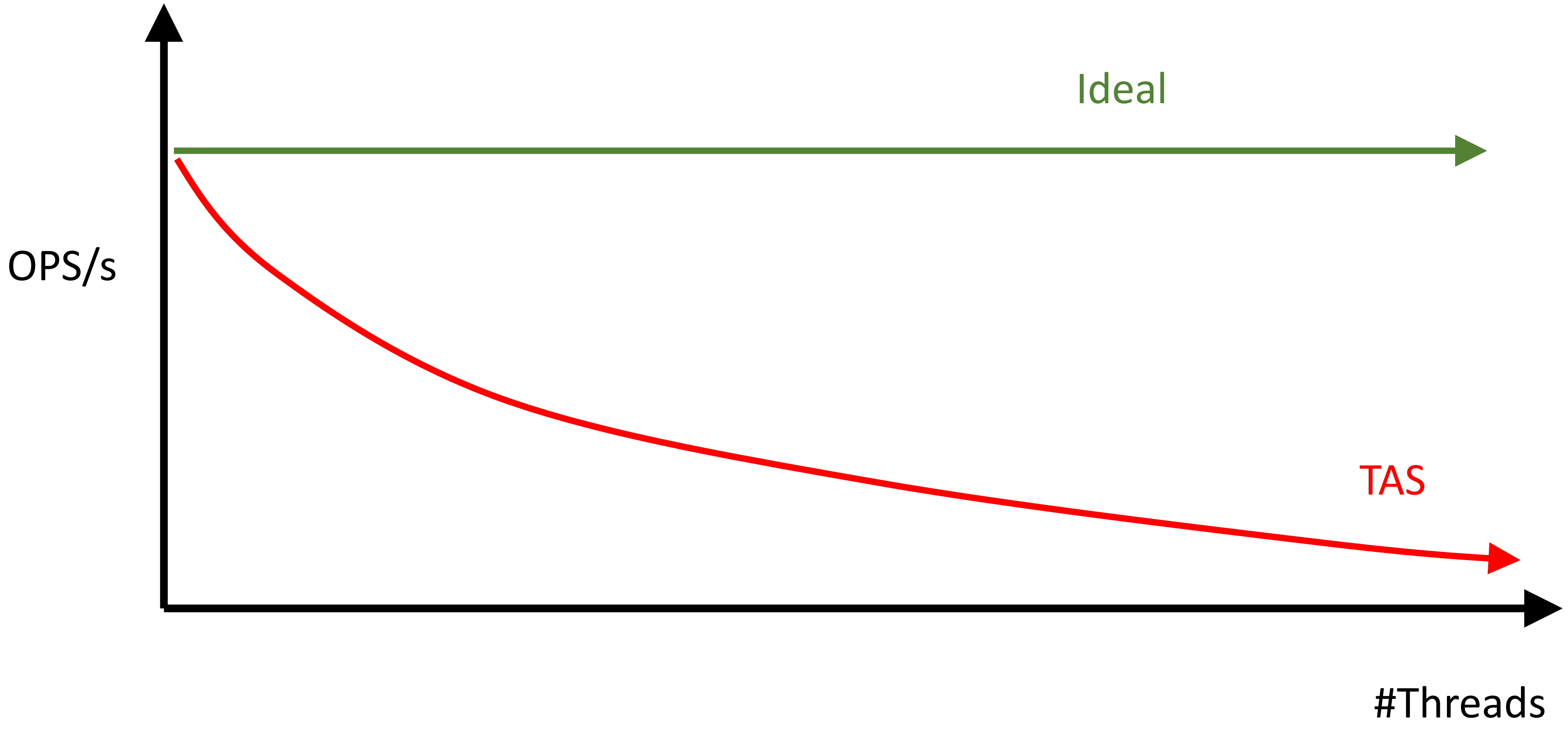
Test-and-set spin lock

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

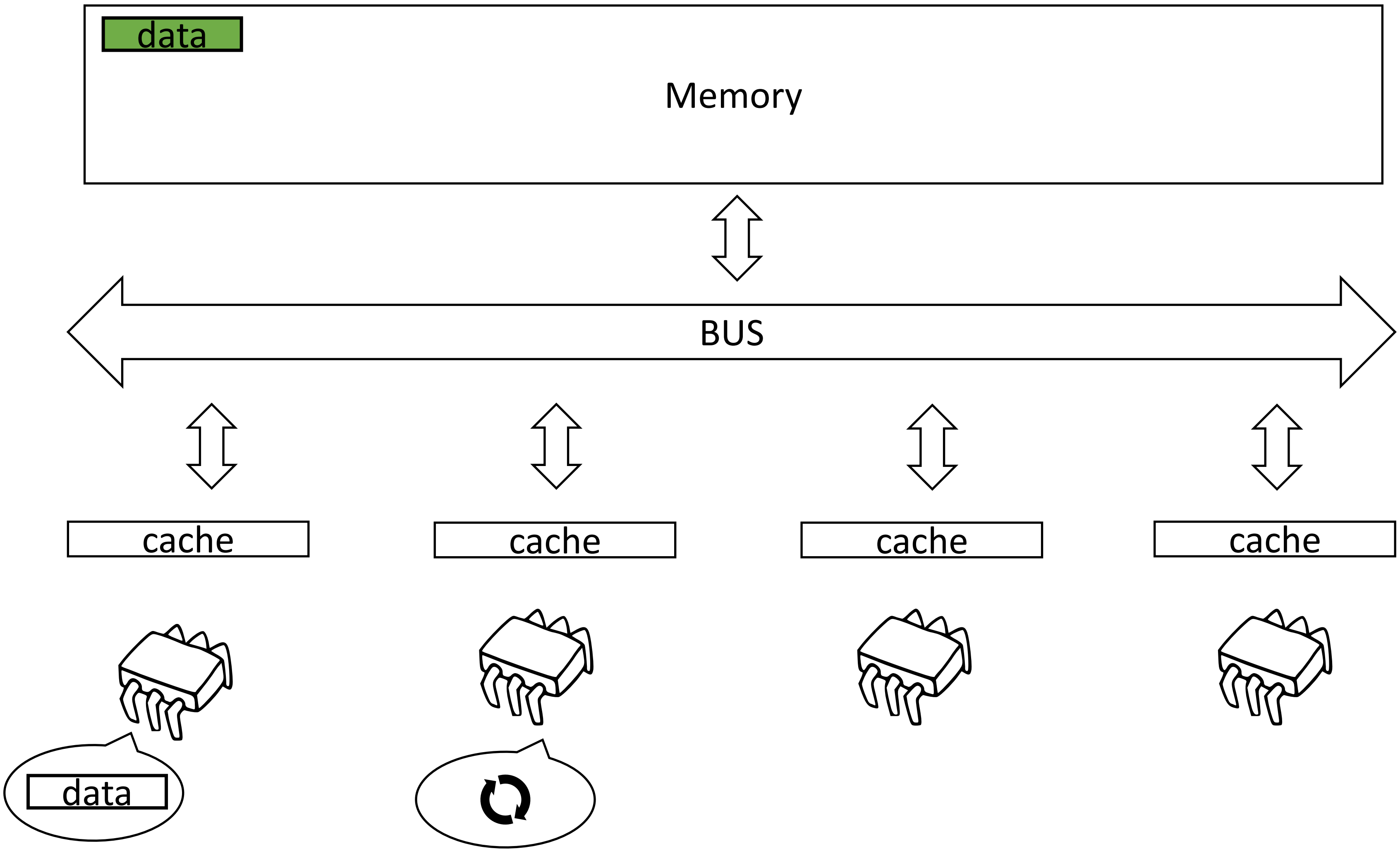
```
int lock = 0;
void acquire(int *lock){
  while(XCHG(lock, 1));
}
```

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

Results



Memory Model

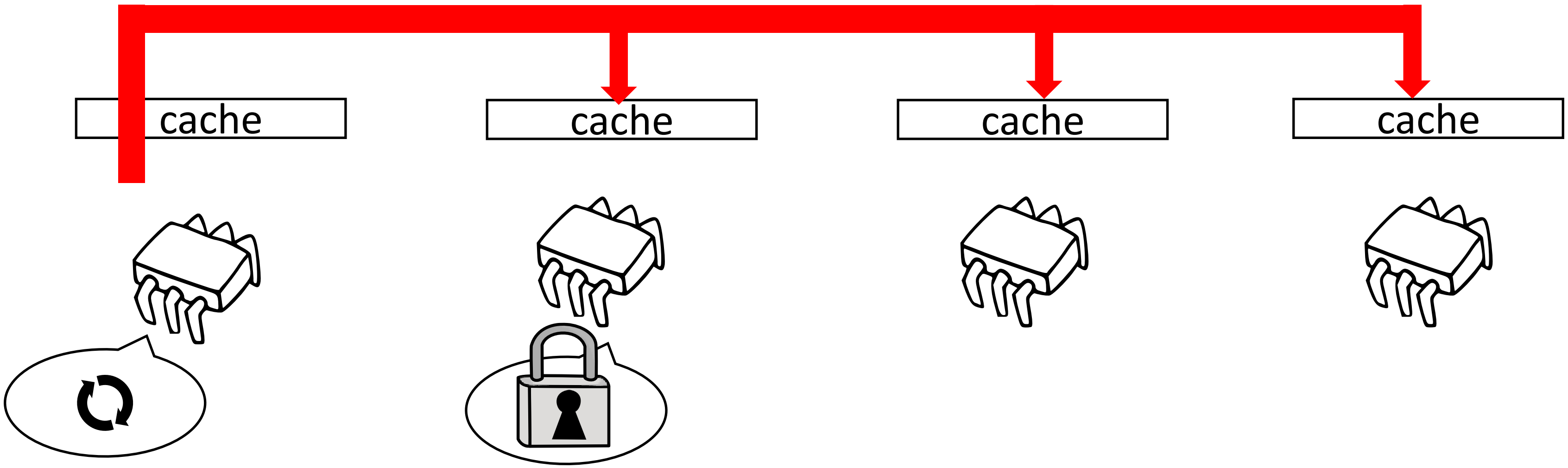


Test-and-set spin lock

- 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;
}
```

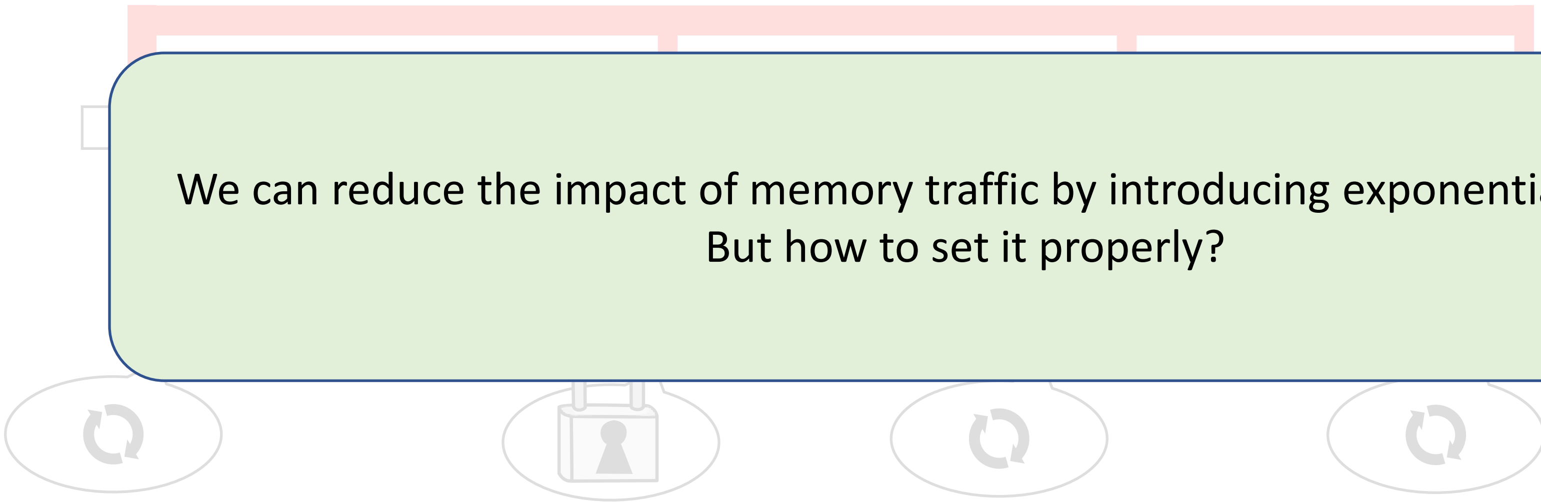


Test-and-set spin lock

- 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;
}
```



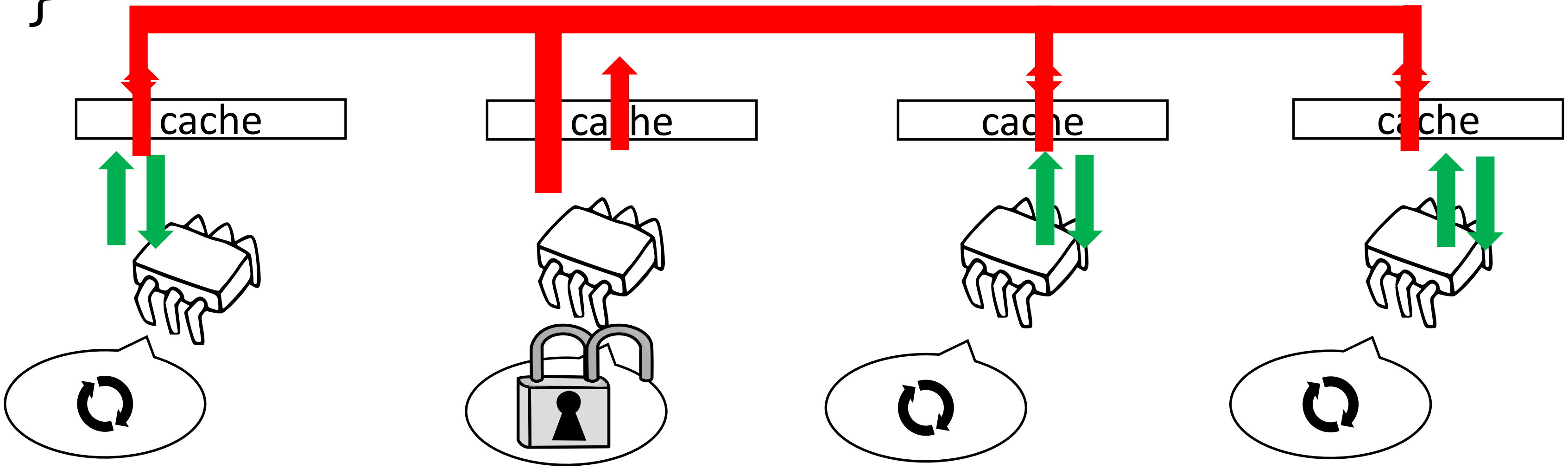
We can reduce the impact of memory traffic by introducing exponential back off!
But how to set it properly?

Test-and-test-and-set spin lock

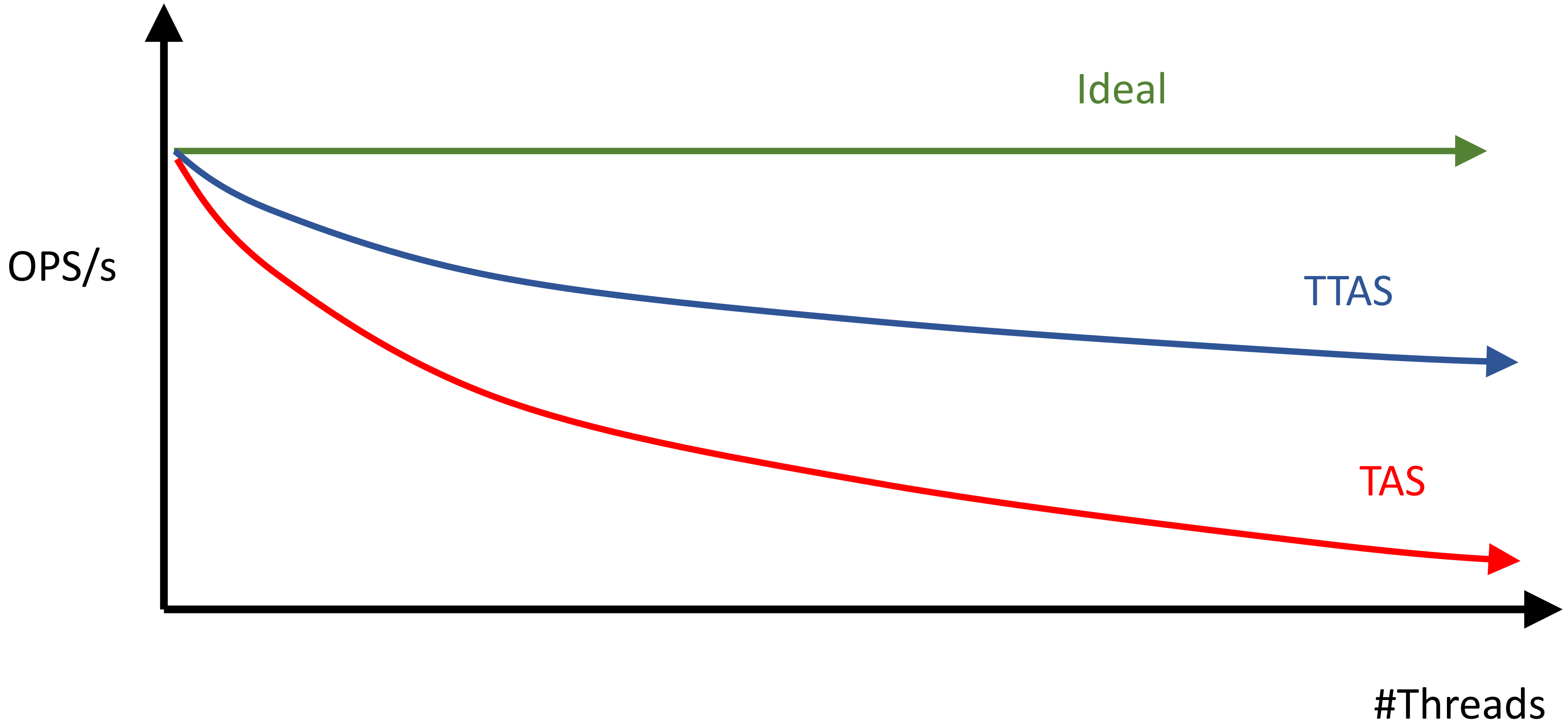
- Like test-and-set, but spins by reading the value of the lock
- Traffic is generated only upon lock handover

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

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



Results

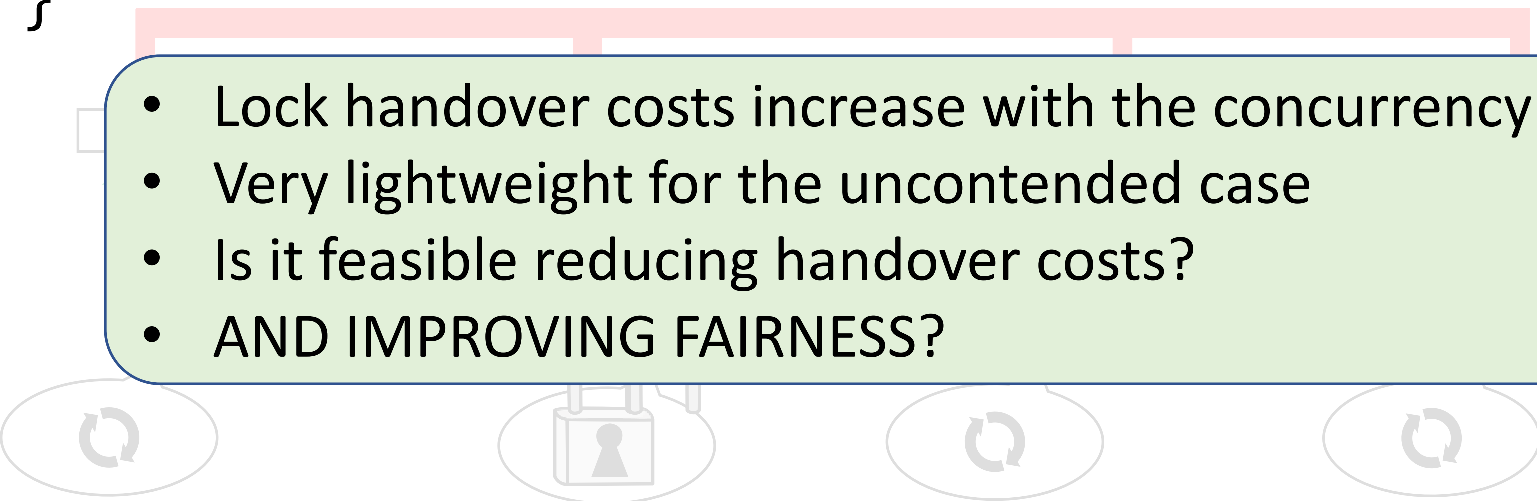


Test-and-test-and-set spin lock

- 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

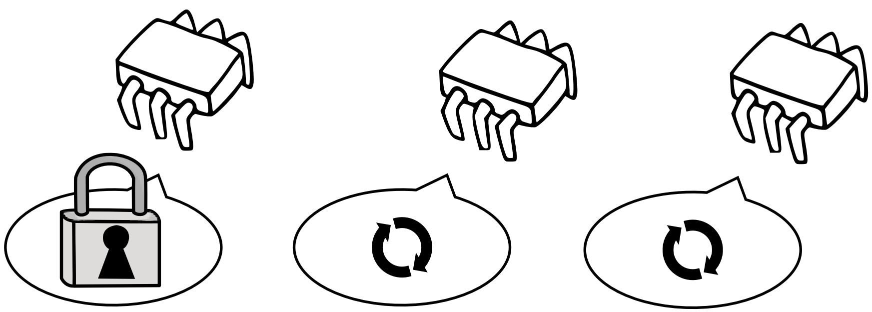
- Ensure fairness
- Similar structure w.r.t. TTAS spinlock
  - One variable updated once at each acquisition (better than TTAS)
  - Write-1-Read-N variable updated at each release (same as TTAS)
- How?

Anderson queue lock

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

Ticket = ~~0 1 2~~ 3

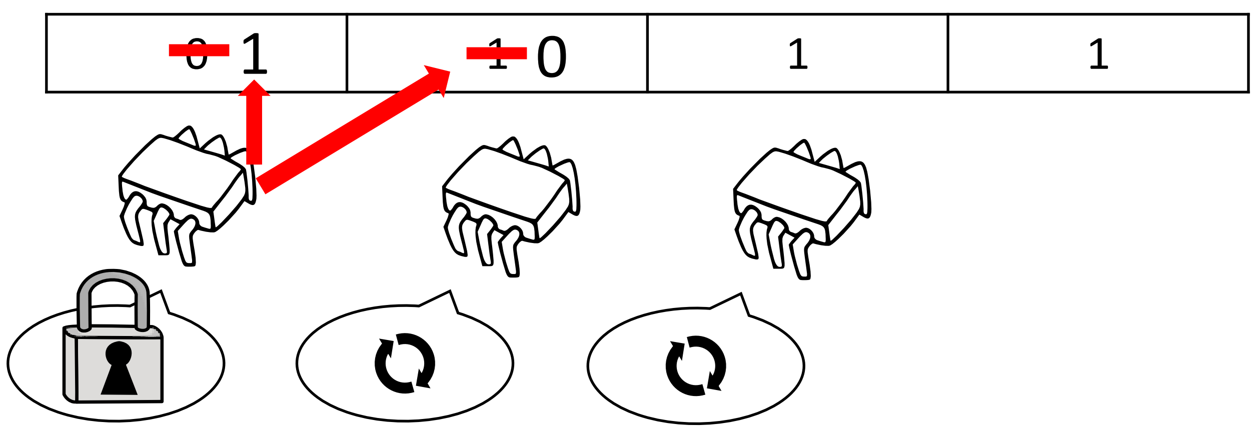
0	1	1	1



Anderson queue lock

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

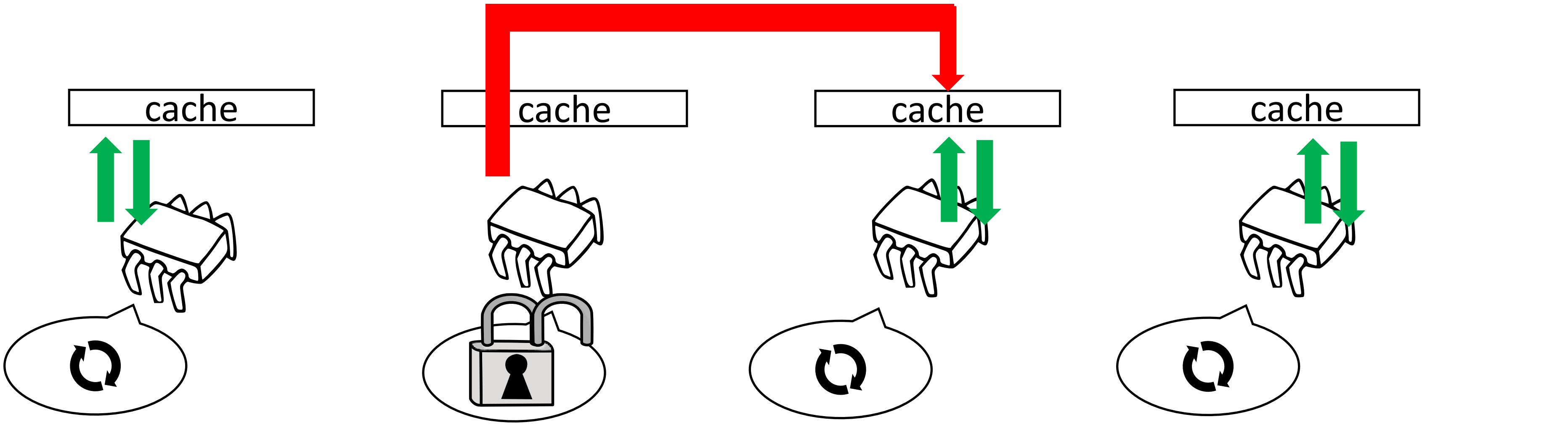
Ticket = ~~0 1 2~~ 3



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;
}
```

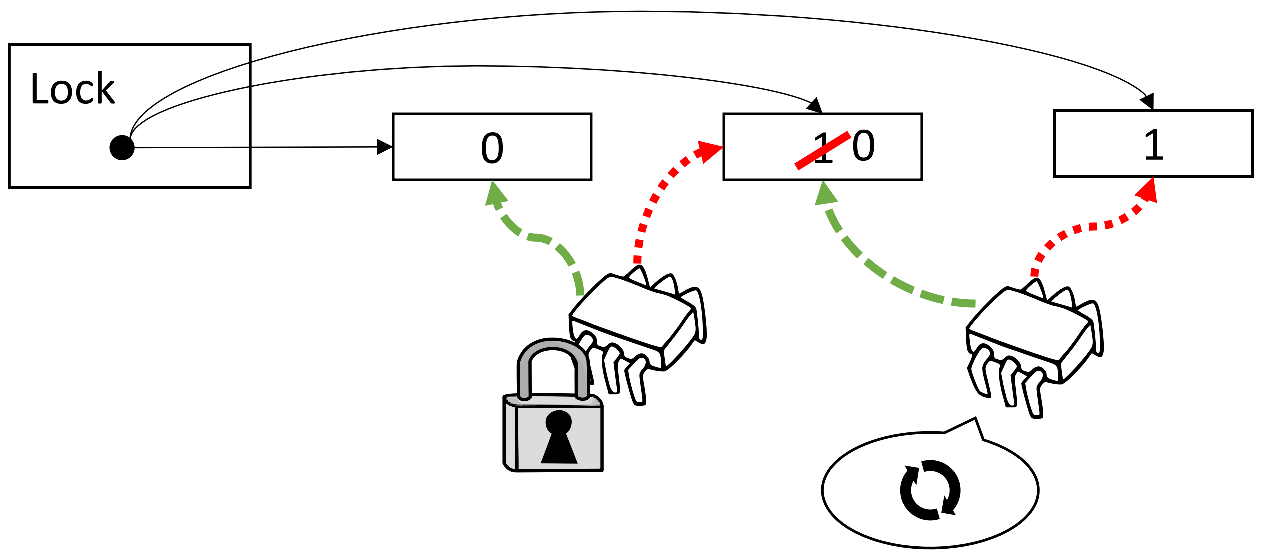


Anderson queue lock

- 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

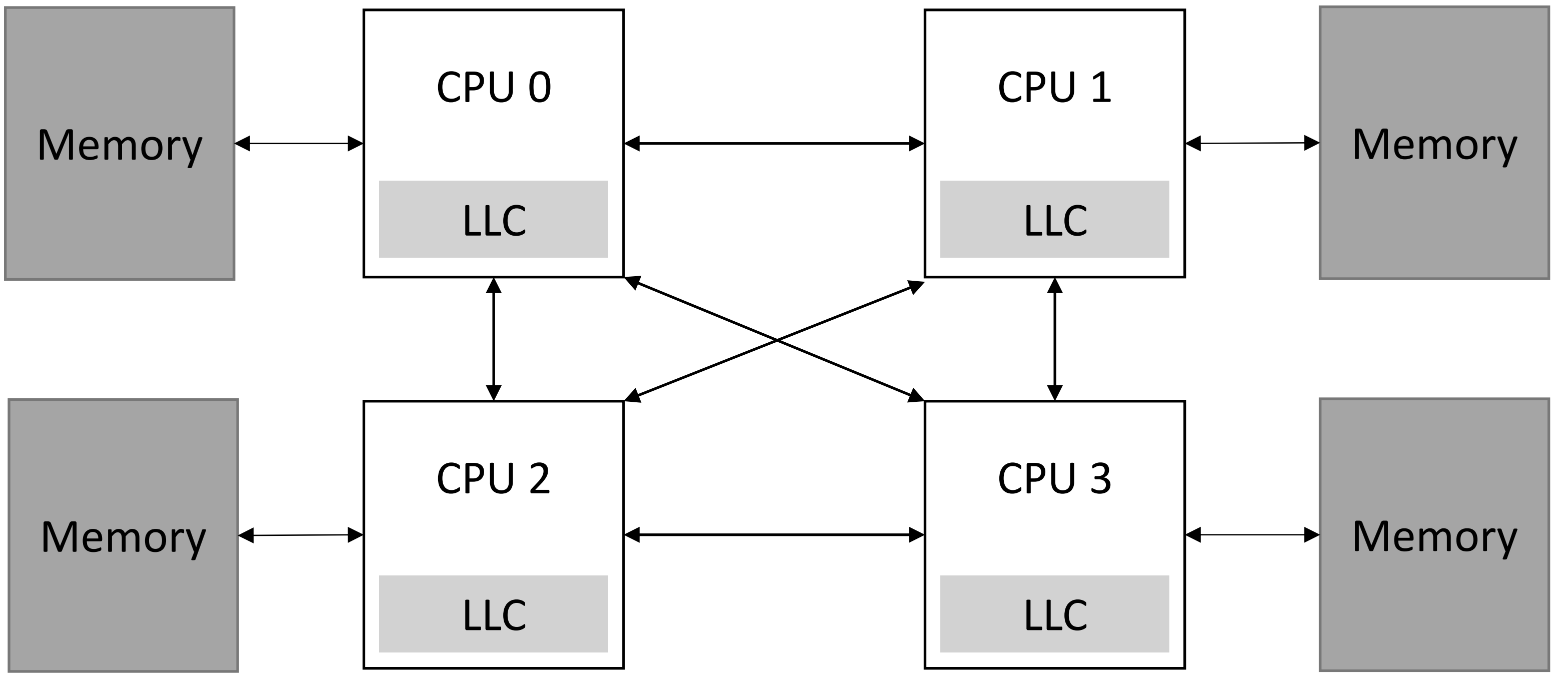


CLH queue lock

- 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)

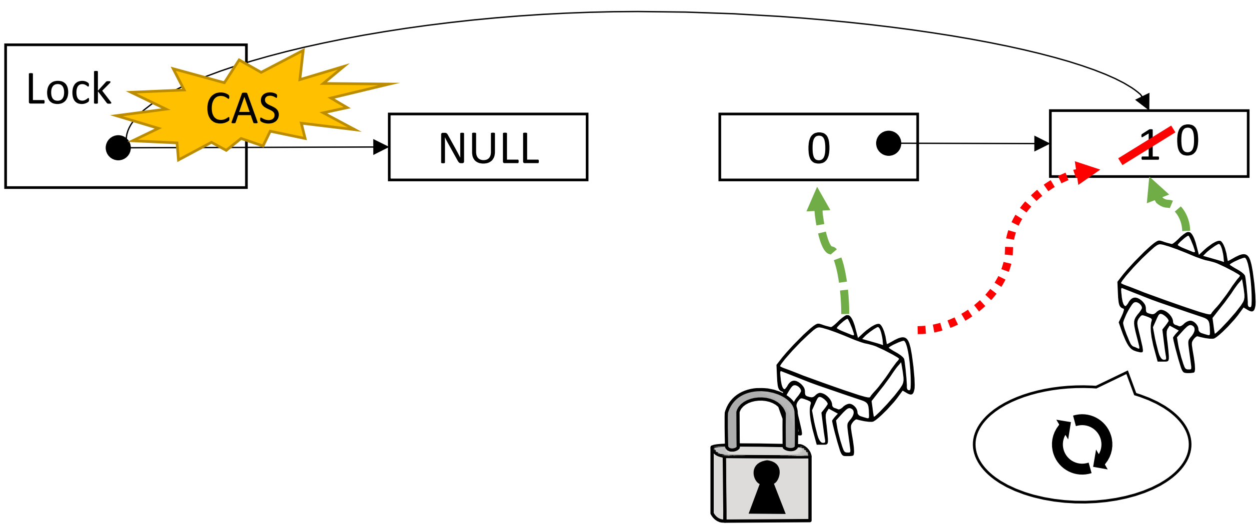
- Cons:
  - Slightly increased memory footprint
- Let:
  - T be the number of threads
  - L be the number of locks
- Space Usage
  - CLH = O(L+T)
  - Anderson = O(LT)
  - TAS, TTAS, Ticket = O(L)

NUMA


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

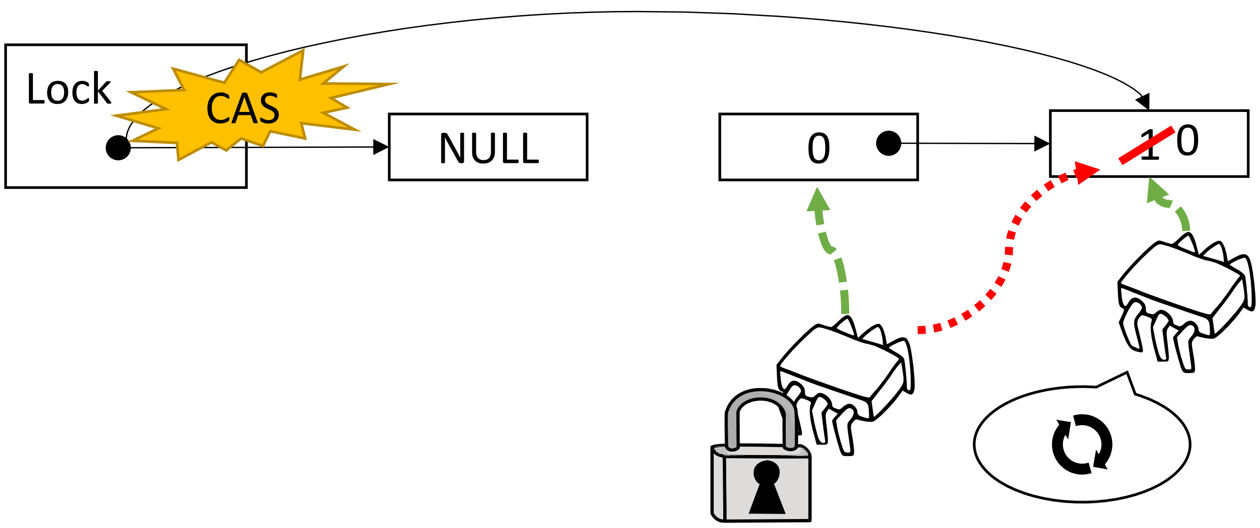
MCS lock

- 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



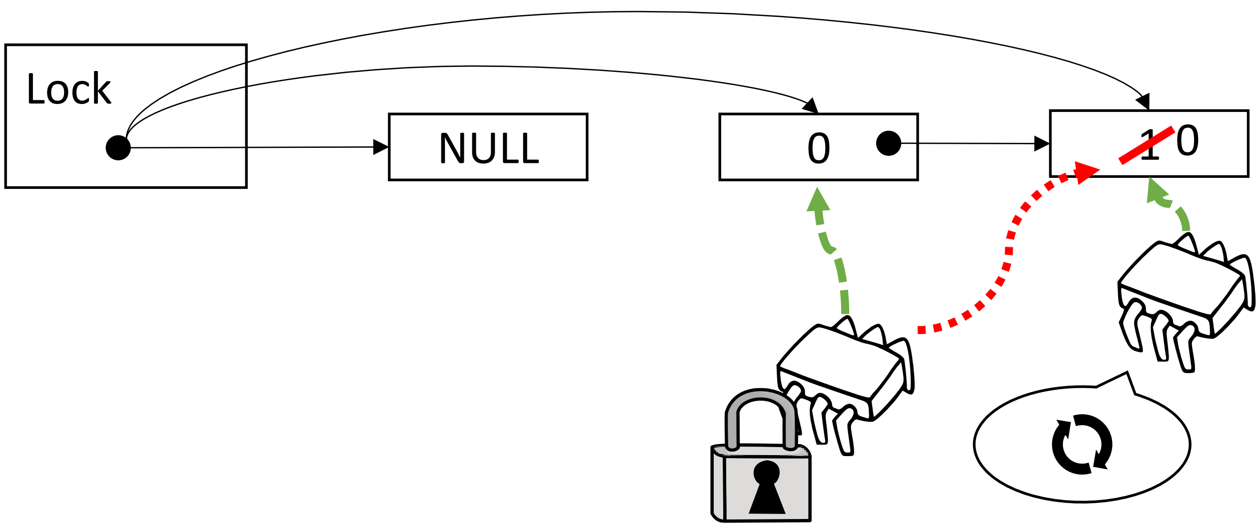
MCS lock

- 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 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

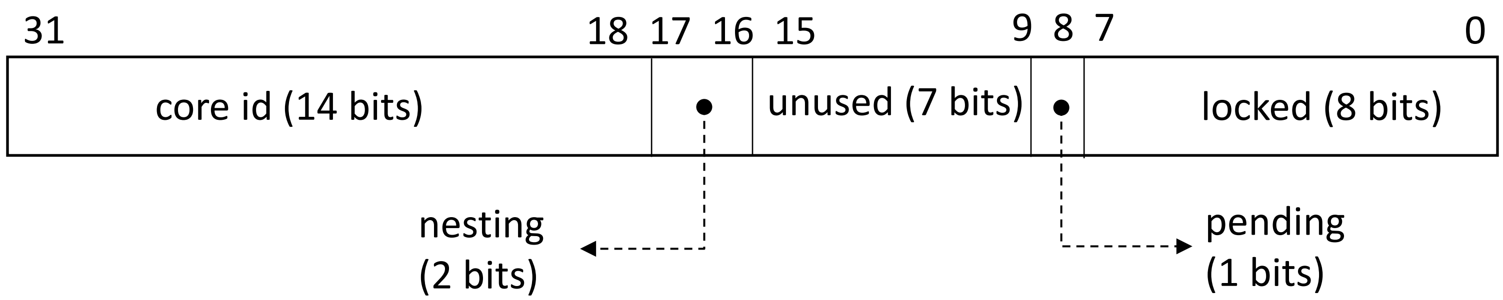


MCS queue lock

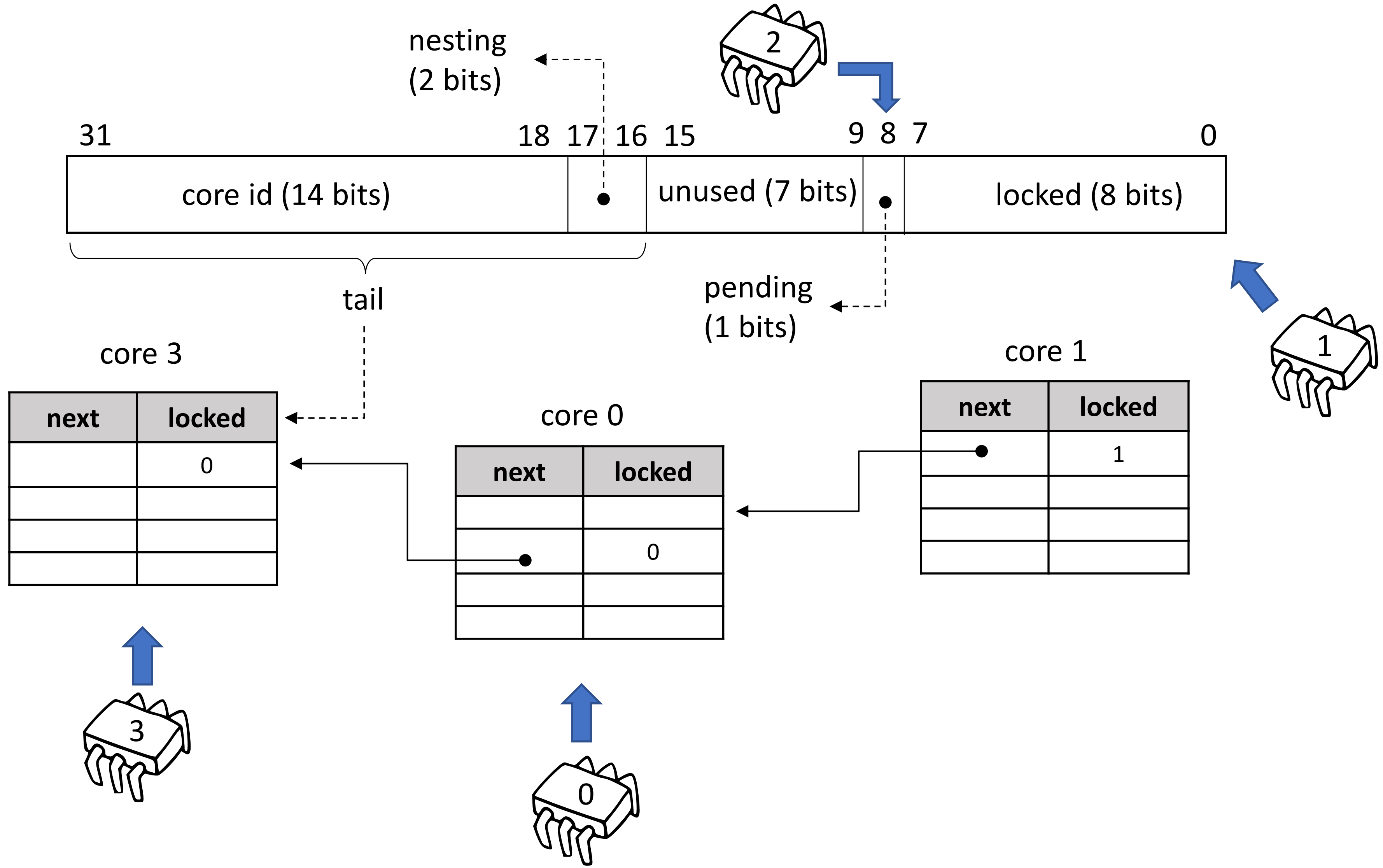
- Pros:
  - One variable updated once at each acquisition (like Ticket lock)
  - Write-1-Read-1 variable updated once per release (better than (T)TAS and Ticket)
  - No-remote spinning
- Cons:
  - Slightly increased memory footprint
- Let:
  - T be the number of threads
  - L be the number of locks
- 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

1	2	3	4	5	6	→	6	5	4	3	2	1

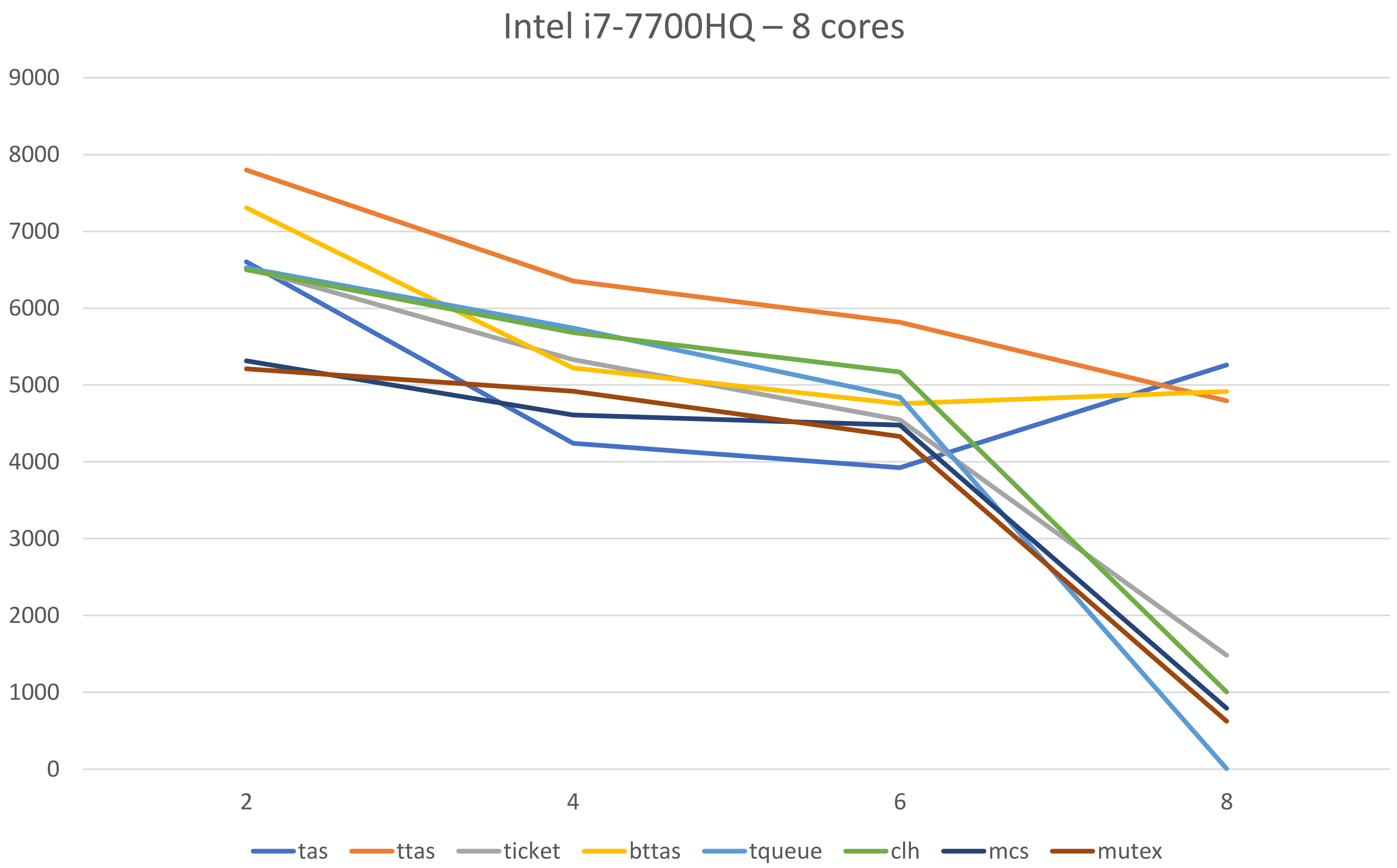
- This is done within a critical section

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

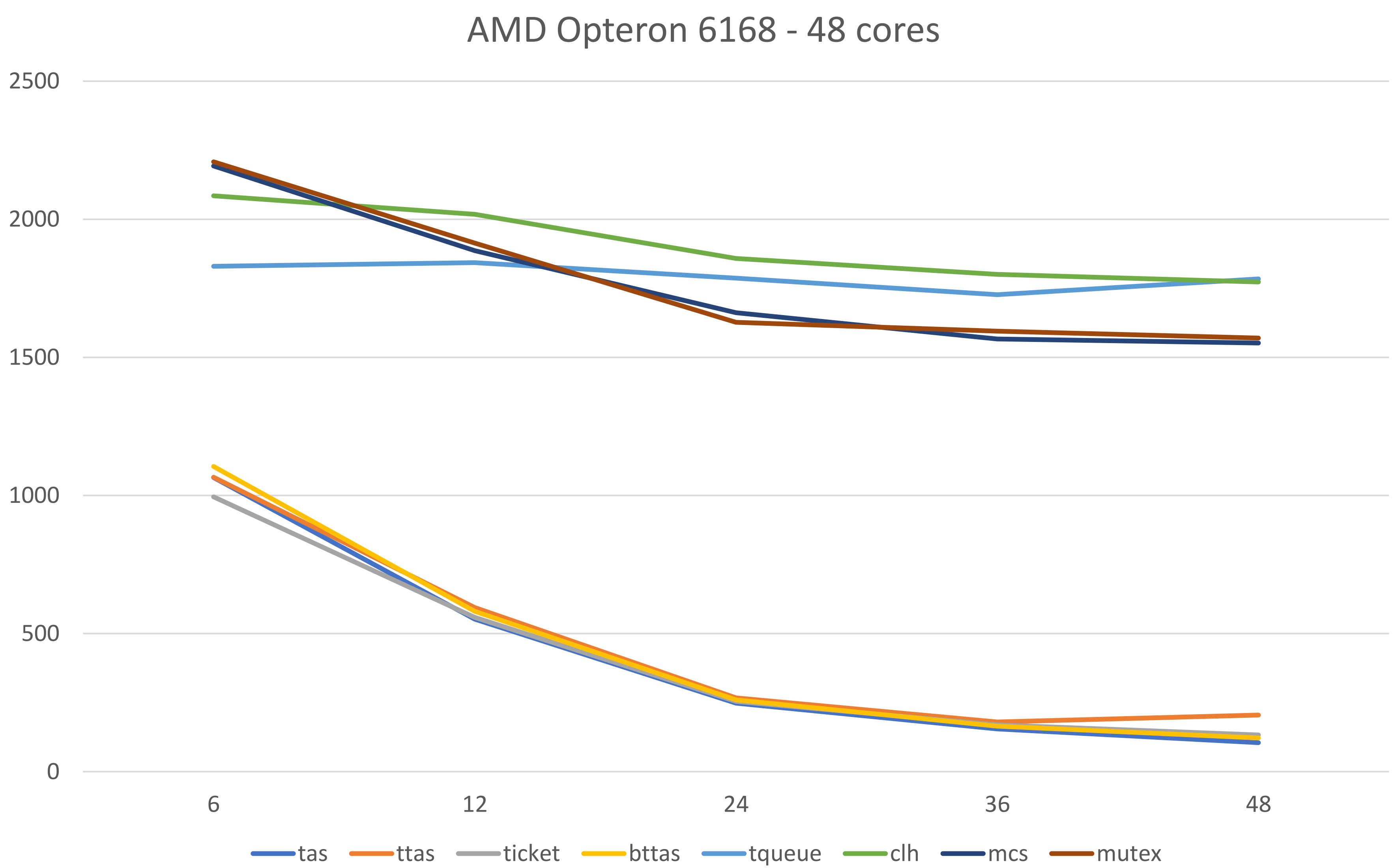
- Performance Metric:
  - Throughput = #Flips per second

One lock
to rule them all…

Performance

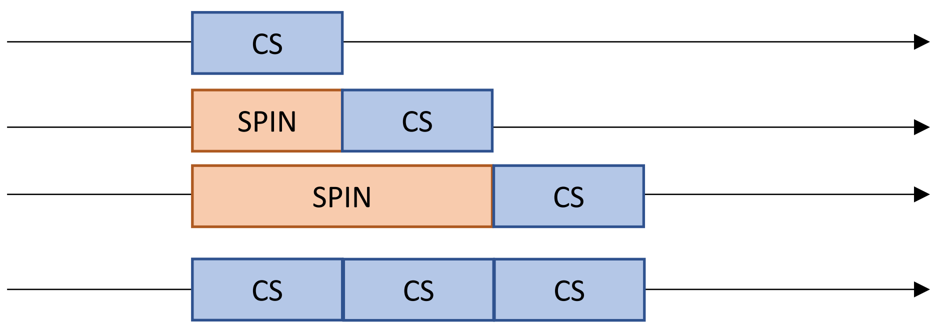


Performance

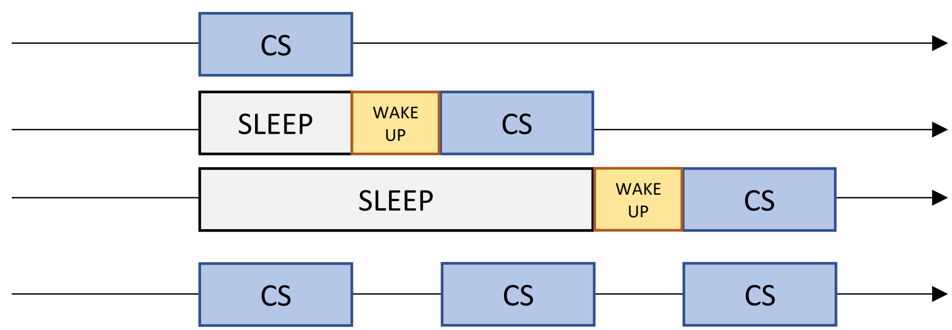


At the beginning was… Spin vs Sleep

	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	

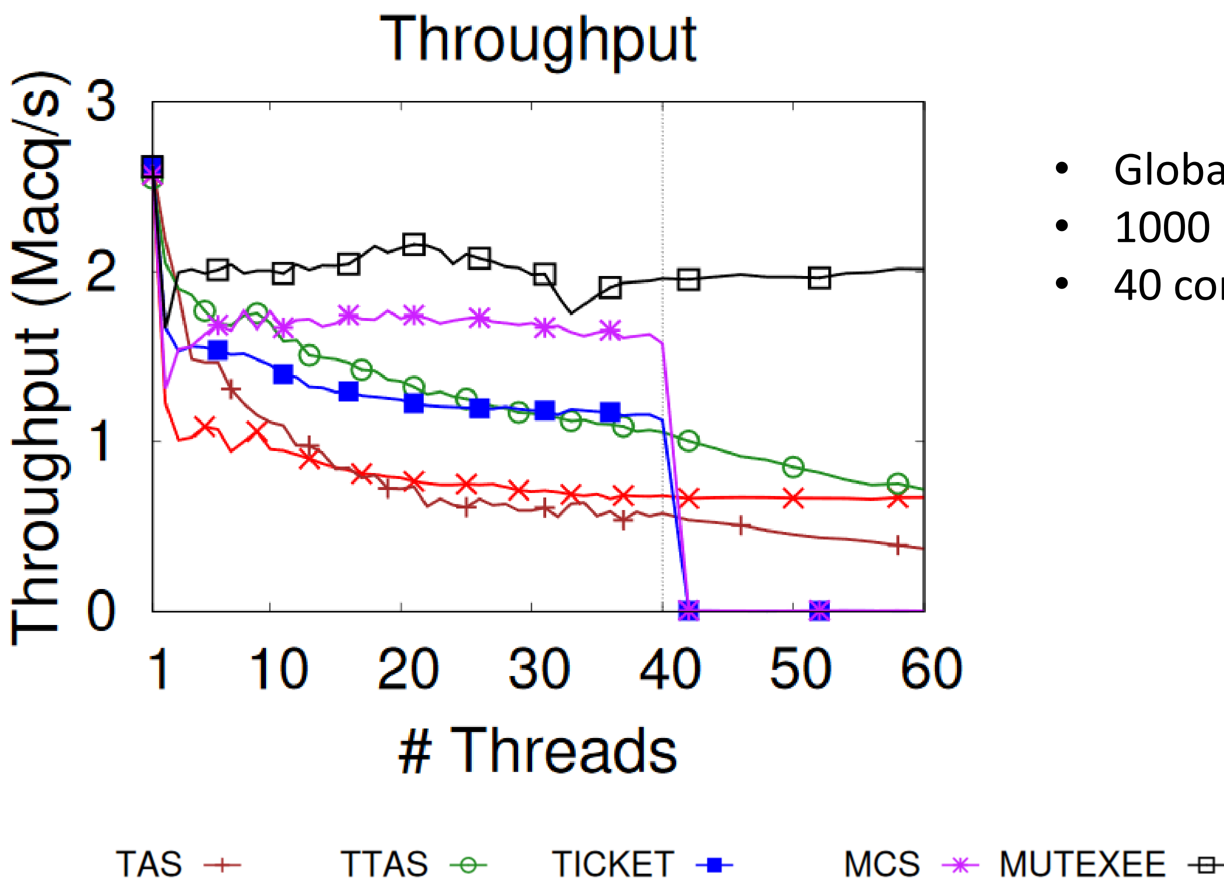
unlock()

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

Credits: Falsafi et all. “Unlocking energy”

An example - MutexEE

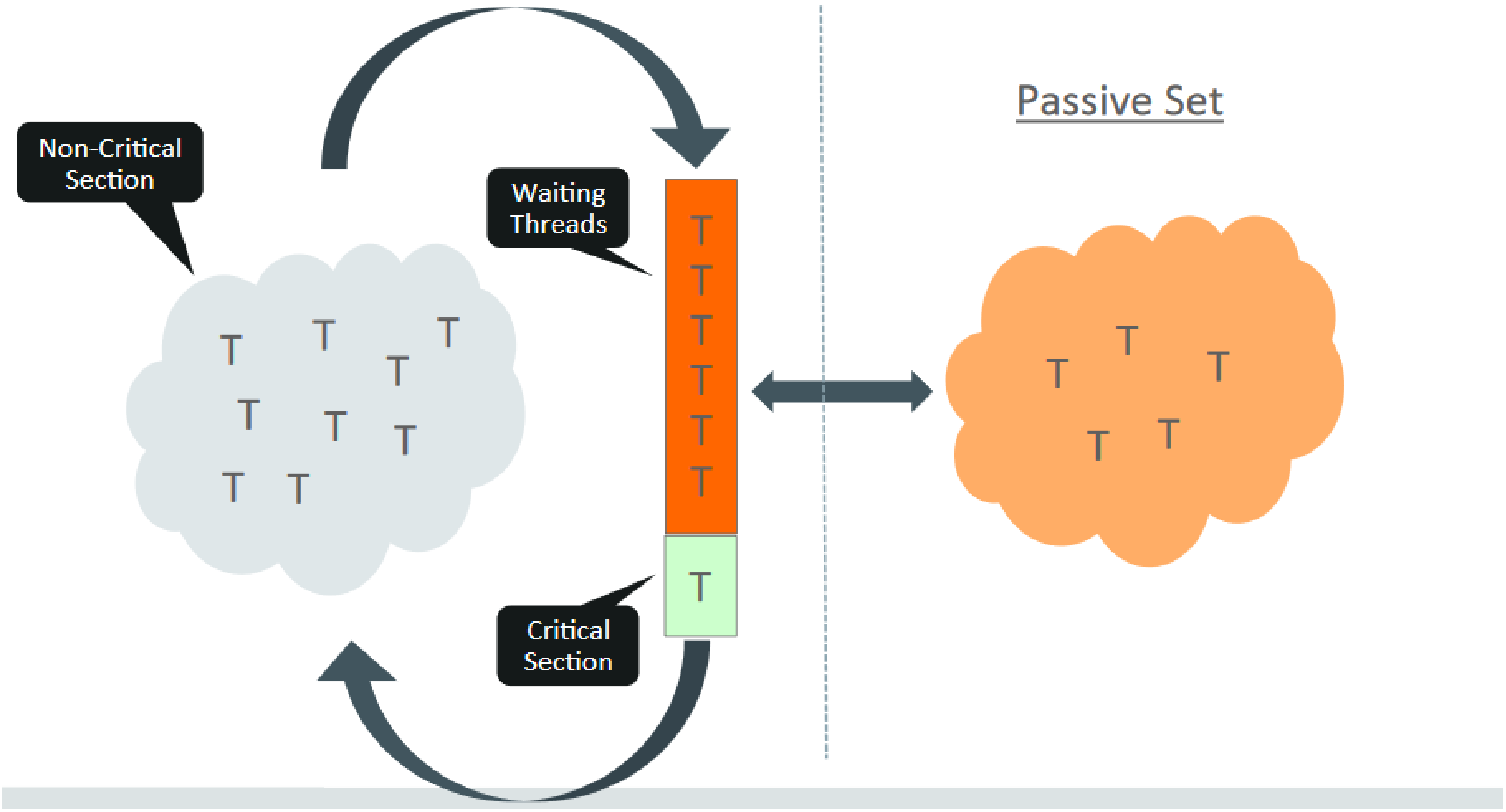
- MutexEE is a pthread_mutex optimized for throughput and energy efficiency



- Global lock
- 1000 cycles CS
- 40 cores

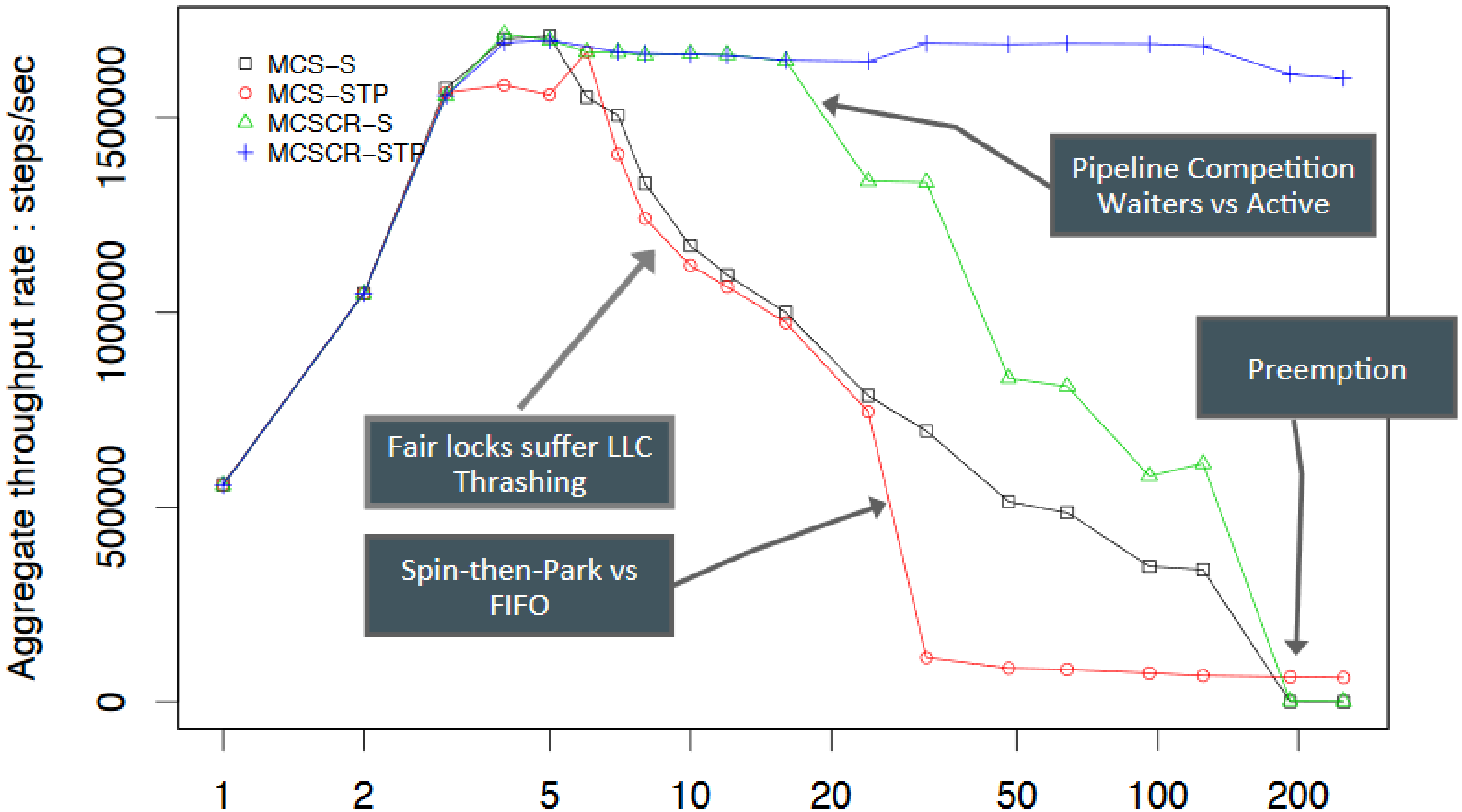
Credits: Falsafi et all. “Unlocking energy”

An example 2 – Malthusian locks



Credits: Dave Dice "Malthusian locks"

An example 2 – Malthusian locks



Credits: Dave Dice “Malthusian locks”