Multicore Synchronization

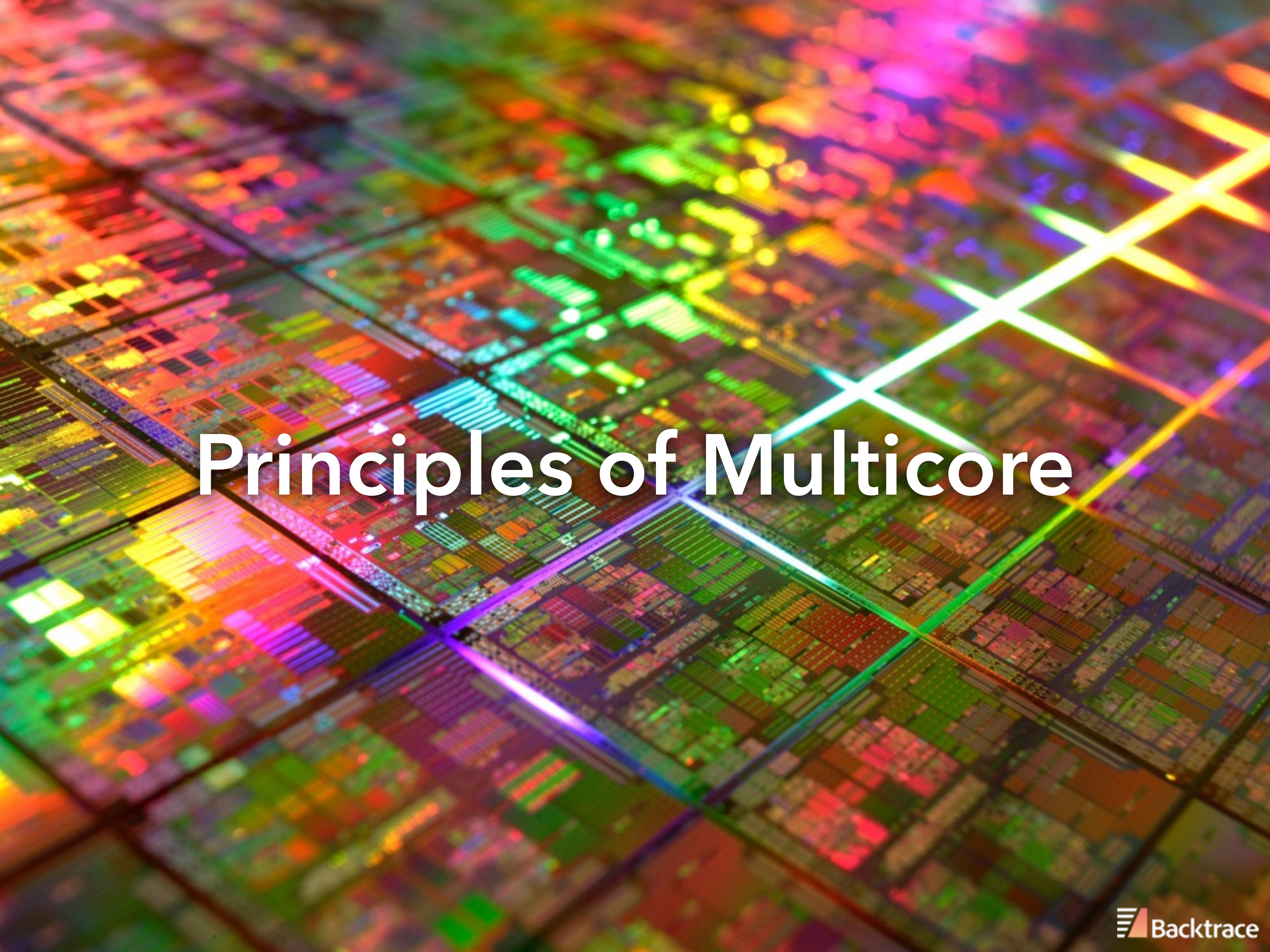
a pragmatic introduction

Backtrace

Multicore Synchronization

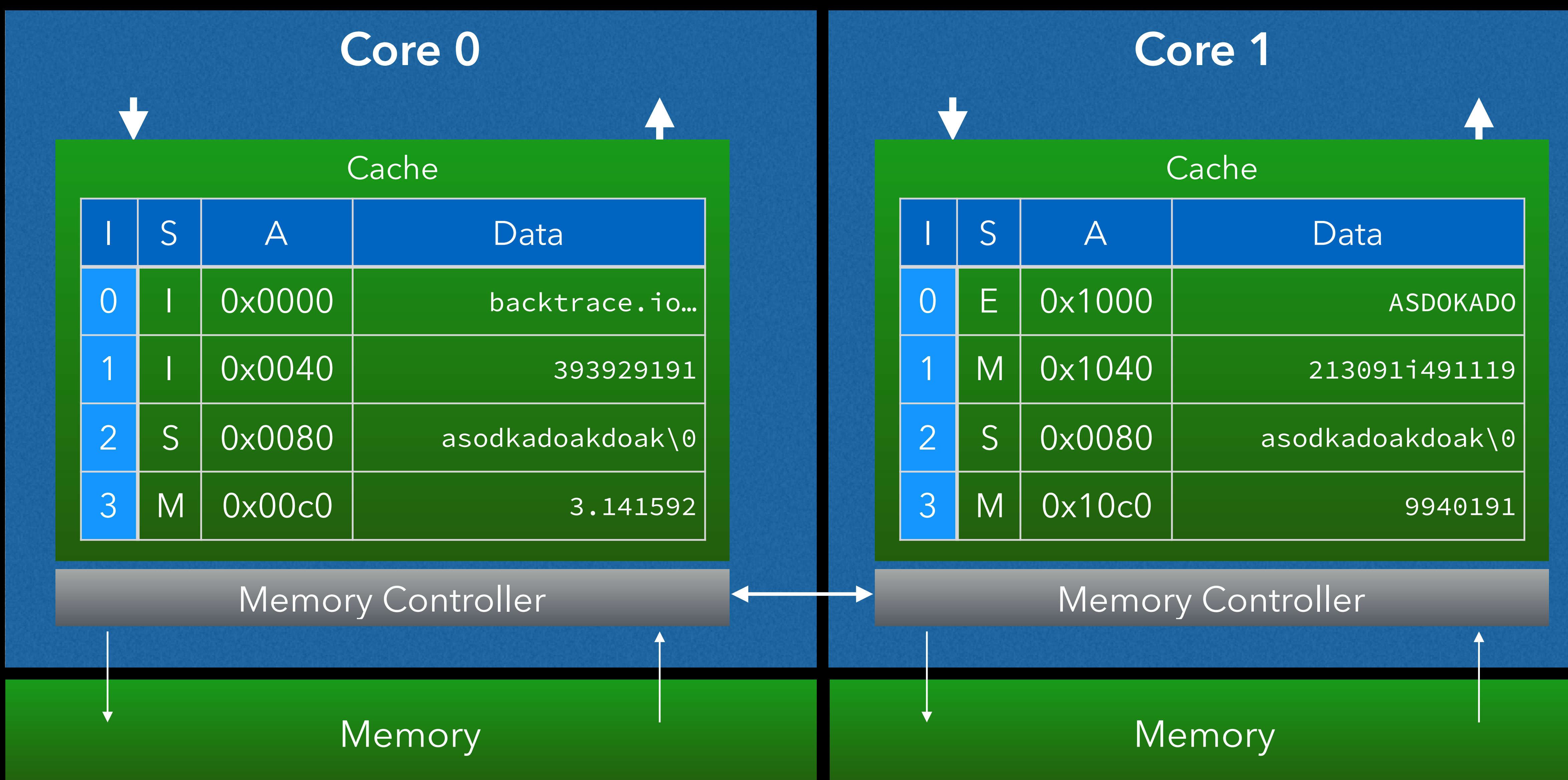
This is a talk on mechanical sympathy of parallel systems on modern multicore systems.

Understanding both your workload and your environment allows for effective optimization.


Principles of Multicore
Backtrace

Cache Coherency

Cache coherency guarantees the eventual consistency of shared state.



Cache Coherency

```
int x = 443; (&x = 0x20c4)
```

Thread 0

```
x = x + 10010;
```

Thread 1

```
printf("%d\n", x);
```

Core 0

Cache

I	S	A	Data
0	I	0x0000	backtrace.io…
1	I	0x0040	393929191
2	S	0x0080	asodkadoakdoak\0
3	M	0x00c0	3.141592

Memory Controller

Memory

Core 1

Cache

I	S	A	Data
0	E	0x1000	ASDOKADO
1	M	0x1040	213091i491119
2	S	0x0080	asodkadoakdoak\0
3	M	0x10c0	9940191

Memory Controller

Memory

Backtrace

Cache Coherency

```
Load x
```

Thread 0

```
x = x + 10010;
```

Thread 1

```
printf(“%d\n”, x);
```

Core 0

Cache

I	S	A	Data
0	I	0x0000	backtrace.io…
1	I	0x0040	393929191
2	S	0x0080	asodkadoakdoak\0
3	E	0x20c0	1921919119**443……..**

Memory Controller

Memory

Core 1

Cache

I	S	A	Data
0	E	0x1000	ASDOKADO
1	M	0x1040	213091i491119
2	S	0x0080	asodkadoakdoak\0
3	M	0x10c0	9940191

Memory Controller

Memory

Backtrace

Cache Coherency

```
Update the value of x
```

Thread 0

```
x = x + 10010;
```

Thread 1

```
printf("%d\n", x);
```

Core 0

Cache

I	S	A	Data
0	I	0x0000	backtrace.io…
1	I	0x0040	393929191
2	S	0x0080	asodkadoakdoak\0
3	M	0x20c0	1921919119**10453**………

Memory Controller

Memory

Core 1

Cache

I	S	A	Data
0	E	0x1000	ASDOKADO
1	M	0x1040	213091i491119
2	S	0x0080	asodkadoakdoak\0
3	M	0x10c0	9940191

Memory Controller

Memory

Backtrace

Cache Coherency

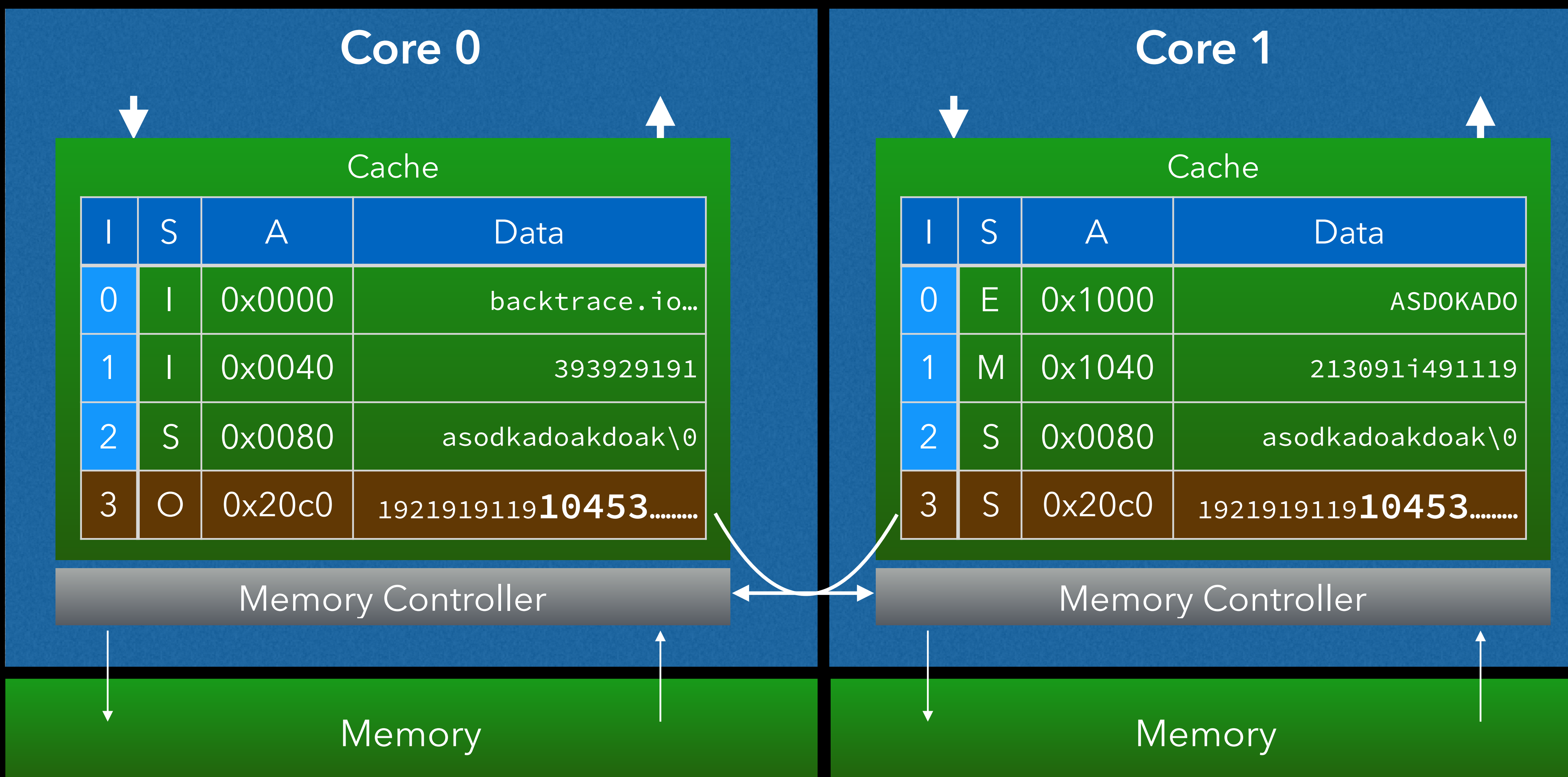
`Thread 1 loads x`

Thread 0

```
x = x + 10010;
```

Thread 1

```
printf("%d\n", x);
```



Core 0 Cache

I	S	A	Data
0	I	0x0000	backtrace.io…
1	I	0x0040	393929191
2	S	0x0080	asodkadoakdoak\0
3	O	0x20c0	1921919119**10453**………

Core 1 Cache

I	S	A	Data
0	E	0x1000	ASDOKADO
1	M	0x1040	213091i491119
2	S	0x0080	asodkadoakdoak\0
3	S	0x20c0	1921919119**10453**………

Backtrace

Cache Coherency

MESI, MOESI and MESIF are common cache coherency protocols.

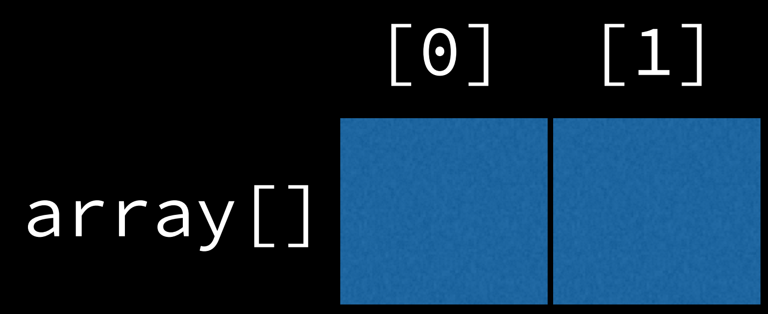
MESI: Modified, Exclusive, Shared, Invalid

MOESI: Modified, Owned, Exclusive, Shared, Invalid

MESIF: Modified, Exclusive, Shared, Forwarding

Cache Coherency

The **cache line** is the unit of coherency and can become an unnecessary source of contention.



Thread 0

```
for (;;) {
    array[0]++;
}
```

Thread 1

```
for (;;) {
    array[1]++;
}
```

Cache Coherency

False sharing occurs when logically disparate objects share the same cache line and contend on it.

```
struct {
    rwlock_t rwlock;
    int value;
} object;
```

Thread 0

```
for (;;) {
    read_lock(&object.rwlock);
    int v = atomic_read(&object.value);
    do_work(v);
    read_unlock(&object.rwlock);
}
```

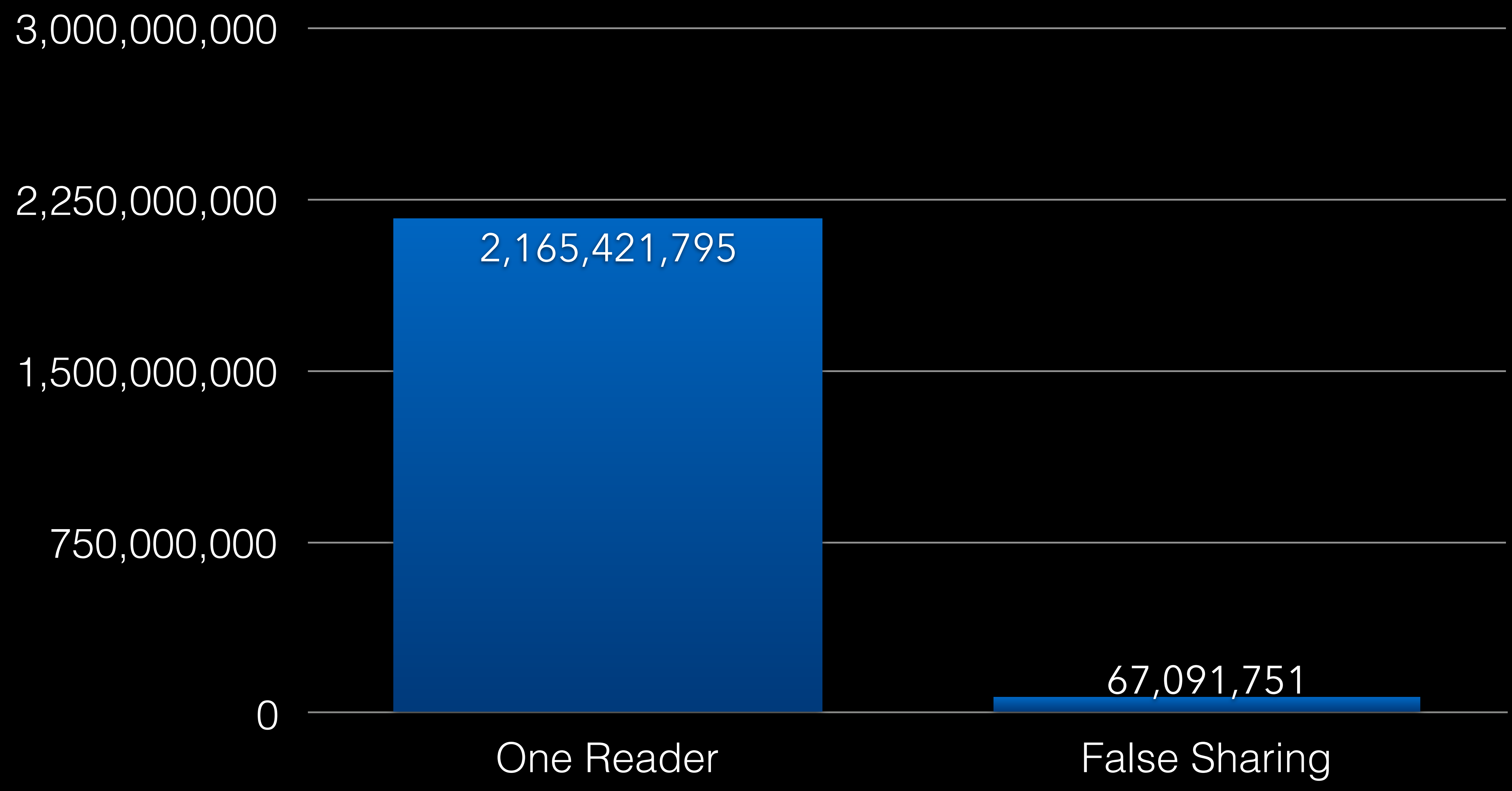
Thread 1

```
for (;;) {
    read_lock(&object.rwlock);
    <short work>
    read_unlock(&object.rwlock);
}
```

Backtrace

Cache Coherency

False sharing occurs when logically disparate objects share the same cache line and contend on it.



Backtrace

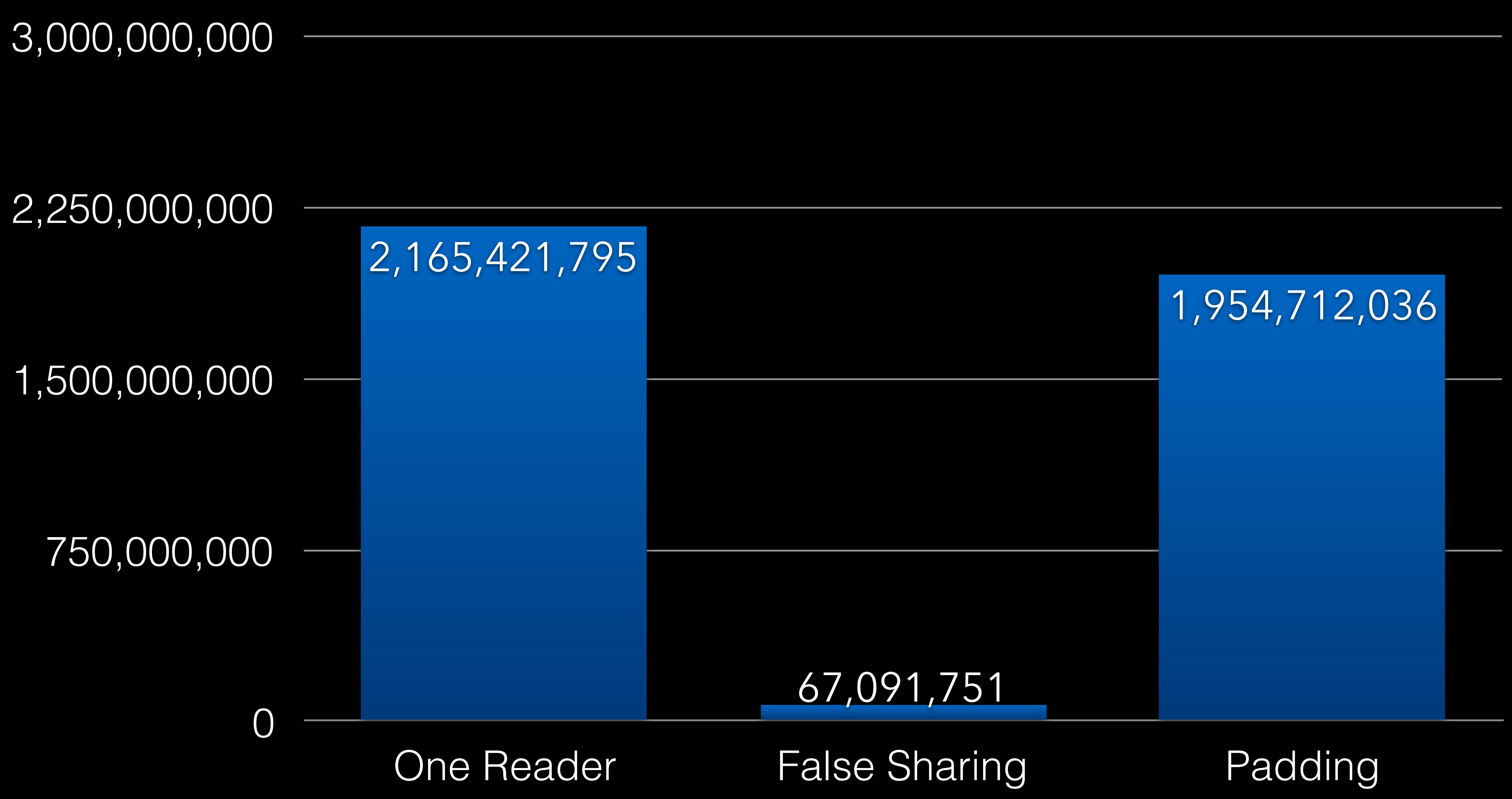
Cache Coherency

Padding can be used to mitigate false sharing.

```
struct {
    rwlock_t rwlock;
    char pad[64 - sizeof(rwlock_t)];
    int value;
} object;
```

Cache Coherency

Padding can be used to mitigate false sharing.



Backtrace

Cache Coherency

Padding must consider access patterns and overall footprint of application.

Too much padding is bad.

Simultaneous Multithreading

SMT technology allows for throughput increases by allowing programs to better utilize processor resources.

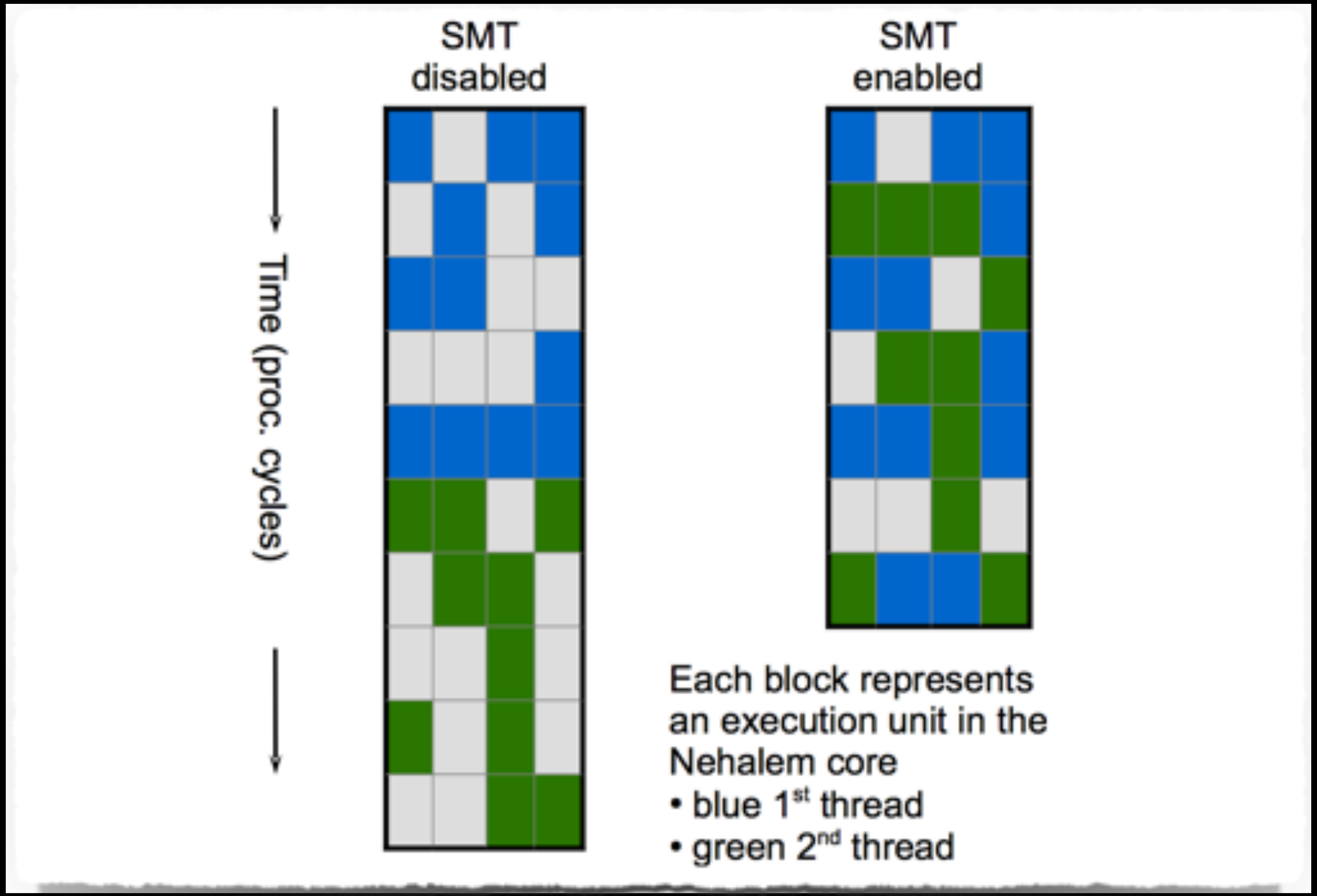


Figure from "The Architecture of the Nehalem Processor and Nehalem-EP SMP Platforms" (Michael E. Thomadakis)

Backtrace

Atomic Operations

Atomic operations are typically implemented with the help of the cache coherency mechanism.

Atomic Operations

Atomic operations are typically implemented with the help of the cache coherency mechanism.

```
lock cmpxchg(target, compare, new):
    register = load_and_lock(target);
    if (register == compare)
        store(target, new);
    unlock(target);
    return register;
```

Atomic Operations

Atomic operations are typically implemented with the help of the cache coherency mechanism.

```
lock cmpxchg(target, compare, new):
    register = load_and_lock(target);
    if (register == compare)
        store(target, new);
    unlock(target);
    return register;
```

Cache line locking typically only serializes accesses to the target cache line.

Atomic Operations

In the old commodity processor days, atomic operations were implemented with a bus lock.

```
lock cmpxchg(target, compare, new):
    lock(memory_bus);
    register = load(target);
    if (register == compare)
        store(target, new);
    unlock(memory_bus);
    return register;
```

x86 will assert a bus lock if an atomic operations goes across a cache line boundary. Be careful!

Atomic Operations

Atomic operations are crucial to efficient synchronization primitives.

Atomic Operations

Atomic operations are crucial to efficient synchronization primitives.

COMPARE_AND_SWAP(a, b, c): updates **a** to **c** if **a** is equal to **b**, atomically.

Atomic Operations

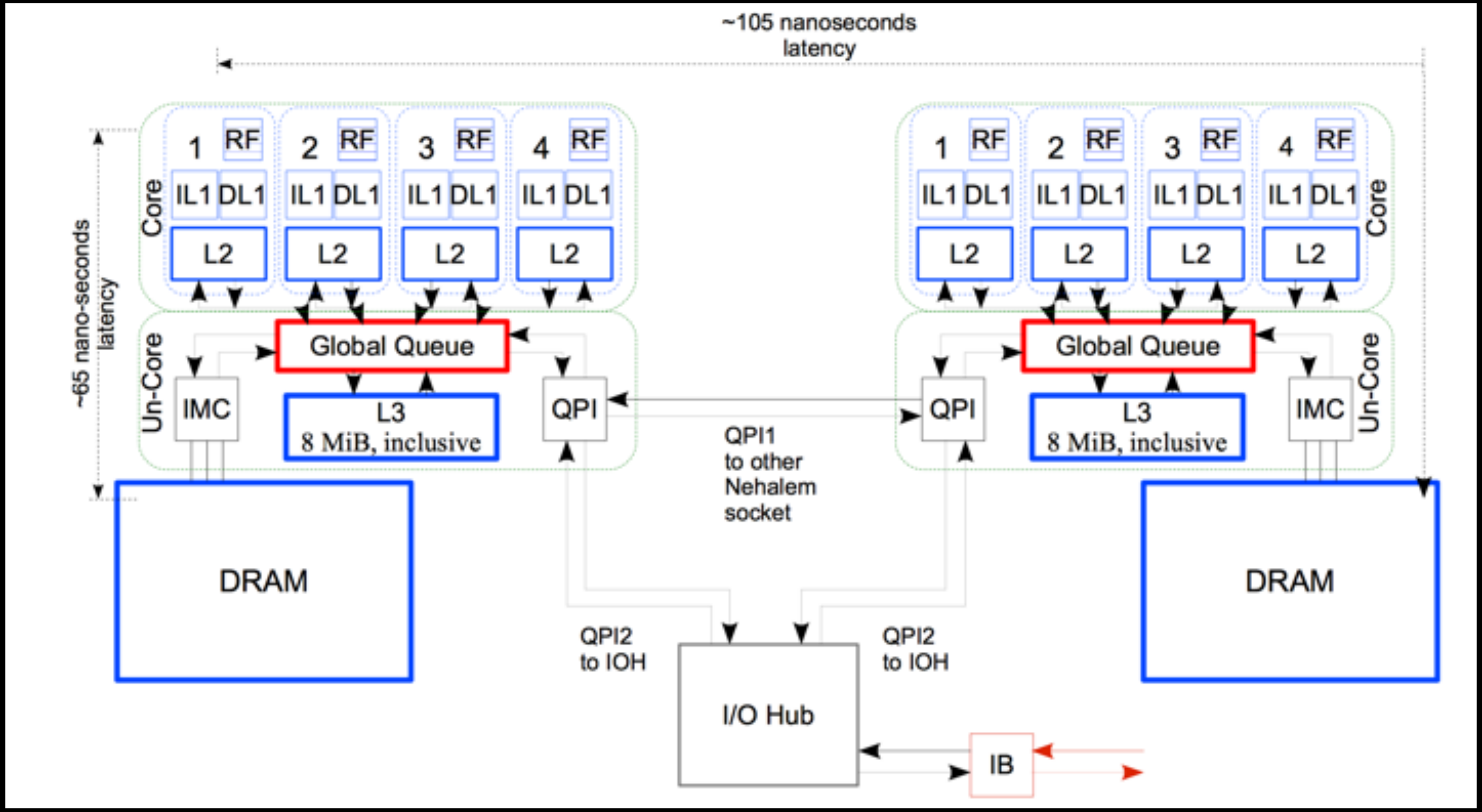
Atomic operations are crucial to efficient synchronization primitives.

COMPARE_AND_SWAP(a, b, c): updates **a** to **c** if **a** is equal to **b**, atomically.

LOAD_LINKED(a)/STORE_CONDITIONAL(a, b): Updates **a** to **b** if **a** was not modified between the load-linked (LL) and store-conditional (SC).

Topology

Most modern multicore systems are NUMA architectures: the throughput and latency of memory accesses varies.



Topology

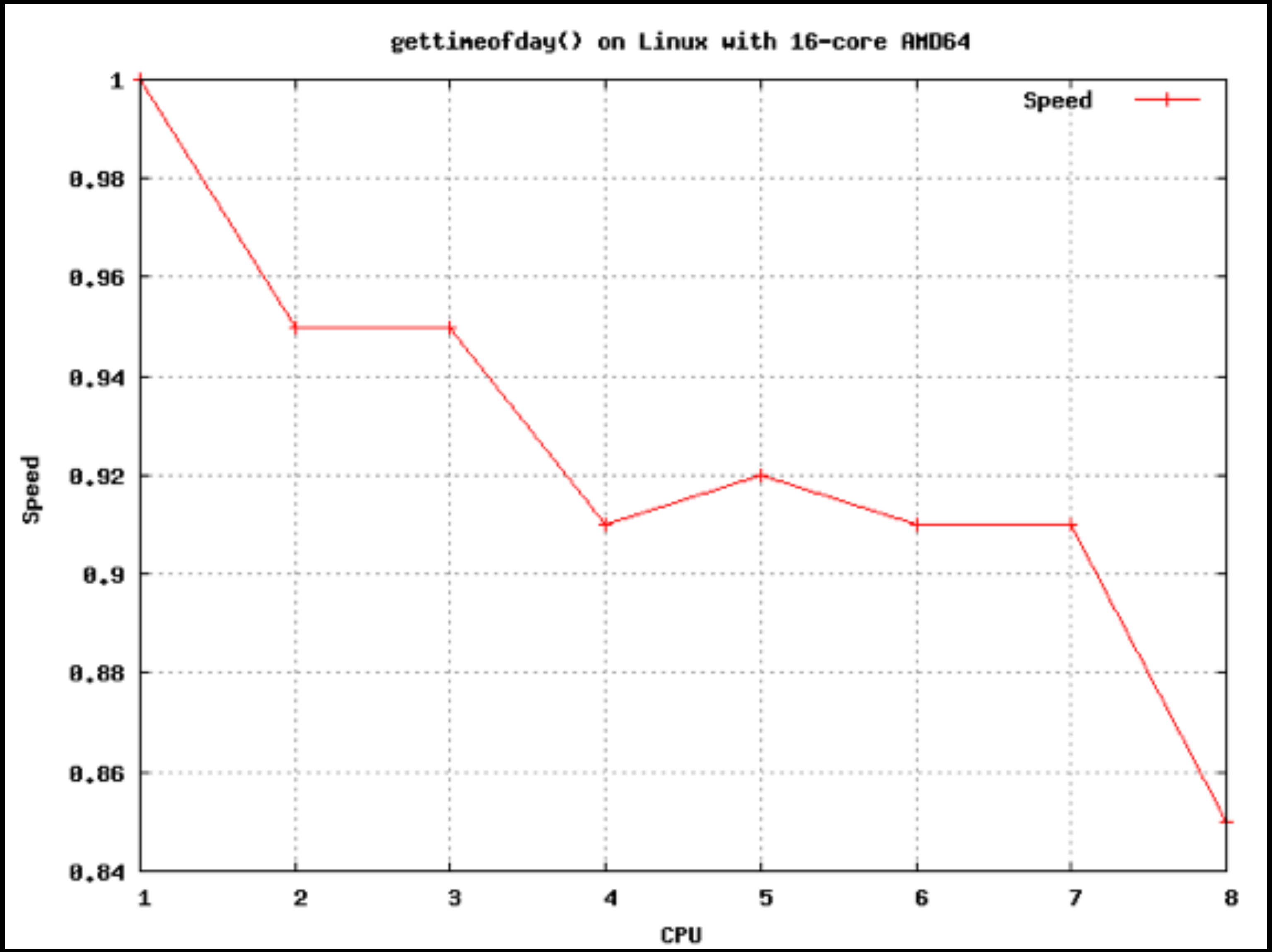
The NUMA factor is a ratio that represents the relative cost of a remote memory access.

	Time
Local wake-up	~140ns
Remote wake-up	~289ns

Intel Xeon L5640 machine at 2.27 GHz (12x2)

Topology

NUMA effects can be pervasive and difficult to mitigate.



Sun x4600

Topology

Be wary of your operating system's memory placement mechanisms.

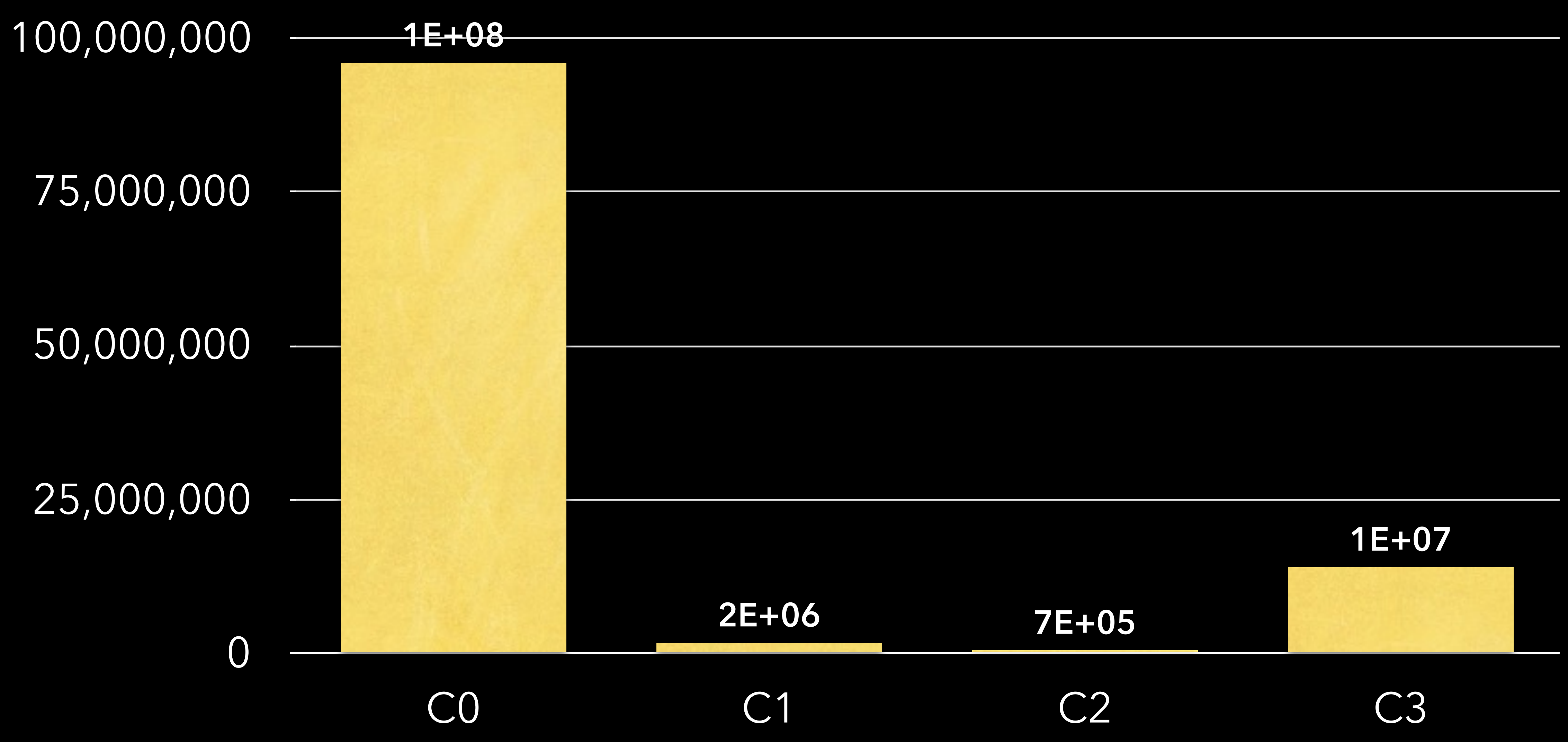
- First Touch: Allocate page on memory of first processor to touch it.

- Interleave: Allocate pages round-robin across nodes.

More sophisticated schemes exist that do hierarchical allocation, page migration, replication and more.

Topology

NUMA-oblivious synchronization objects are not only susceptible to performance mismatch but starvation and even livelock under extreme load.



Intel(R) Xeon(R) CPU E7- 4850 @ 2.00GHz @ 10x4

Backtrace

Fairness

Fair locks guarantee starvation-freedom.

Next	Position
0	0

```
CK_CC_INLINE static void
ck_spinlock_ticket_lock(struct ck_spinlock_ticket *ticket)
{
        unsigned int request;

        request = ck_pr_faa_uint(&ticket->next, 1);

        while (ck_pr_load_uint(&ticket->position) != request)
                ck_pr_stall();

        return;
}
```

Fairness

request = 0

```
CK_CC_INLINE static void
ck_spinlock_ticket_lock(struct ck_spinlock_ticket *ticket)
{
        unsigned int request;

        request = ck_pr_faa_uint(&ticket->next, 1);

        while (ck_pr_load_uint(&ticket->position) != request)
                ck_pr_stall();

        return;
}
```

Fairness

Next	Position
1	0

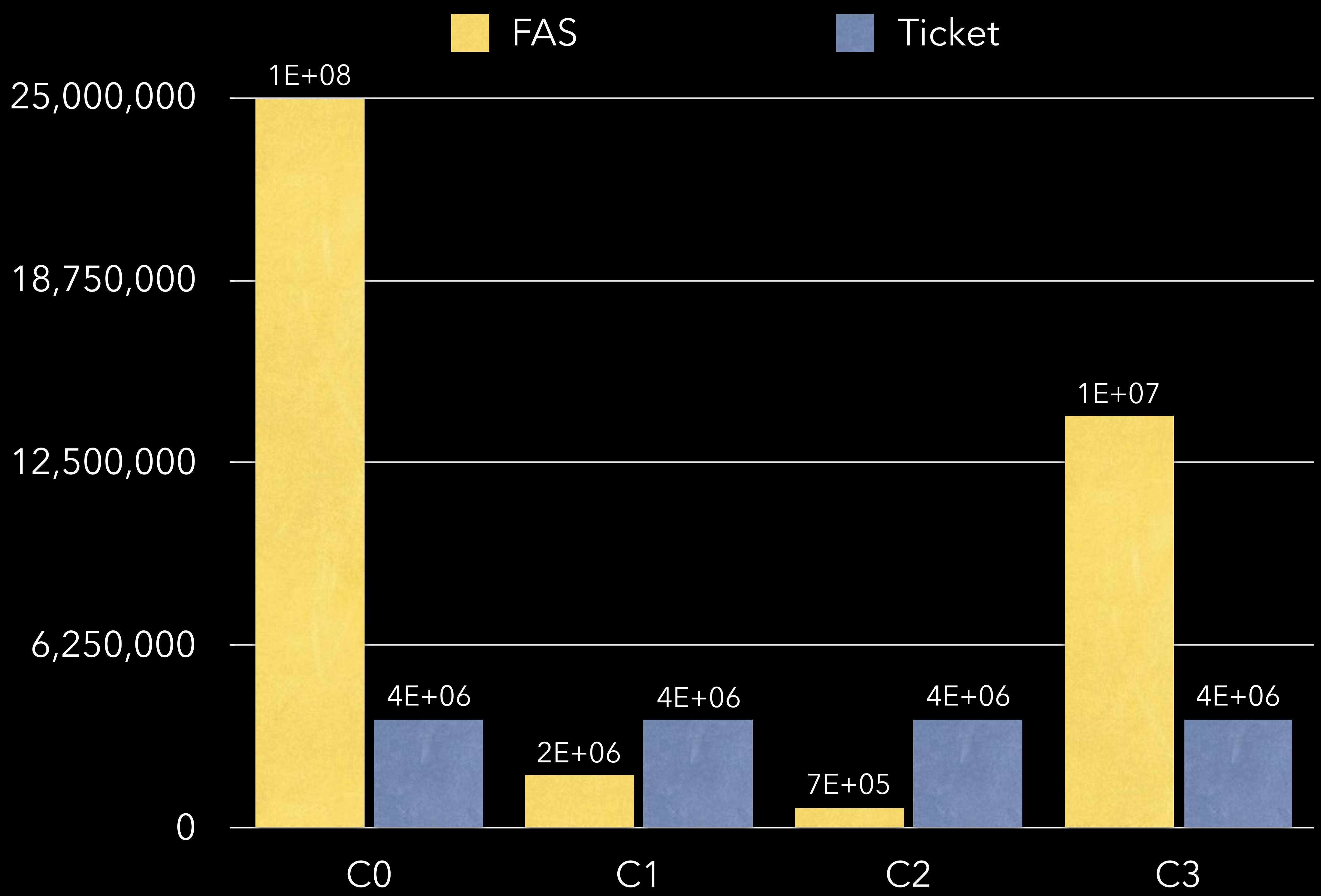
```
request = 0
```

```
CK_CC_INLINE static void
ck_spinlock_ticket_lock(struct ck_spinlock_ticket *ticket)
{
        unsigned int request;

        request = ck_pr_faa_uint(&ticket->next, 1);

        while (ck_pr_load_uint(&ticket->position) != request)
                ck_pr_stall();

        return;
}
```

Fairness


```
CK_CC_INLINE static void
ck_spinlock_ticket_unlock(struct ck_spinlock_ticket *ticket)
{
        unsigned int update;

        update = ck_pr_load_uint(&ticket->position);
        ck_pr_store_uint(&ticket->position, update + 1);
        return;
}
```

Fairness



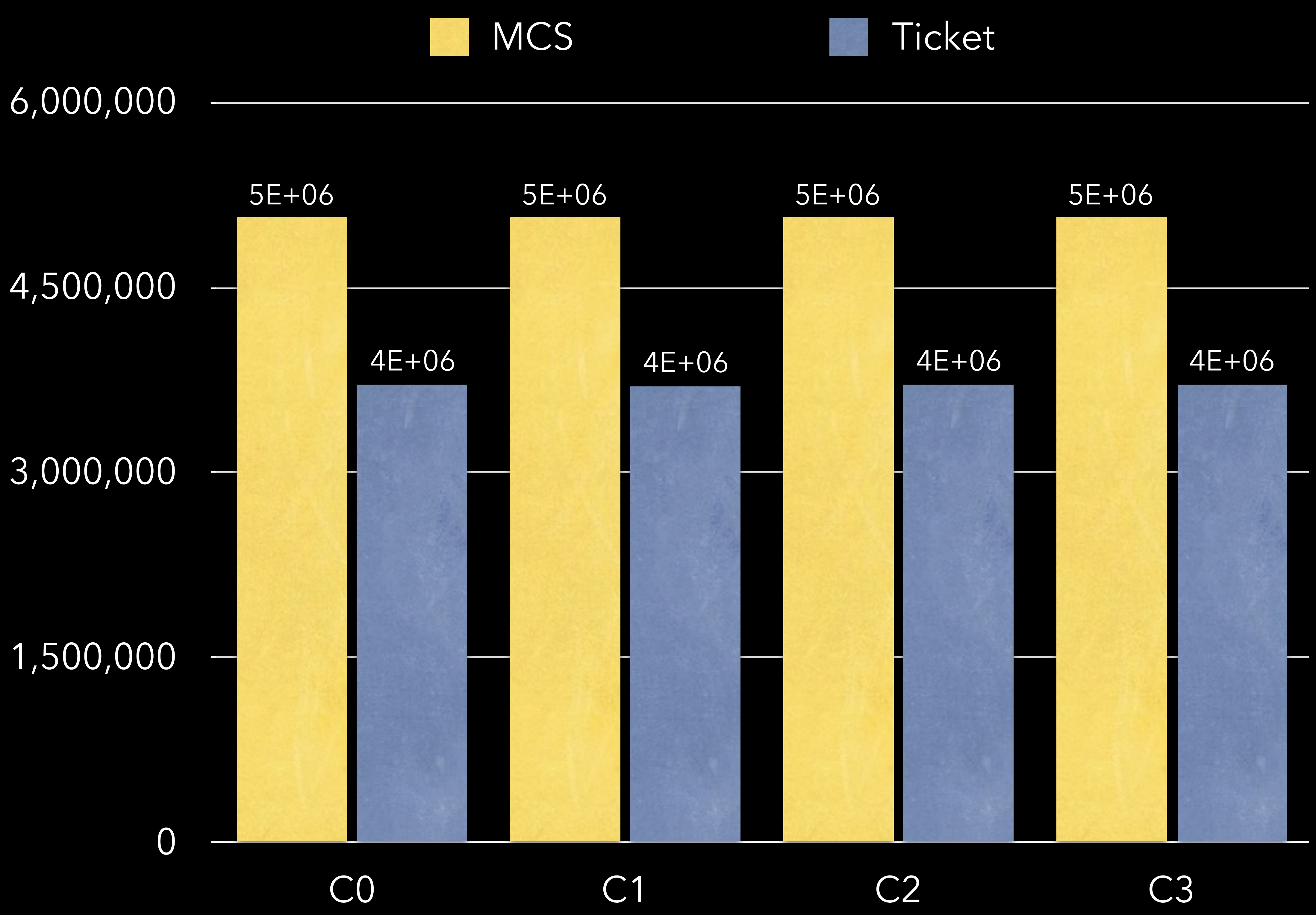
Intel(R) Xeon(R) CPU E7- 4850 @ 2.00GHz @ 10x4

Fairness

Fair locks are not a silver bullet and may negatively impact throughput.

Fairness comes at the cost of increased sensitivity to preemption and other sources of jitter.

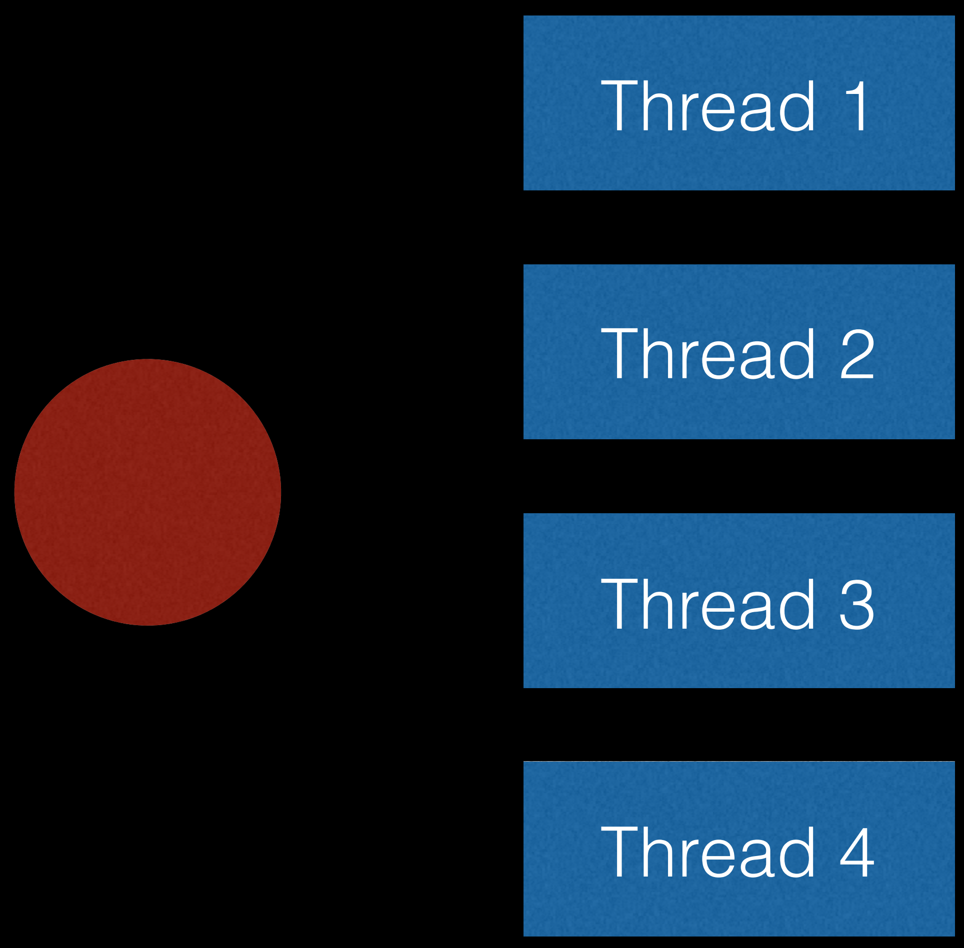
Fairness



Intel(R) Xeon(R) CPU E7- 4850 @ 2.00GHz @ 10x4

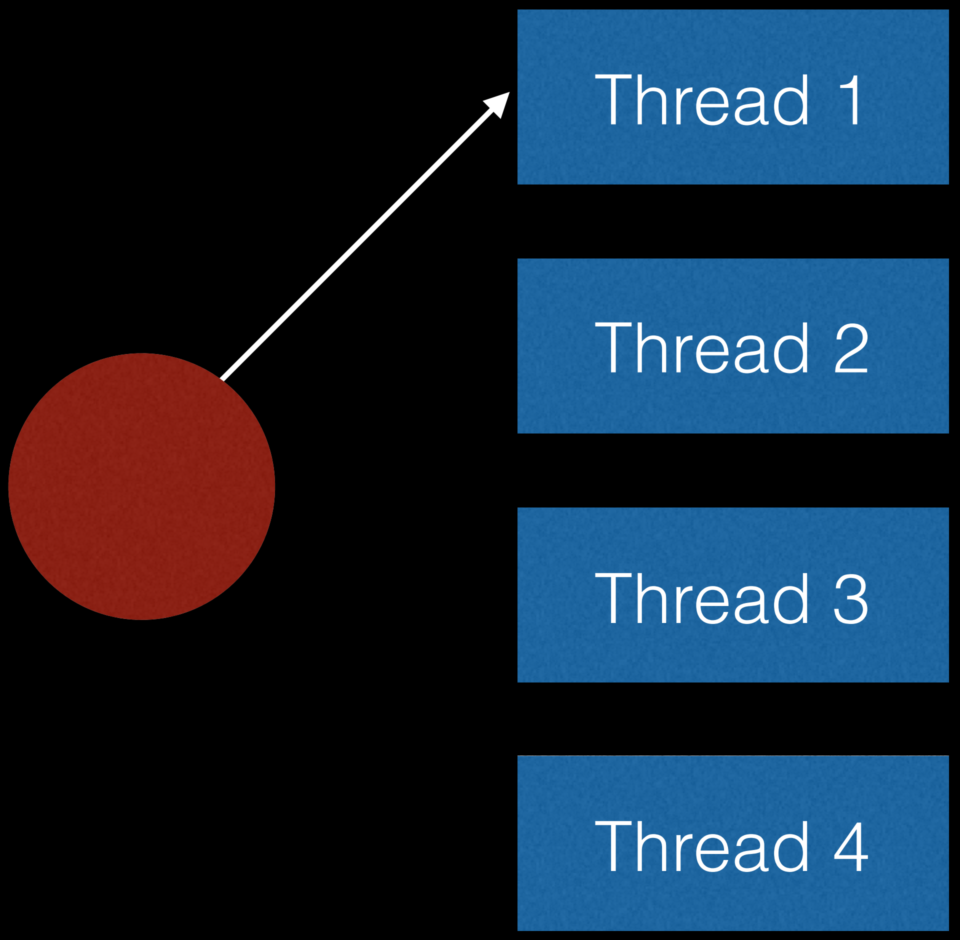
Distributed Locks

Array and queue-based locks provide lock scalability and fairness with distributing spinning and point-to-point wake-up.



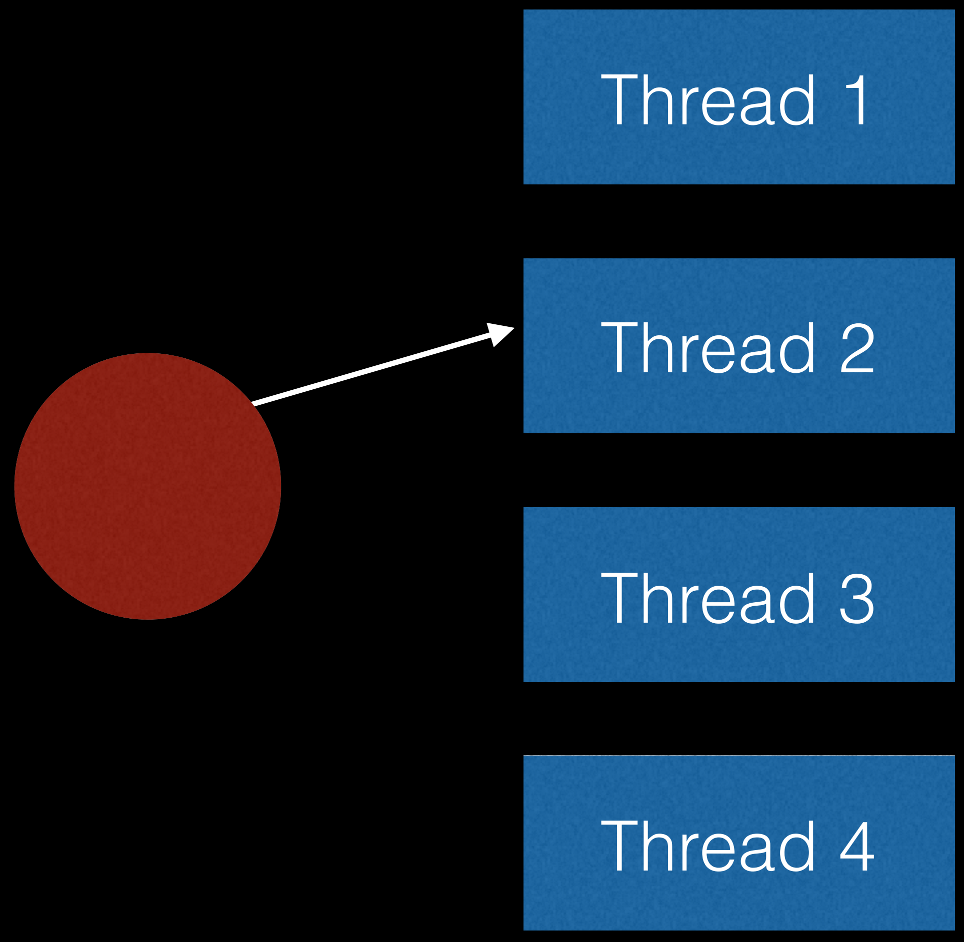
Distributed Locks

The **MCS** lock was a seminal contribution to the area and introduced queue locks to the masses.



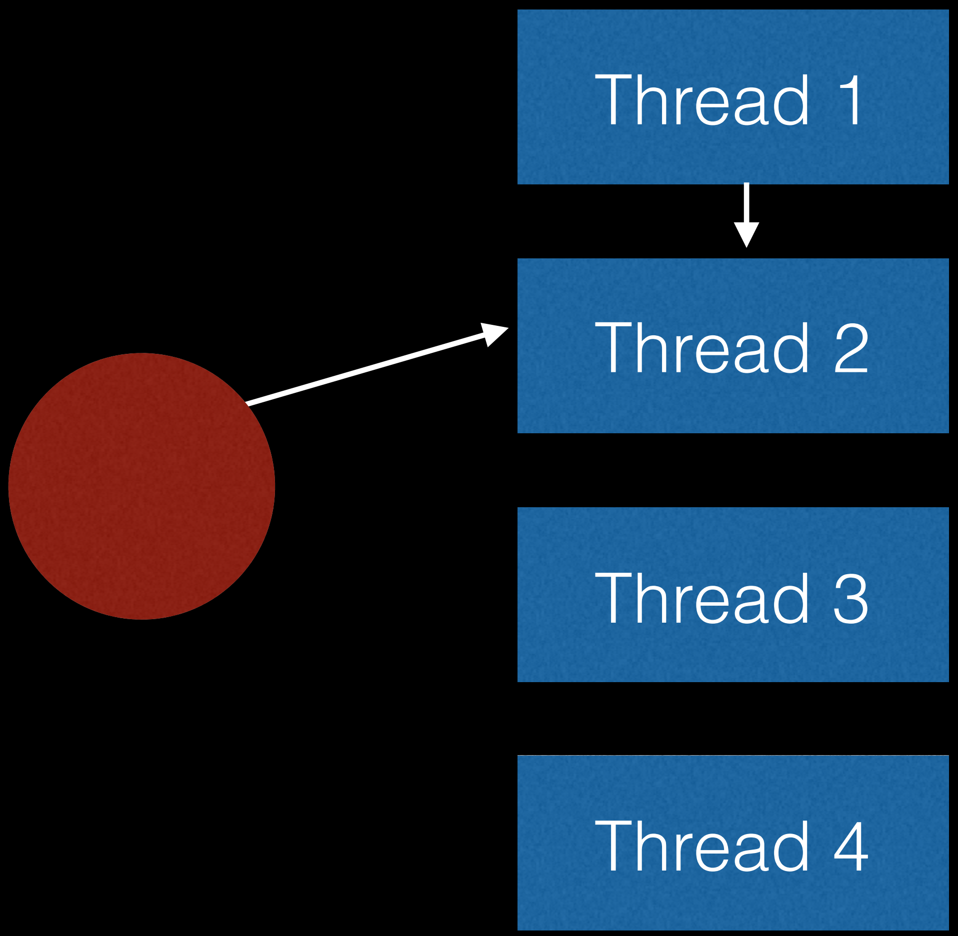
Backtrace

Distributed Locks



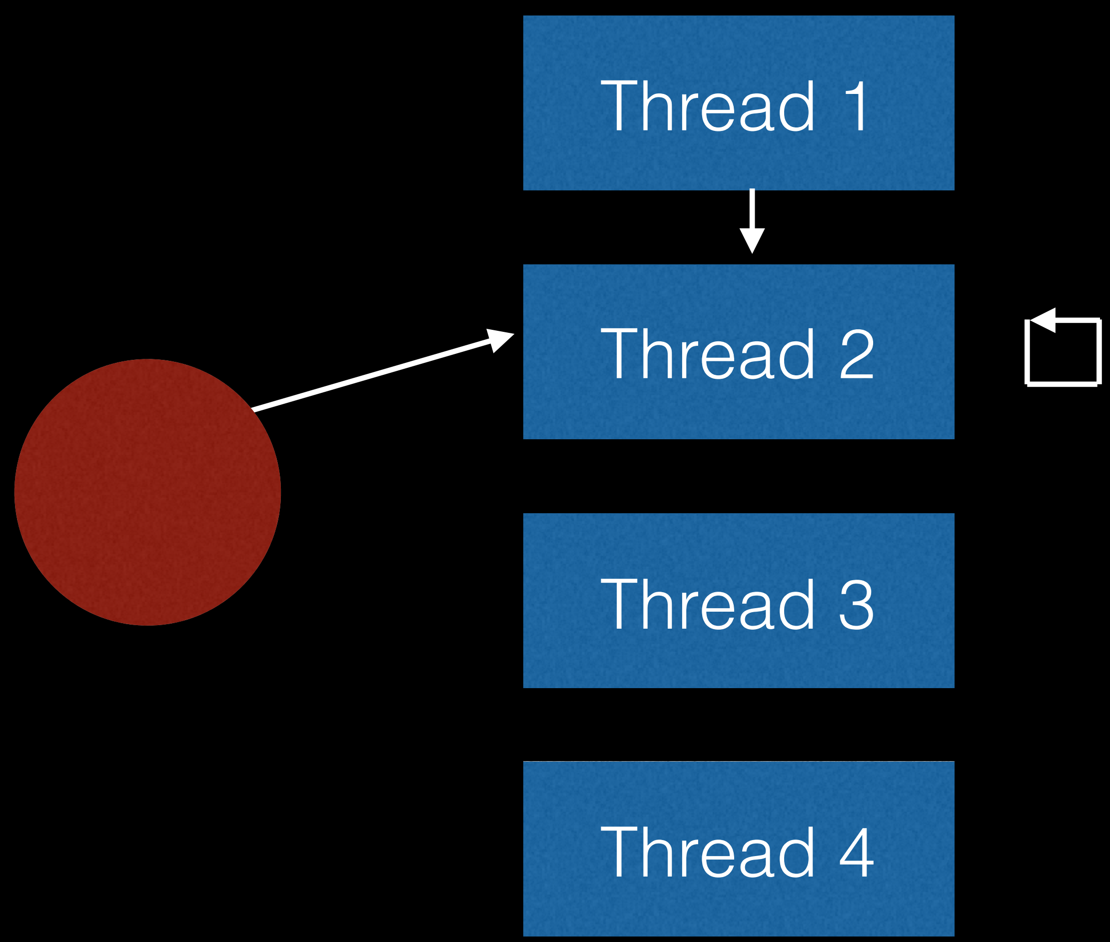
Backtrace

Distributed Locks



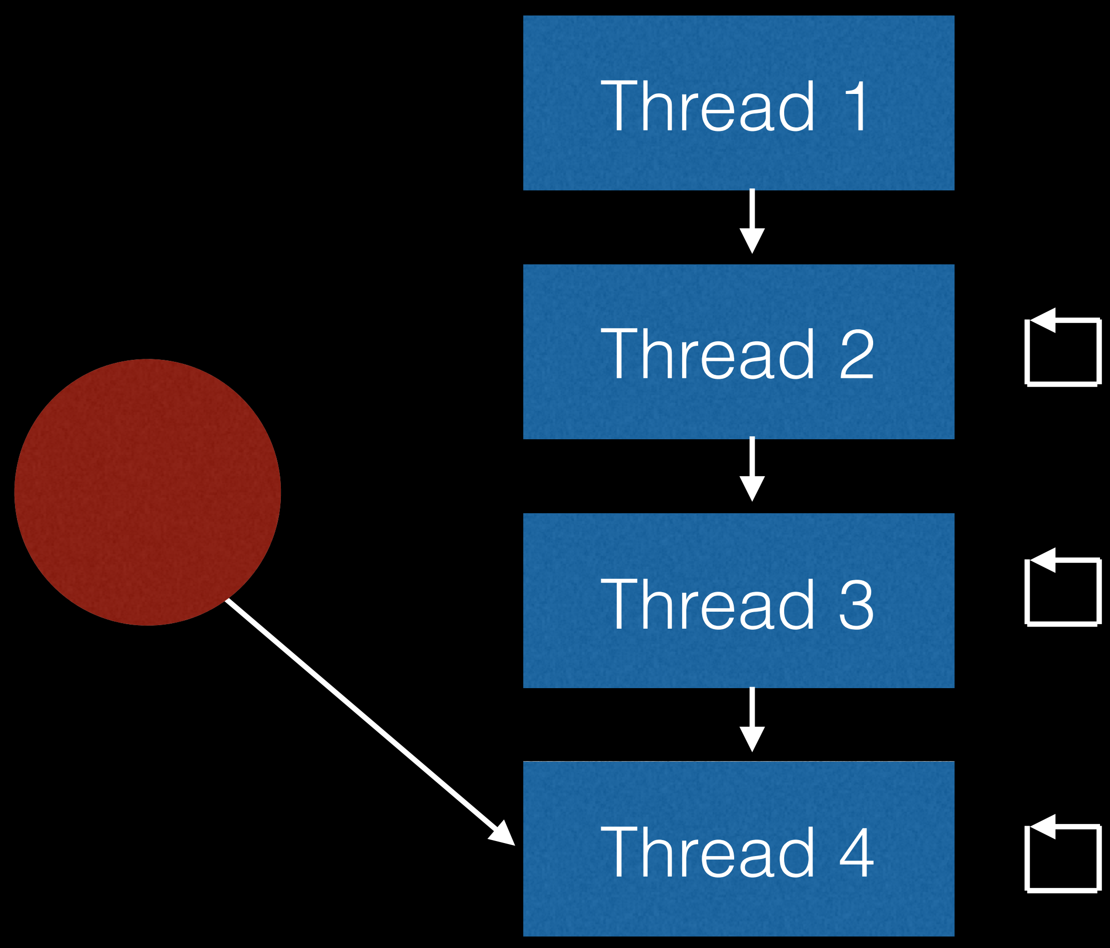
Backtrace

Distributed Locks

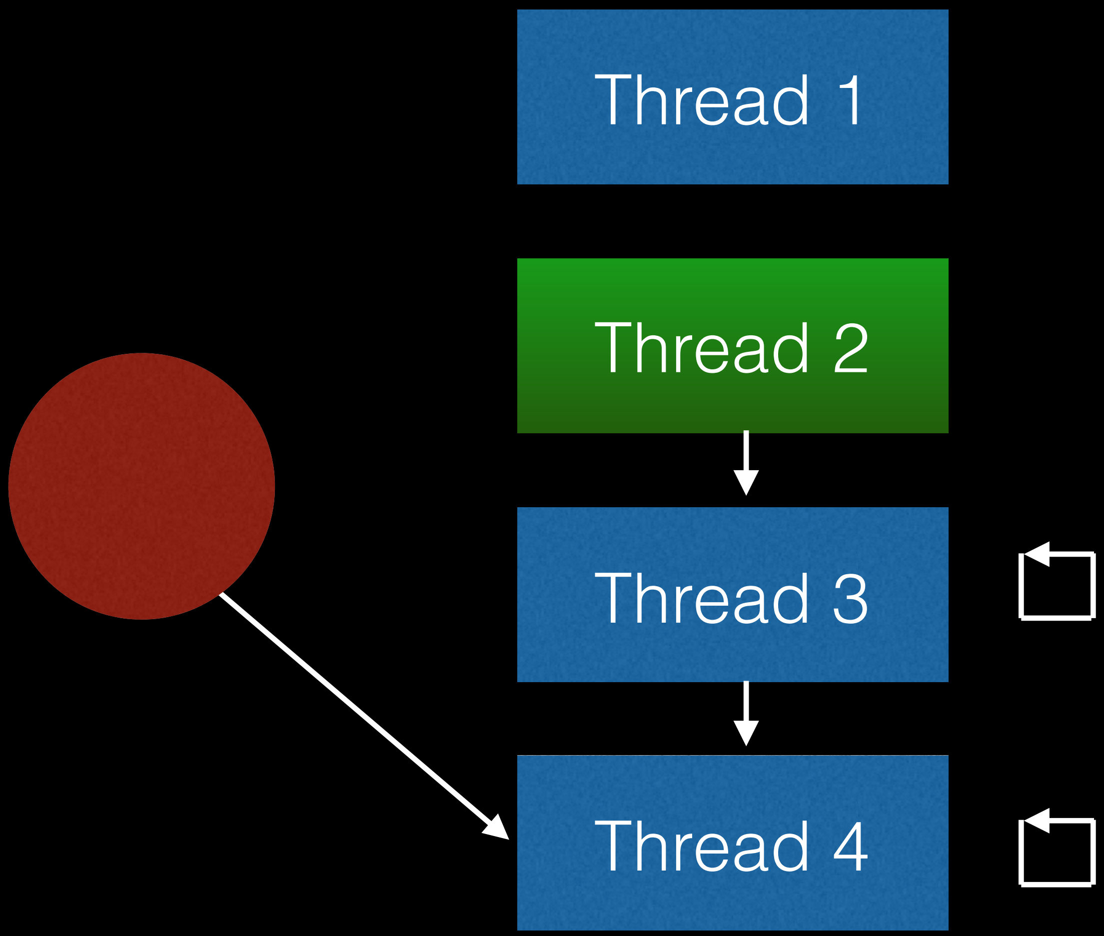


Backtrace

Distributed Locks

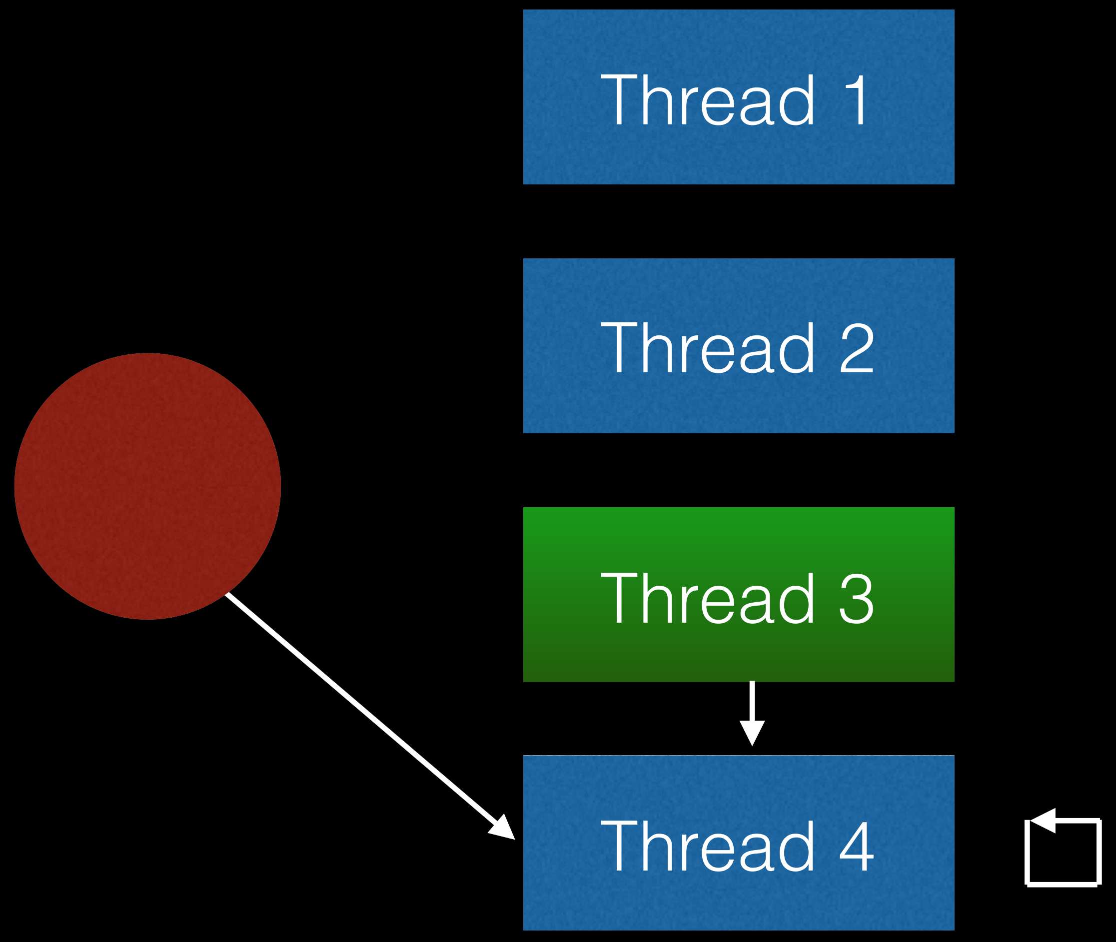


Distributed Locks

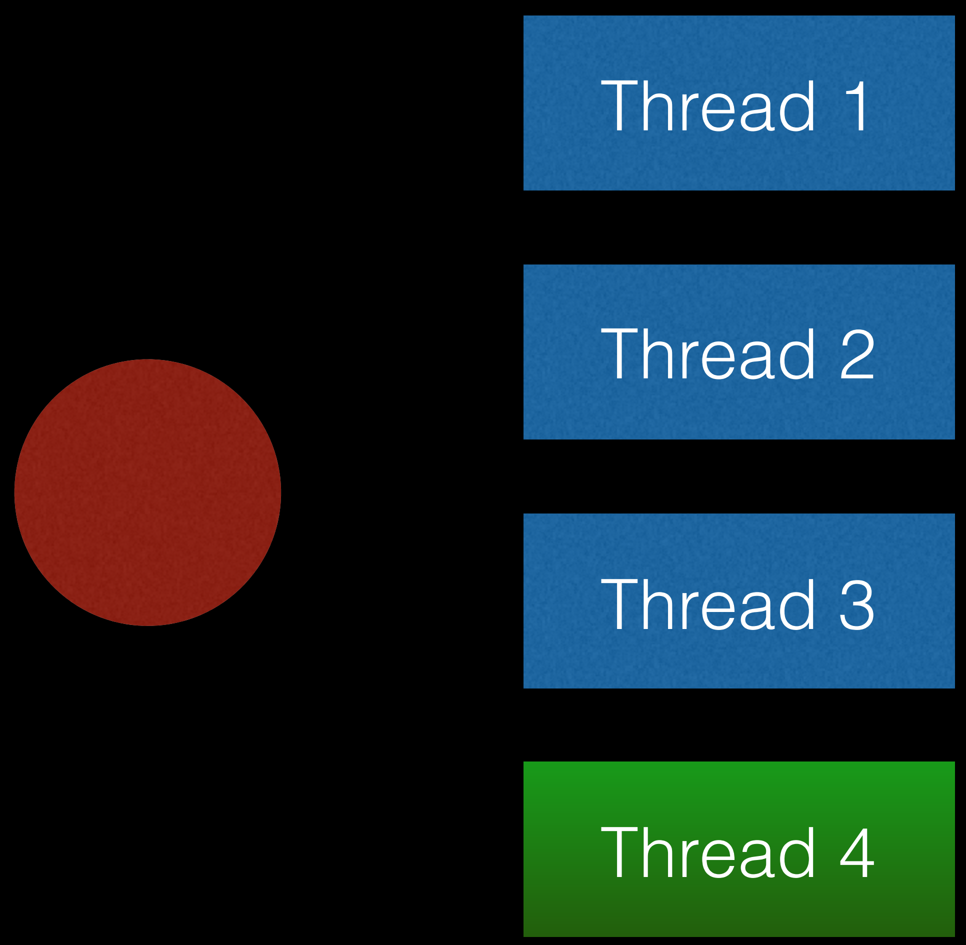


Backtrace

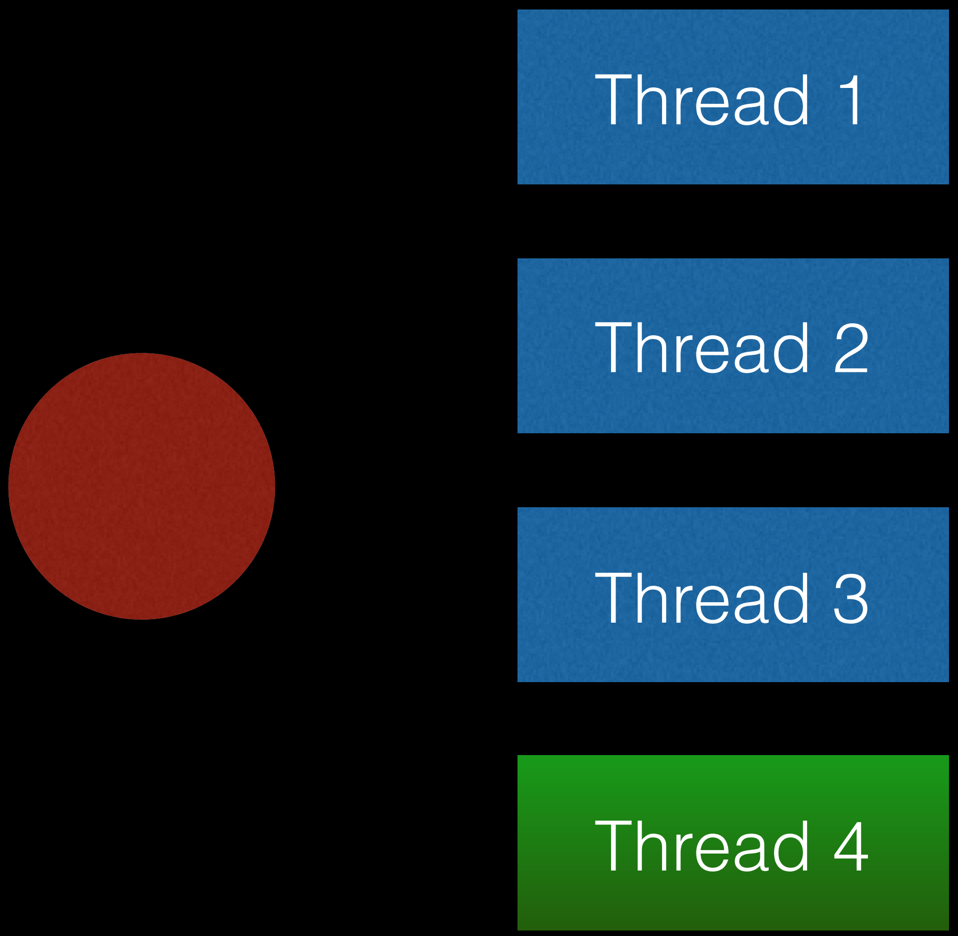
Distributed Locks



Distributed Locks

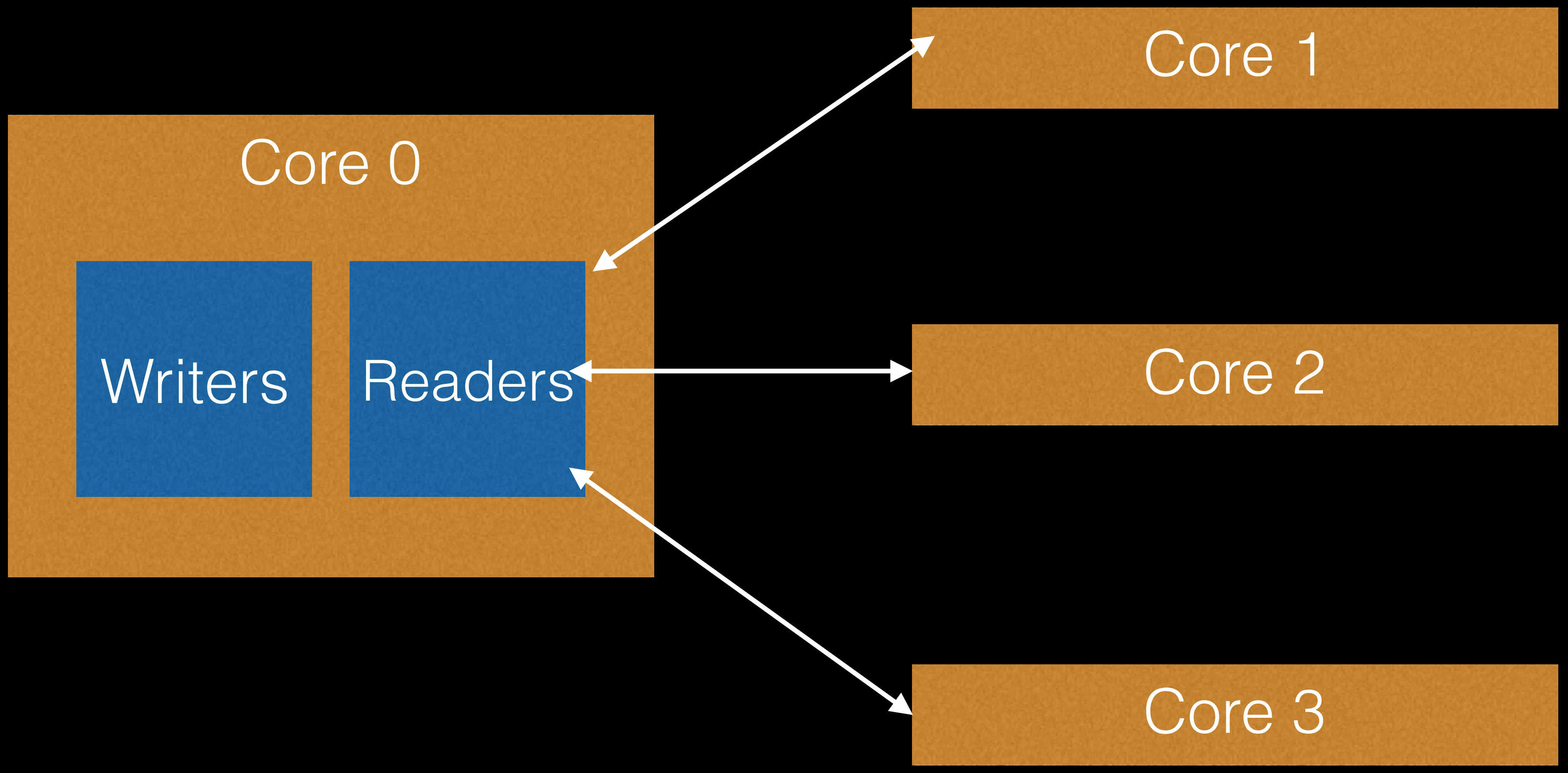


Distributed Locks



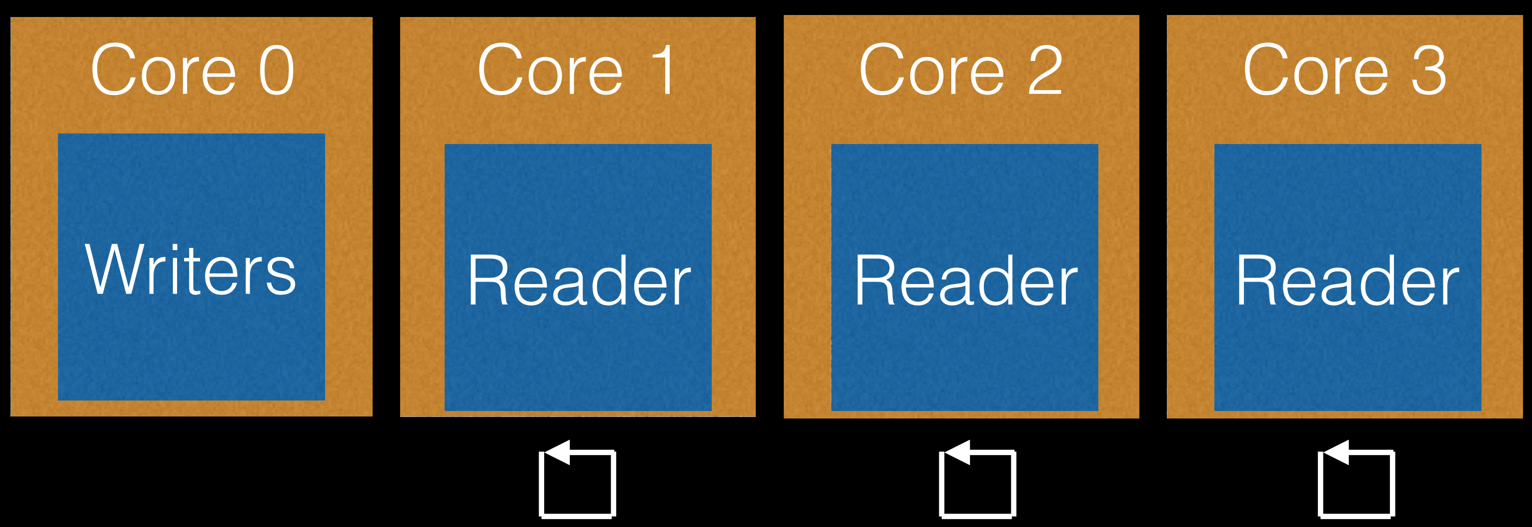
Distributed Locks

Similar mechanisms exist for read-write locks.

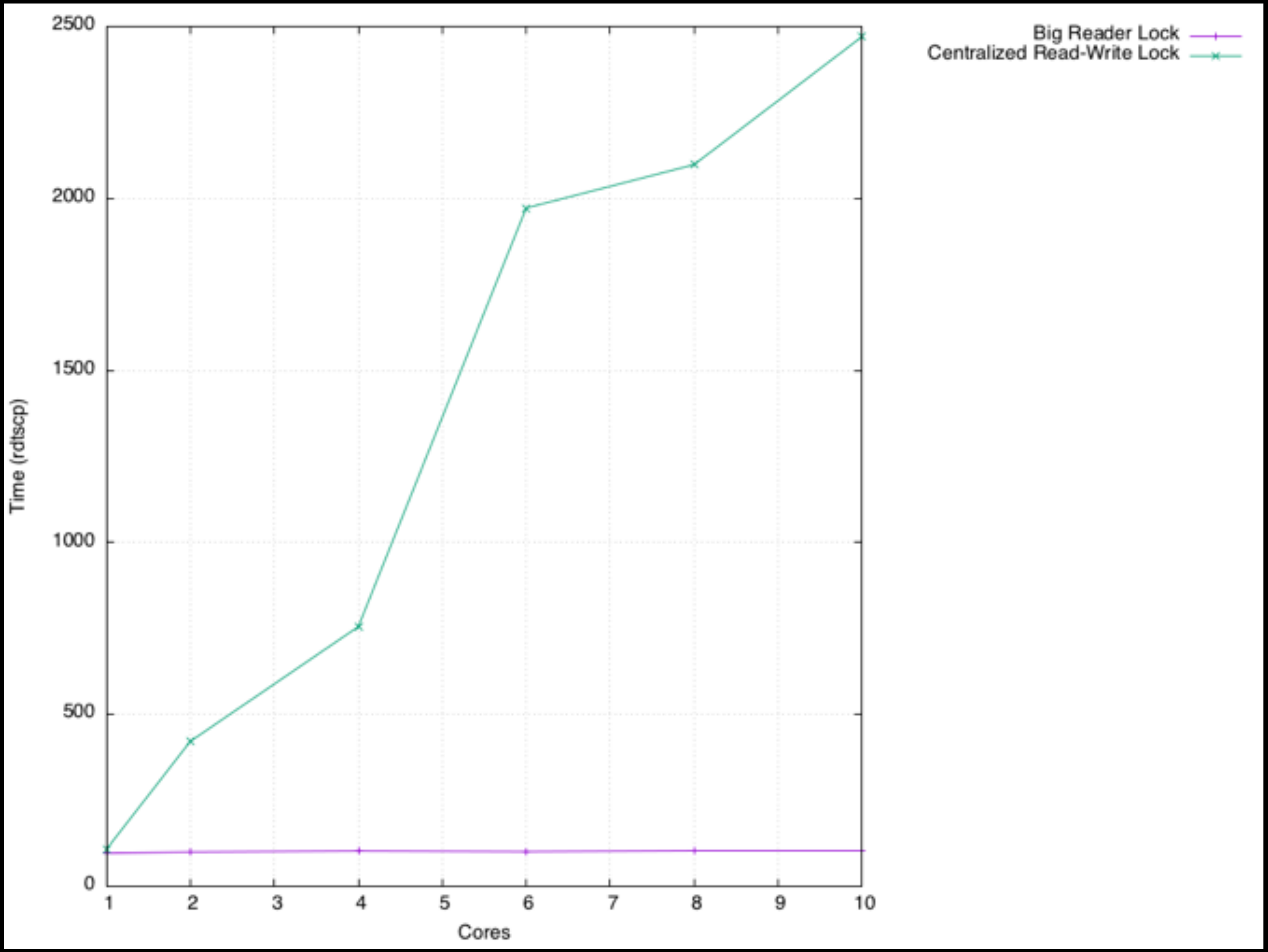


Distributed Locks

Big reader locks (brlocks) or Read-Mostly Locks (rmlocks) distribute read-side flags so that readers spin only on local memory.



Distributed Locks



Intel Xeon E5-2630L at 2.40 GHz

Limitations of Locks

Locks are not composable and are susceptible to priority inversion, livelock, starvation, deadlock and more.

Limitations of Locks

Locks are not composable and are susceptible to priority inversion, livelock, starvation, deadlock and more.

A delicate balance must be found between lock hierarchies, granularity and quality of service.

Limitations of Locks

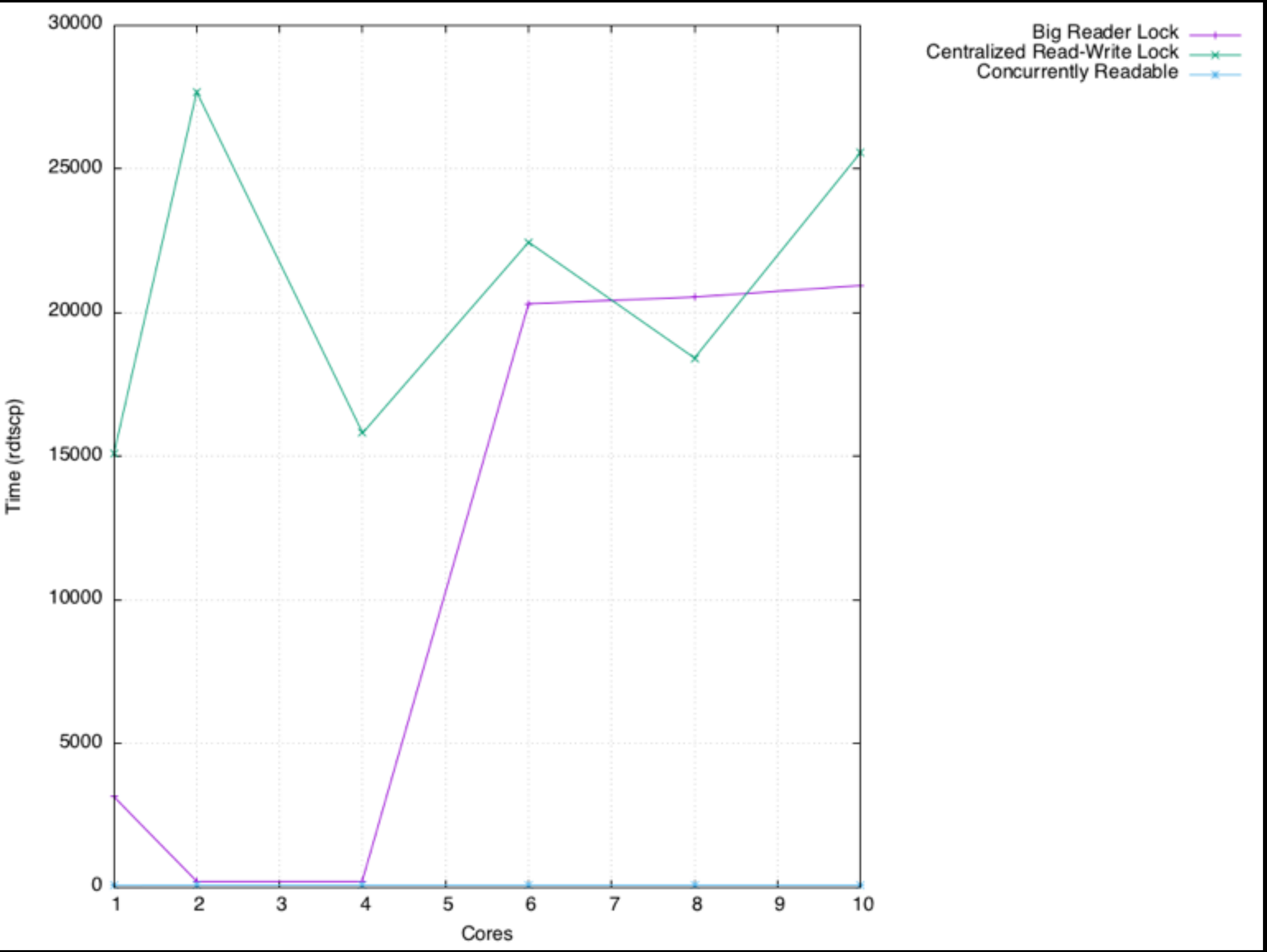
Locks are not composable and are susceptible to priority inversion, livelock, starvation, deadlock and more.

A delicate balance must be found between lock hierarchies, granularity and quality of service.

A significant delay in one thread holding a synchronization object leads to significant delays for all other threads waiting on the same synchronization object.

Backtrace

Limitations of Locks



Intel Xeon E5-2630L at 2.40 GHz

Lock-less Synchronization

Backtrace

Lock-less Synchronization

With lock-based synchronization, it is sufficient to reason in terms of lock dependencies and critical sections.

Lock-less Synchronization

With lock-based synchronization, it is sufficient to reason in terms of lock dependencies and critical sections.

This model doesn't work with lock-less synchronization where we must guarantee correctness in much more subtle ways.

Memory Models

These days, cache coherency helps implement the consistency model.

The memory model is specified by the runtime environment and defines the correct behavior of shared memory accesses.

Memory Models

```
int x = 0;
int y = 0;
```

Core 0

```
int r_0;

x = 1;
r_0 = y;
```

Core 1

```
int r_1;

y = 1;
r_1 = x;
```

```
if (r_0 == 0 && r_1 == 0)
    abort();
```

Memory Models

```
int x = 0;
int y = 0;
```

Core 0

```
int r_0;

x = 1;
-----------
r_0 = y;
```

Core 1

```
int r_1;

y = 1;
-----------
r_1 = x;
```

```
(r_0, r_1) = (1,1)
```

Backtrace

Memory Models

```
int x = 0;
int y = 0;
```

Core 0

```
int r_0;

x = 1;
r_0 = y;
```

Core 1

```
int r_1;

y = 1;
r_1 = x;
```

```
(r_0, r_1) = (0,1)
```

Backtrace

Memory Models

```
int x = 0;
int y = 0;
```

Core 1

```
int r_1;

y = 1;
r_1 = x;
```

Core 0

```
int r_0;

x = 1;
r_0 = y;
```

```
(r_0, r_1) = (1,0)
```

Memory Models

This condition is possible and is an example of **store-to-load** re-ordering.

Core 0

```
int r_0;

r_0 = y;
x = 1;
```

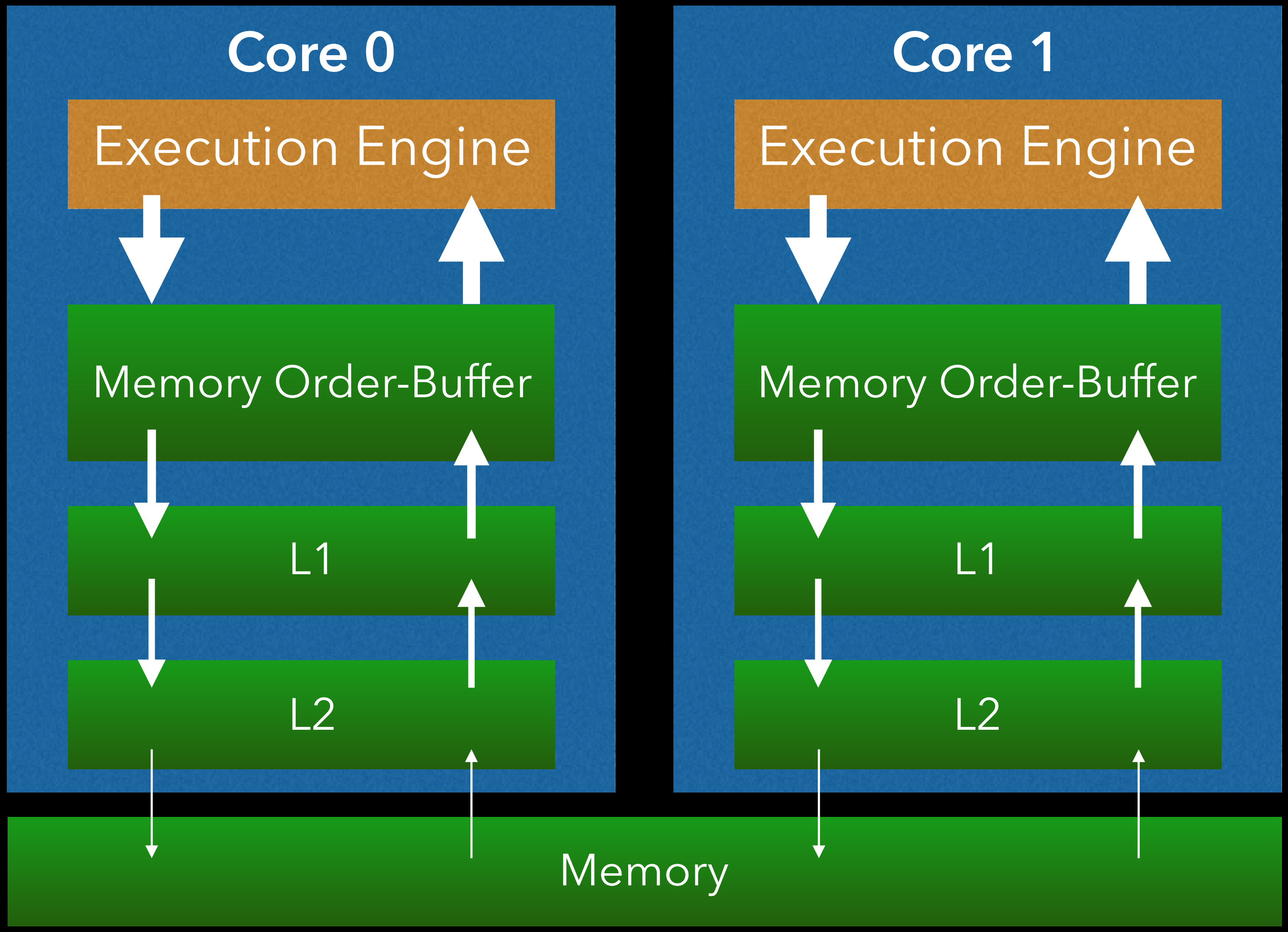
Core 1

```
int r_1;

r_1 = x;
y = 1;
```

```
(r_0, r_1) = (0,0)
```

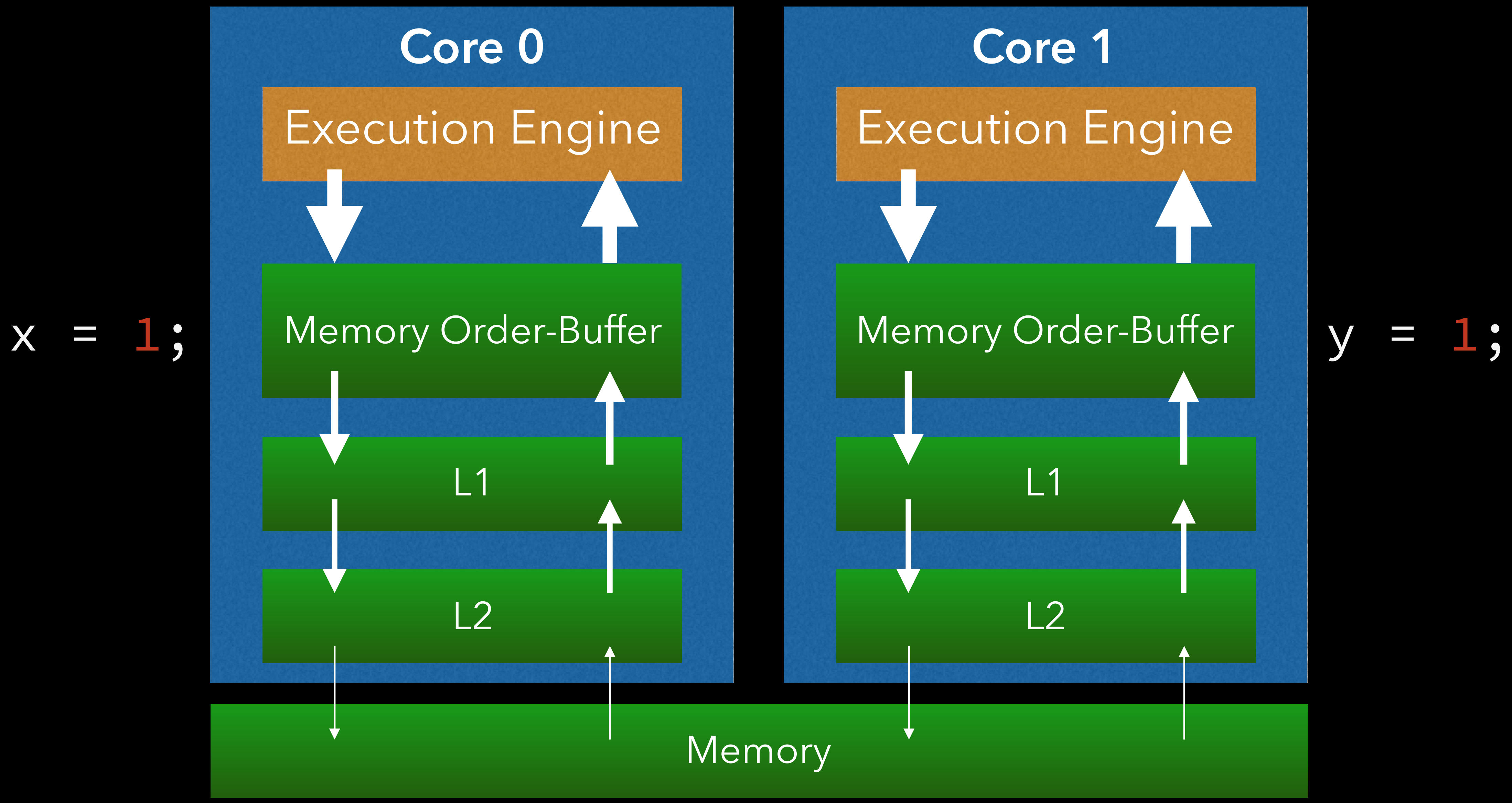
Memory Models

Modern processors rely on a myriad of techniques to achieve high levels of instruction-level parallelism.



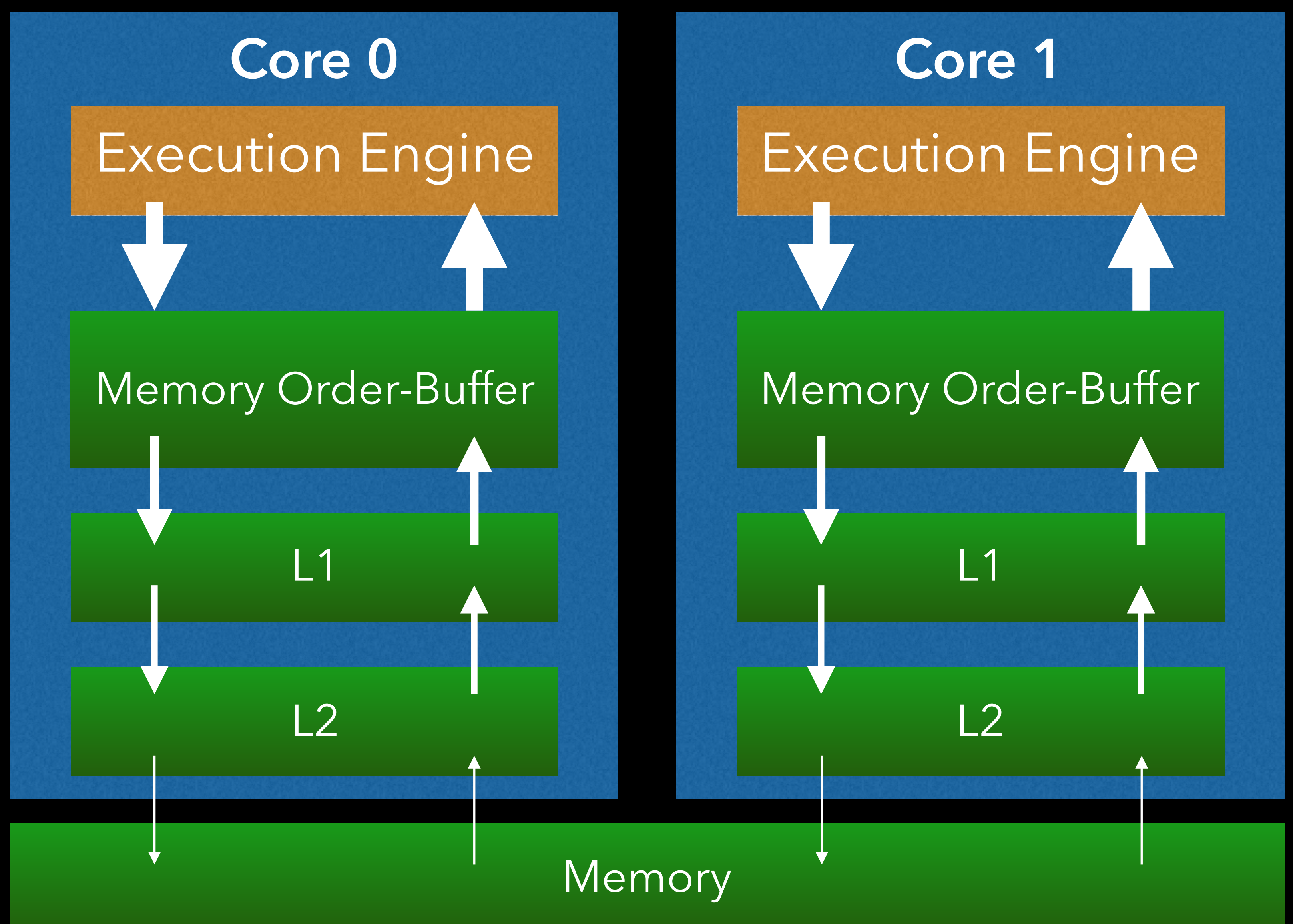
Memory Models

Modern processors rely on a myriad of techniques to achieve high levels of instruction-level parallelism.



Memory Models

Modern processors rely on a myriad of techniques to achieve high levels of instruction-level parallelism.




```
x = 1;
r_0 = y;
```

```
y = 1;
r_1 = x;
```

Memory Models

The processor memory model is specified with respect to loads, stores and atomic operations.

	TSO	RMO
Load to Load		
Load to Store		
Store to Store		
Store to Load		
Atomics		
Examples	x86*, SPARC-TSO	ARM, Power

Memory Models

Ordering guarantees are provided by serializing instructions such as memory fences.

?

```
mutex_lock(&mutex);
x = x + 1;
mutex_unlock(&mutex);
```

Backtrace

Memory Models

Ordering guarantees are provided by serializing instructions such as memory fences.

?

```
mutex_lock(&mutex);
x = x + 1;
mutex_unlock(&mutex);
```

```
CK_CC_INLINE static void
mutex_lock(struct mutex *lock)
{

        while (ck_pr_fas_uint(&lock->value, true) == true);
        ck_pr_fence_memory();
        return;
}
```

* Simplified

Memory Models

Serializing instructions are expensive because they disable some processor optimizations.

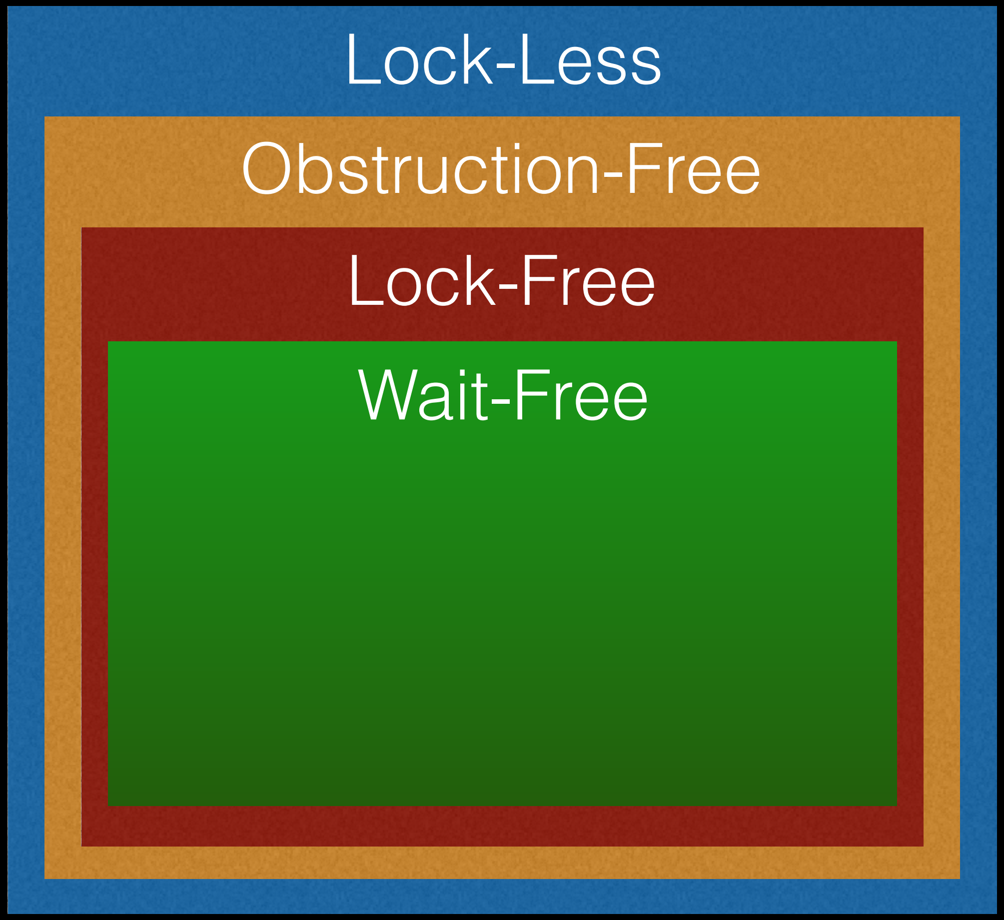
Atomic instructions are expensive because they either involve serialization (and locking) or are just plain old complex.

	Throughput (/ second)
`lock cmpxchg`	147,304,564
`cmpxchg`	458,940,006

Intel Core i7-3615QM at 2.30 GHz

Lock-less Synchronization

Non-blocking synchronization provides very specific progress guarantees and high levels of resilience at the cost of complexity on the fast path.



Lock-freedom provides system-wide progress guarantees.

Wait-freedom provides per-operation progress guarantees.

Lock-less Synchronization

```c
struct node {
   void *value;
   struct node *next;
};

void
stack_push(struct node **top, struct node *entry, void *value)
{

   entry->value = value;
   entry->next = *top;
   *top = entry;
   return;
}

struct node *
stack_pop(struct node **top)
{
   struct node *r;

   r = *top;
   *top = r->next;
   return r;
}
```

Lock-less Synchronization

```
struct node {
    void *value;
    struct node *next;
};

void
stack_push(struct node **top, struct node *entry,
     void *value)
{

    entry->value = value;

    do {
        entry->next = ck_pr_load_ptr(top);
    } while (ck_pr_cas_ptr(top, entry->next,
          entry) == false);

    return;
}
```

Lock-less Synchronization

```
struct node {
    void *value;
    struct node *next;
};

void
stack_push(struct node **top, struct node *entry,
     void *value)
{

    entry->value = value;

    do {
        entry->next = ck_pr_load_ptr(top);
    } while (ck_pr_cas_ptr(top, entry->next,
          entry) == false);

    return;
}
```

```
struct node *
stack_pop(struct node **top)
{
    struct node *r, *next;

    do {
        r = ck_pr_load_ptr(top);
        if (r == NULL)
            return NULL;

        next = ck_pr_load_ptr(&r->next);
    } while (ck_pr_cas_ptr(top, r, next) ==
        false);

    return r;
}
```

Backtrace

Lock-less Synchronization

Non-blocking synchronization
is not a silver bullet.

Lock-less Synchronization

Non-blocking synchronization
is not a silver bullet.

Operation	Intel Core i7-3615QM	IBM Power 730 Express
spinlock_push	17	29
lockfree_push	25	12
spinlock_pop	18	29
lockfree_pop	27	12

Lock-less Synchronization

Non-blocking synchronization
is not a silver bullet.

Operation	Intel Core i7-3615QM	IBM Power 730 Express
spinlock_push	17	29
lockfree_push	25	12
spinlock_pop	18	29
lockfree_pop	27	12

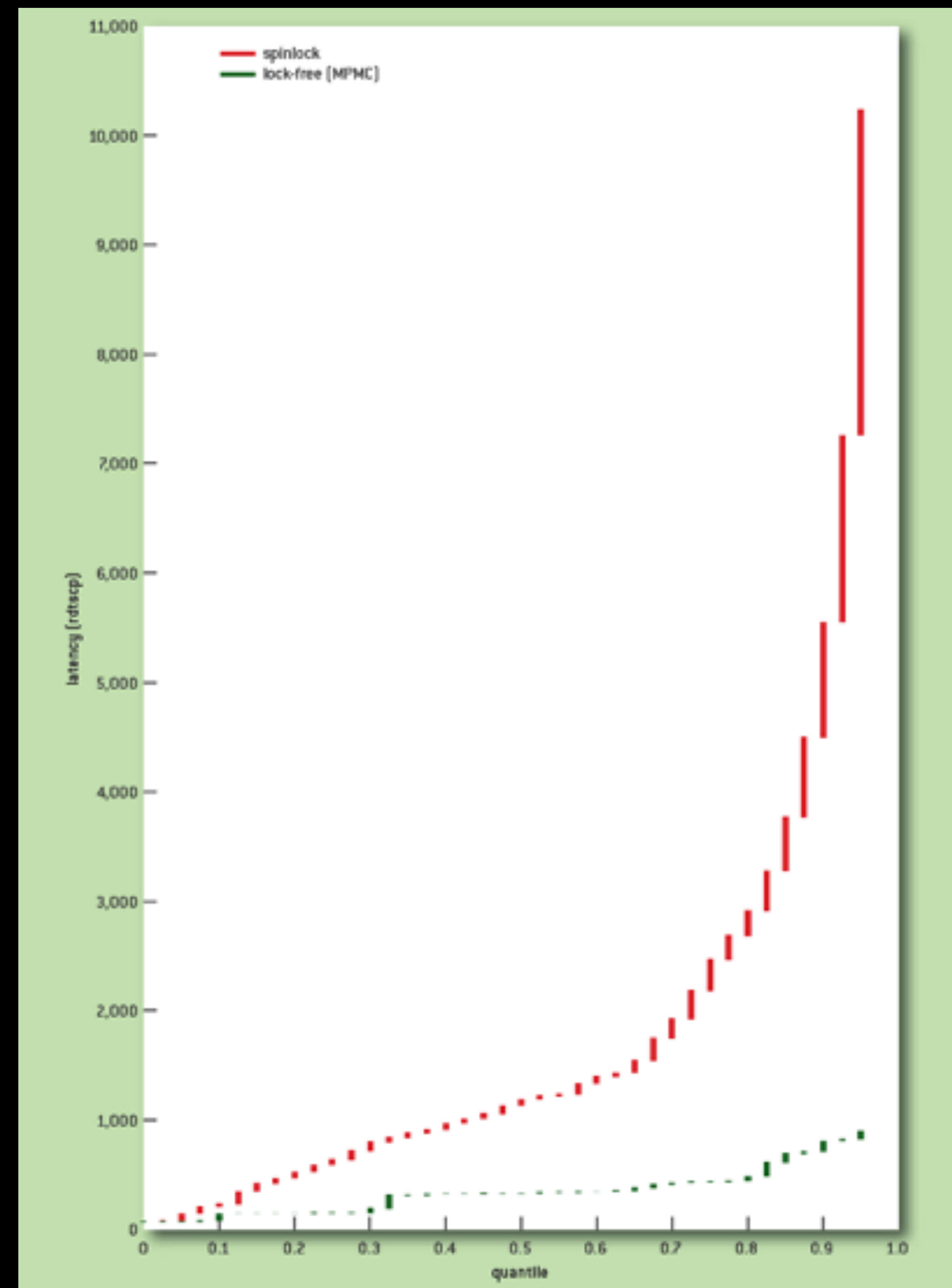
The cost of complexity on the fast path will outweigh the benefits until sufficient levels of contention are reached.

Lock-less Synchronization

Non-blocking synchronization is not a silver bullet.

Operation	Intel Core i7-3615QM	IBM Power 730 Express
spinlock_push	17	29
lockfree_push	25	12
spinlock_pop	18	29
lockfree_pop	27	12

The cost of complexity on the fast path will outweigh the benefits until sufficient levels of contention are reached.



Source: Nonblocking Algorithms and Scalable Multicore Programming, Samy Bahra

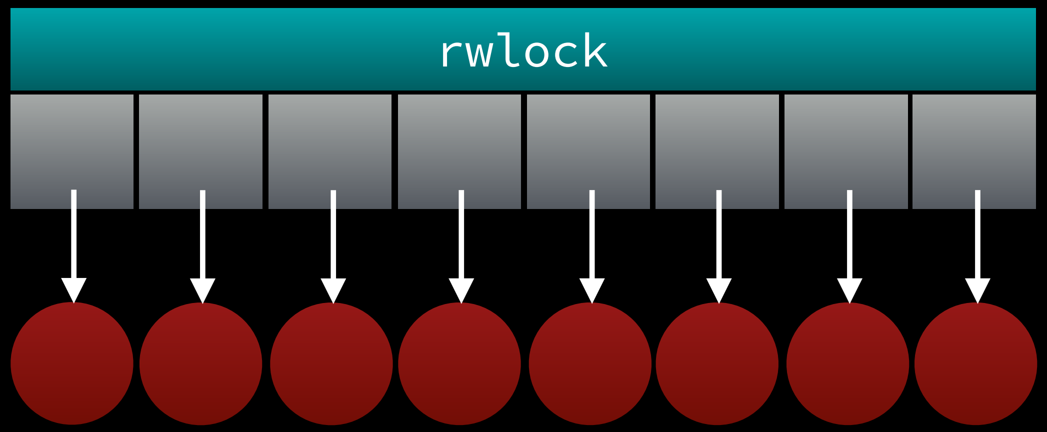
Backtrace

Lock-less Synchronization

Relaxing correctness constraints and constraining runtime requirements allows for many of the benefits without as much additional complexity and impact on the fast path.

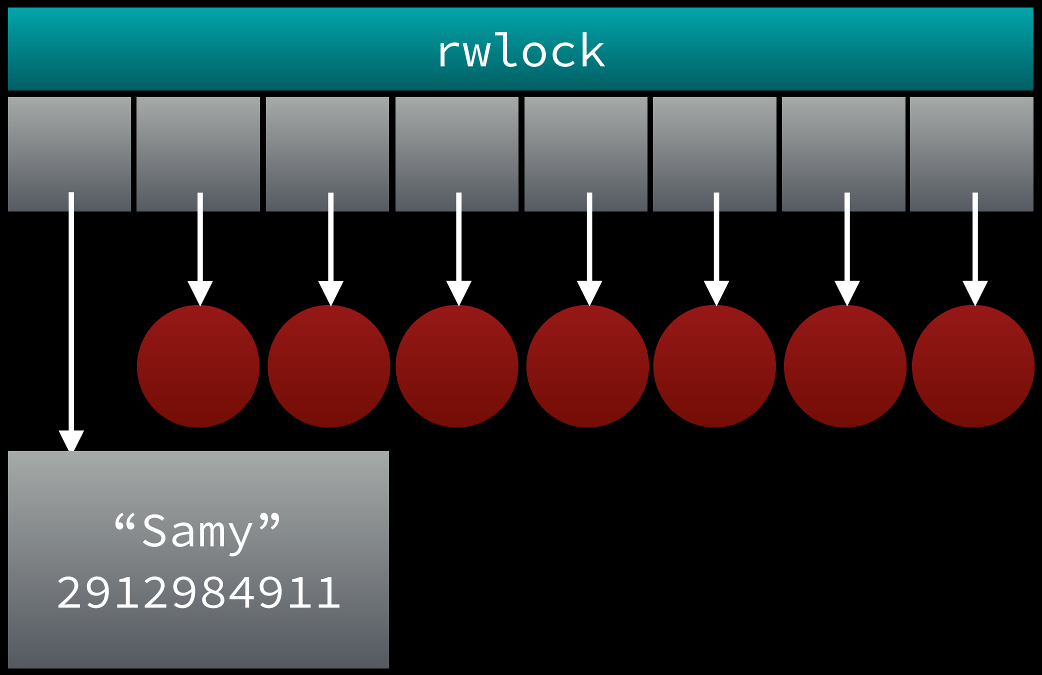
Lock-less Synchronization

```c
#define EMPLOYEE_MAX 8

struct employee {
    const char *name;
    unsigned long long number;
};

struct directory {
    struct employee *employee[EMPLOYEE_MAX];
    rwlock_t rwlock;
};

bool employee_add(struct directory *, const char *,
    unsigned long long);
void employee_delete(struct directory *, const char *);
unsigned long long employee_number_get(struct directory *,
    const char *);
```



Lock-less Synchronization

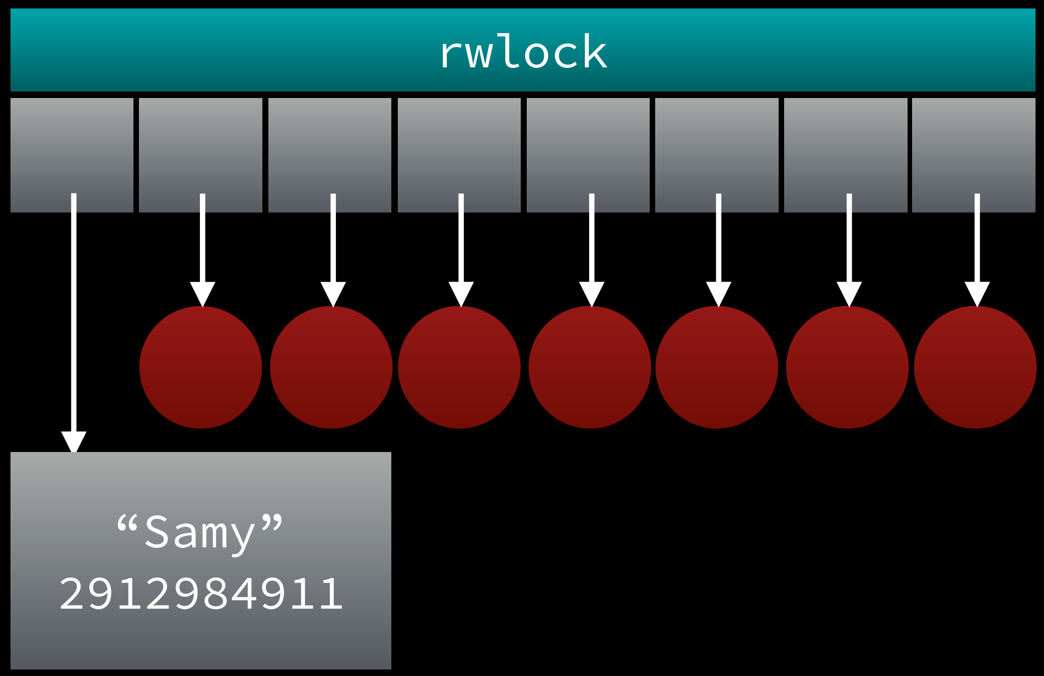
```
unsigned long long
employee_number_get(struct directory *d, const char *n)
{
    struct employee *em;
    unsigned long number;
    size_t i;

    rwlock_read_lock(&d->rwlock);
    for (i = 0; i < EMPLOYEE_MAX; i++) {
        em = d->employee[i];
        if (em == NULL)
            continue;

        if (strcmp(em->name, n) != 0)
            continue;

       number = em->number;
       rwlock_read_unlock(&d->rwlock);

       return number;
   }
   rwlock_read_unlock(&d->rwlock);

   return 0;
}
```



The rwlock_t object provides correctness at cost of forward progress.

Lock-less Synchronization

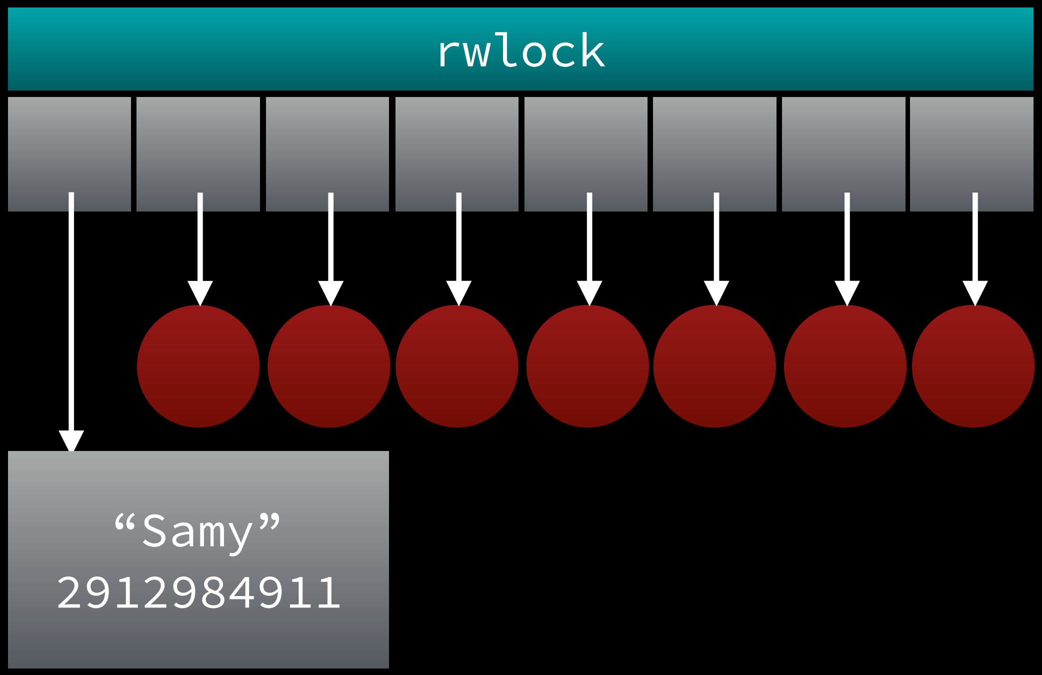
```
bool
employee_add(struct directory *d, const char *n,
    unsigned long long number)
{
    struct employee *em;
    size_t i;

    rwlock_write_lock(&d->rwlock);
    for (i = 0; i < EMPLOYEE_MAX; i++) {
        if (d->employee[i] != NULL)
            continue;

        em = xmalloc(sizeof *em);
        em->name = n;
        em->number = number;
        d->employee[i] = em;
        rwlock_write_unlock(&d->rwlock);
        return true;
   }
   rwlock_write_unlock(&d->rwlock);

   return false;
}
```



The rwlock_t object provides correctness at cost of forward progress.

Lock-less Synchronization

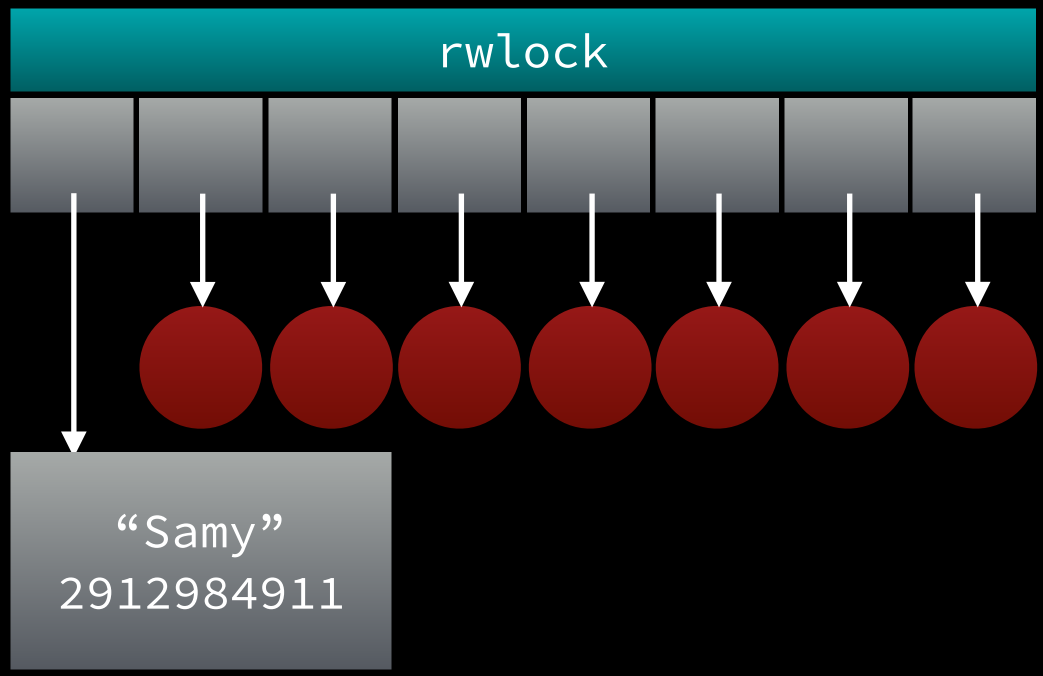
```
void
employee_delete(struct directory *d, const char *n)
{
    struct employee *em;
    size_t i;

    rwlock_write_lock(&d->rwlock);
    for (i = 0; i < EMPLOYEE_MAX; i++) {
        if (d->employee[i] == NULL)
            continue;

        if (strcmp(d->employee[i]->name, n) != 0)
            continue;

       em = d->employee[i];
       d->employee[i] = NULL;
       rwlock_write_unlock(&d->rwlock);
       free(em);

       return;
    }
    rwlock_write_unlock(&d->rwlock);

    return;
}
```



The rwlock_t object provides correctness at cost of forward progress.

Lock-less Synchronization

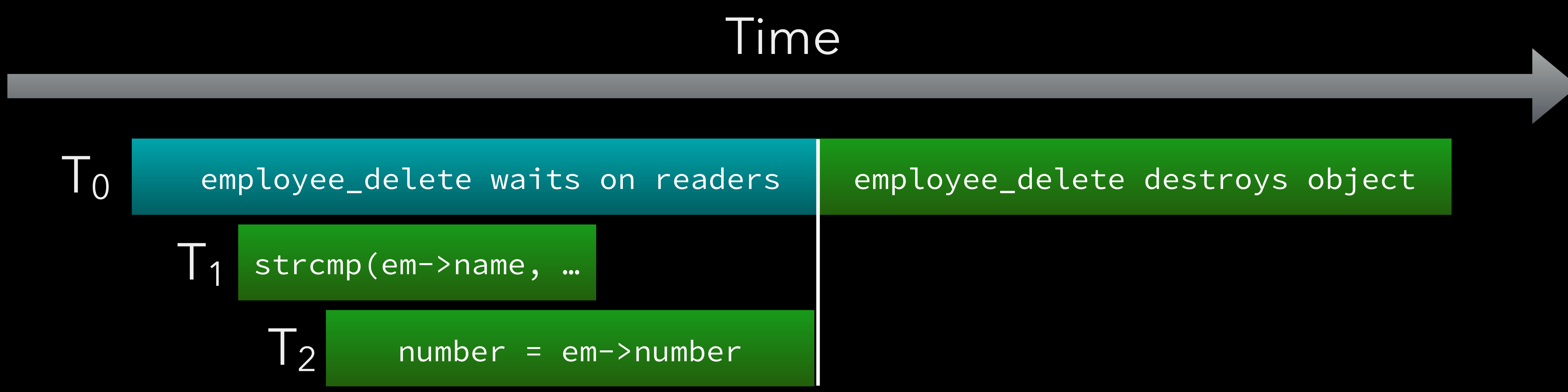
```
void
employee_delete(struct directory *d, const char *n)
{
    struct employee *em;
    size_t i;

    rwlock_write_lock(&d->rwlock);
    for (i = 0; i < EMPLOYEE_MAX; i++) {
        if (d->employee[i] == NULL)
            continue;

        if (strcmp(d->employee[i]->name, n) != 0)
            continue;

      em = d->employee[i];
      d->employee[i] = NULL;
      rwlock_write_unlock(&d->rwlock);
      free(em);

      return;
    }
    rwlock_write_unlock(&d->rwlock);

    return;
}
```



If reachability and liveness are coupled, you also protect against a **read-reclaim** race.

Lock-less Synchronization

If reachability and liveness are coupled, you also protect against a **read-reclaim** race.



Lock-less Synchronization

Decoupling is sometimes necessary, but requires a safe memory reclamation scheme to guarantee that an object cannot be physically destroyed if there are active references to it.

```
static struct employee *
employee_number_get(struct directory *d, const char *n,
   ck_brlock_reader_t *reader)
{
…
        ck_brlock_read_lock(&d->brlock, reader);
        for (i = 0; i < EMPLOYEE_MAX; i++) {
                em = d->employee[i];
                if (em == NULL)
                        continue;

                if (strcmp(em->name, n) != 0)
                        continue;

                ck_pr_inc_uint(&em->ref);
                ck_brlock_read_unlock(reader);
                return em;
        }
        ck_brlock_read_unlock(reader);
…
```

Backtrace

Lock-less Synchronization

Decoupling is sometimes necessary, but requires a safe memory reclamation scheme to guarantee that an object cannot be physically destroyed if there are active references to it.

```
static void
employee_delref(struct employee *em)
{
	bool z;

	ck_pr_dec_uint_zero(&em->ref, &z);
	if (z == true)
		free(em);
	return;
}
```

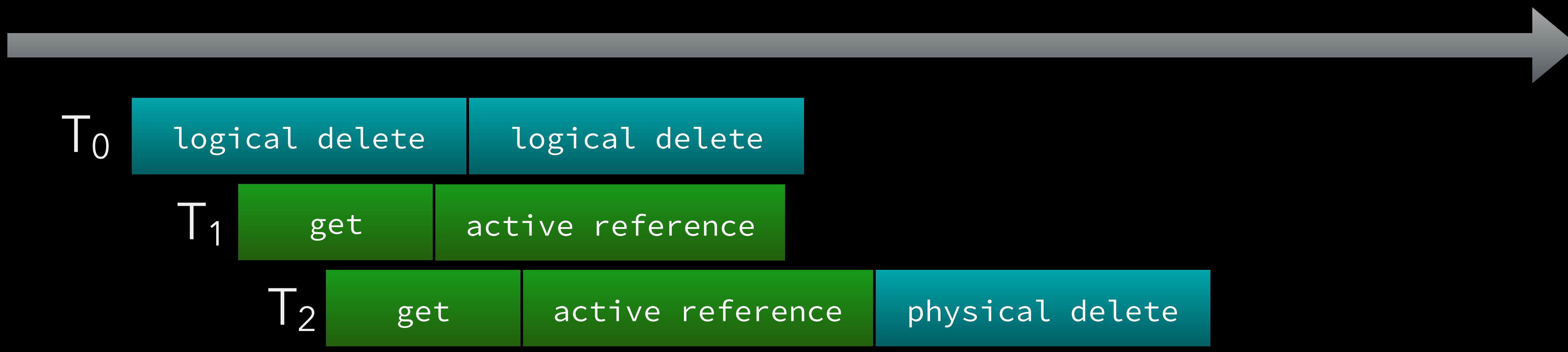
Lock-less Synchronization

Decoupling is sometimes necessary, but requires a safe memory reclamation scheme to guarantee that an object cannot be physically destroyed if there are active references to it.

```
static void
employee_delete(struct directory *d, const char *n)
{
…
        ck_brlock_write_lock(&d->brlock);
        for (i = 0; i < EMPLOYEE_MAX; i++) {
                if (d->employee[i] == NULL)
                        continue;

                if (strcmp(d->employee[i]->name, n) != 0)
                        continue;

                em = d->employee[i];
                d->employee[i] = NULL;
                ck_brlock_write_unlock(&d->brlock);
                employee_delref(em);

                return;
        }
        ck_brlock_write_unlock(&d->brlock);
…
```

Lock-less Synchronization

Decoupling is sometimes necessary, but requires a safe memory reclamation scheme to guarantee that an object cannot be physically destroyed if there are active references to it.



Concurrent Data Structures

```
static bool
employee_add(struct directory *d, const char *n,
        unsigned long long number)
{
        struct employee *em;
        size_t i;

        ck_rwlock_write_lock(&d->rwlock);
        for (i = 0; i < EMPLOYEE_MAX; i++) {
                if (d->employee[i] != NULL)
                        continue;

                em = malloc(sizeof *em);
                em->name = n;
                em->number = number;
                ck_pr_fence_store();
                ck_pr_store_ptr(&d->employee[i], em);
                ck_rwlock_write_unlock(&d->rwlock);
                return true;
        }
        ck_rwlock_write_unlock(&d->rwlock);

        return false;
}
```

```
static unsigned long long
employee_number_get(struct directory *d, const char *n)
{
        struct employee *em;
        unsigned long number;
        size_t i;

        for (i = 0; i < EMPLOYEE_MAX; i++) {
                em = ck_pr_load_ptr(&d->employee[i]);
                if (em == NULL)
                        continue;

                ck_pr_fence_load_depends();
                if (strcmp(em->name, n) != 0)
                        continue;

                number = em->number;
                return number;
        }

        return 0;
}
```

Backtrace

Concurrent Data Structures

```c
static void
employee_delete(struct directory *d, const char *n)
{
        struct employee *em;
        size_t i;

        ck_rwlock_write_lock(&d->rwlock);
        for (i = 0; i < EMPLOYEE_MAX; i++) {
                if (d->employee[i] == NULL)
                        continue;

                if (strcmp(d->employee[i]->name, n) != 0)
                        continue;

                em = d->employee[i];
                ck_pr_store_ptr(&d->employee[i], NULL);
                ck_rwlock_write_unlock(&d->rwlock);

                /* XXX: When is it safe to free em? */
                return;
        }
        ck_rwlock_write_unlock(&d->rwlock);

        return;
}
```

```c
static unsigned long long
employee_number_get(struct directory *d, const char *n)
{
        struct employee *em;
        unsigned long number;
        size_t i;

        for (i = 0; i < EMPLOYEE_MAX; i++) {
                em = ck_pr_load_ptr(&d->employee[i]);
                if (em == NULL)
                        continue;

                ck_pr_fence_load_depends();
                if (strcmp(em->name, n) != 0)
                        continue;

                number = em->number;
                return number;
        }

        return 0;
}
```

Backtrace

EXPERIMENT

Workload

- Uniform read-mostly workload
- Single writer attempts pessimistic add operation at fixed frequency
- Readers attempt to get the number of the first employee

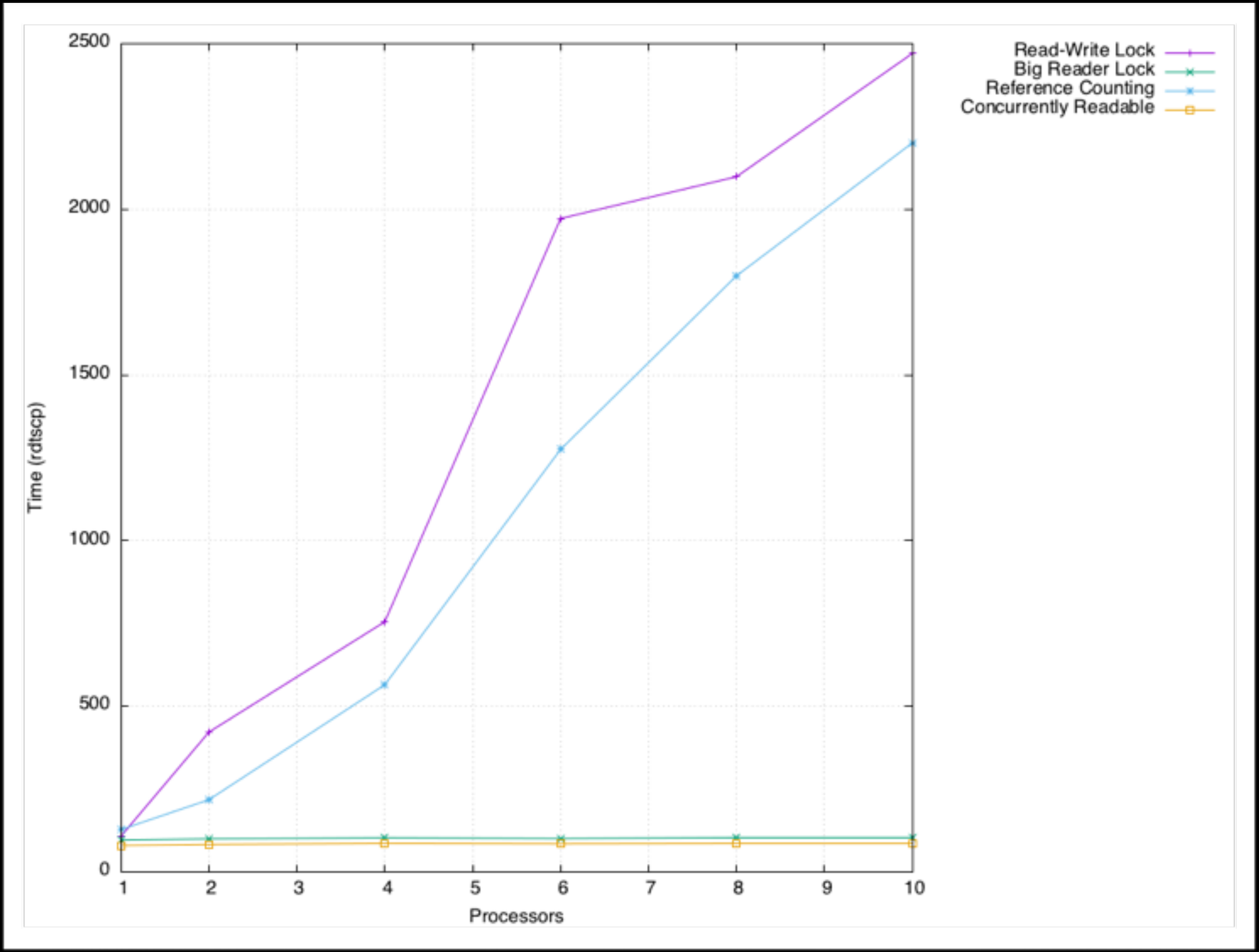
Environment

- 12 cores across 2 sockets
- Intel Xeon E5-2630L at 2.40 GHz
- Linux 2.6.32

```
Machine (64GB)
  NUMANode L#0 (P#0 32GB)
    Socket L#0 + L3 L#0 (15MB)
      L2 L#0 (256KB) + L1d L#0 (32KB) + L1i L#0 (32KB) + Core L#0 + PU L#0 (P#0)
```

Read Latency

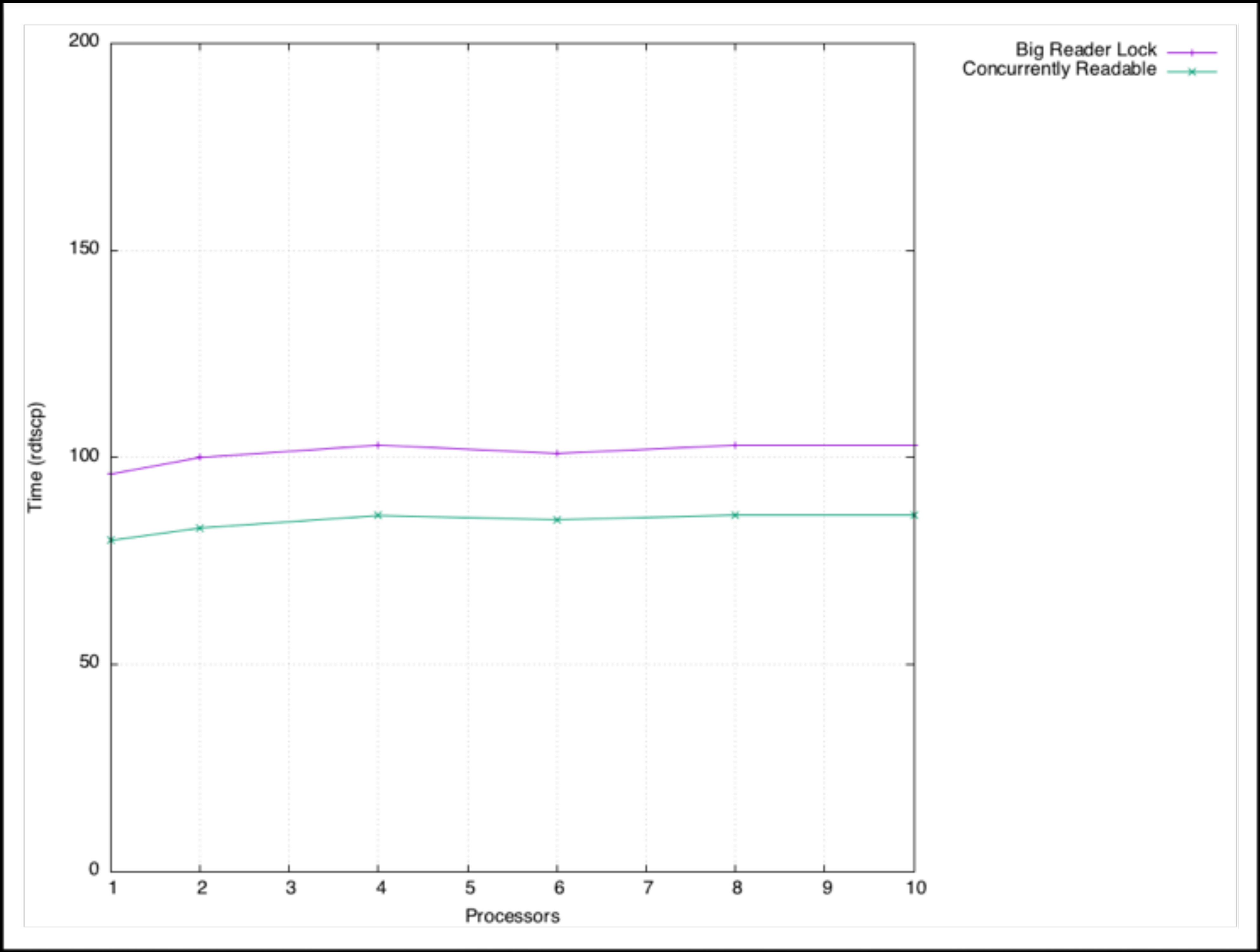
No updates



Backtrace

Read Latency

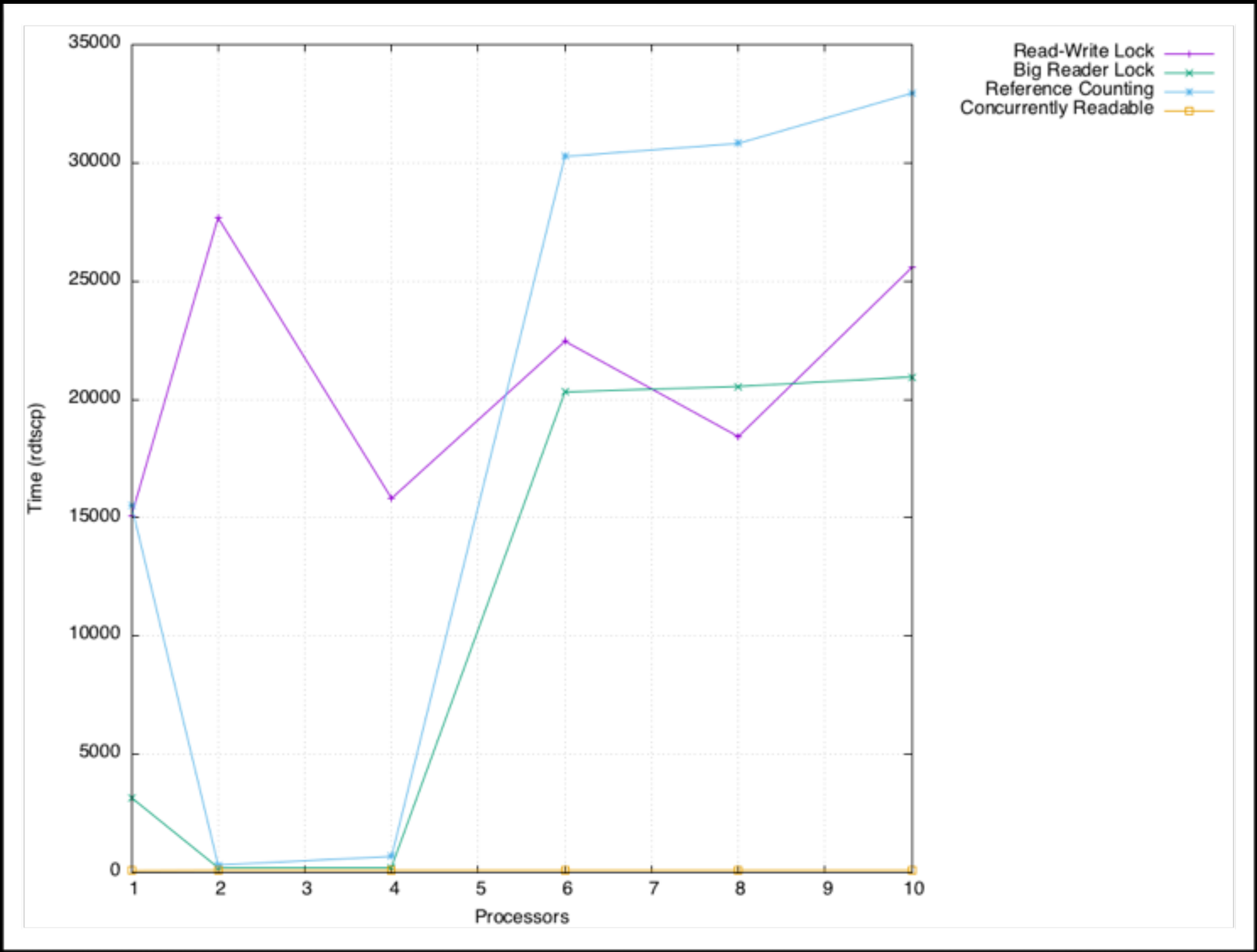
No updates



Backtrace

Read Latency

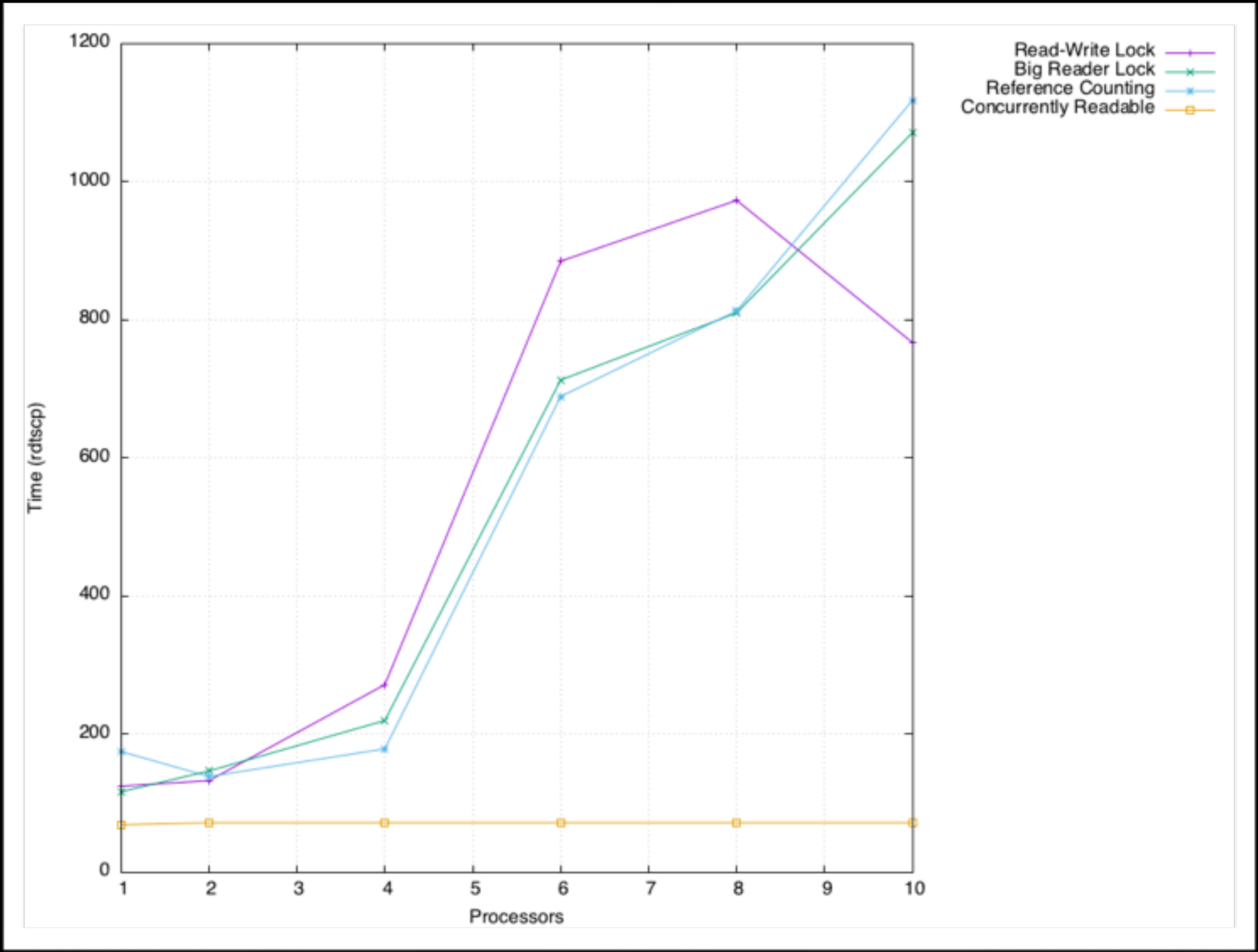
Single writer



Backtrace

Write Latency

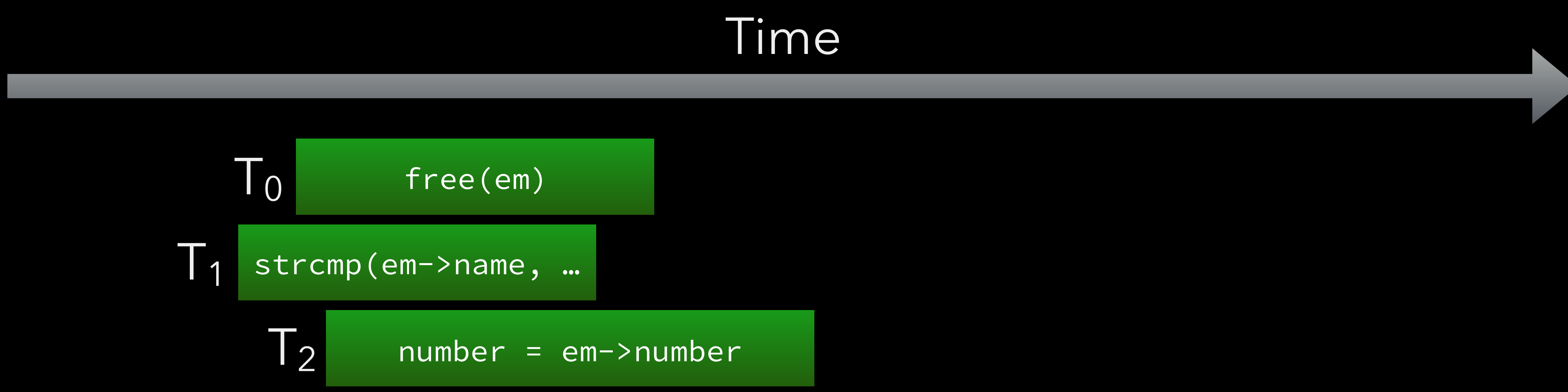
Single writer



Backtrace

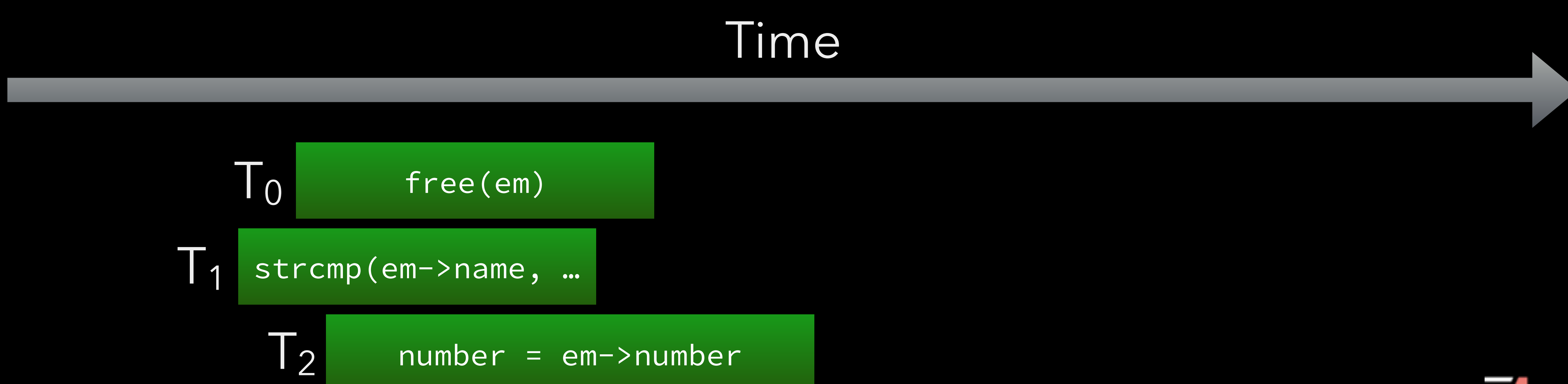
Safe Memory Reclamation

A **read-reclaim** race occurs if an object is destroyed while there are references or accesses to it.



Safe Memory Reclamation

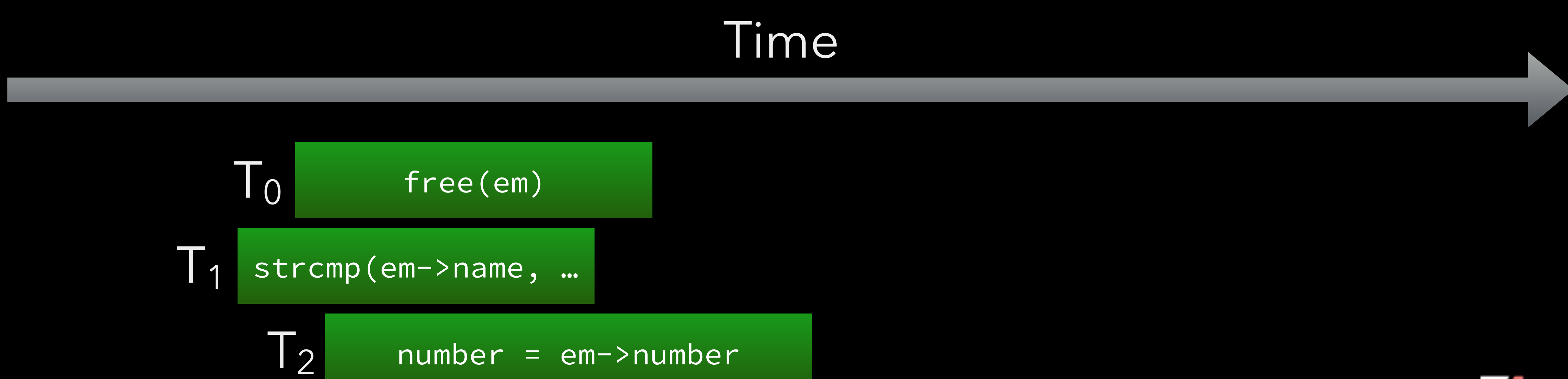
Techniques such as **hazard pointers**, **quiescent-state-based reclamation** and **epoch-based reclamation** protect against read-reclaim races.



Backtrace

Safe Memory Reclamation

Techniques such as **hazard pointers**, **quiescent-state-based reclamation** and **epoch-based reclamation** protect against read-reclaim races.

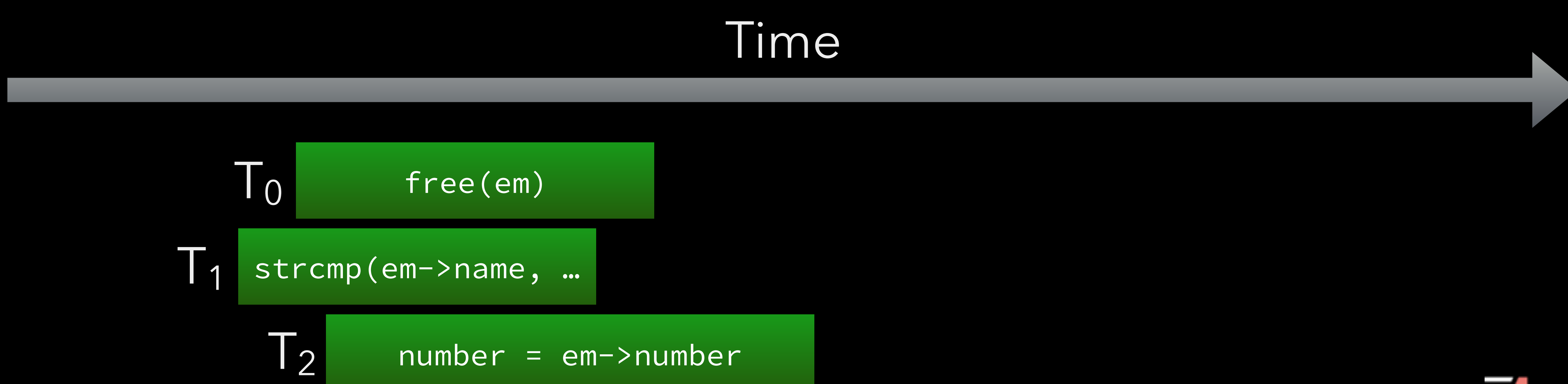


Backtrace

Safe Memory Reclamation

Techniques such as **hazard pointers**, **quiescent-state-based reclamation** and **epoch-based reclamation** protect against read-reclaim races.

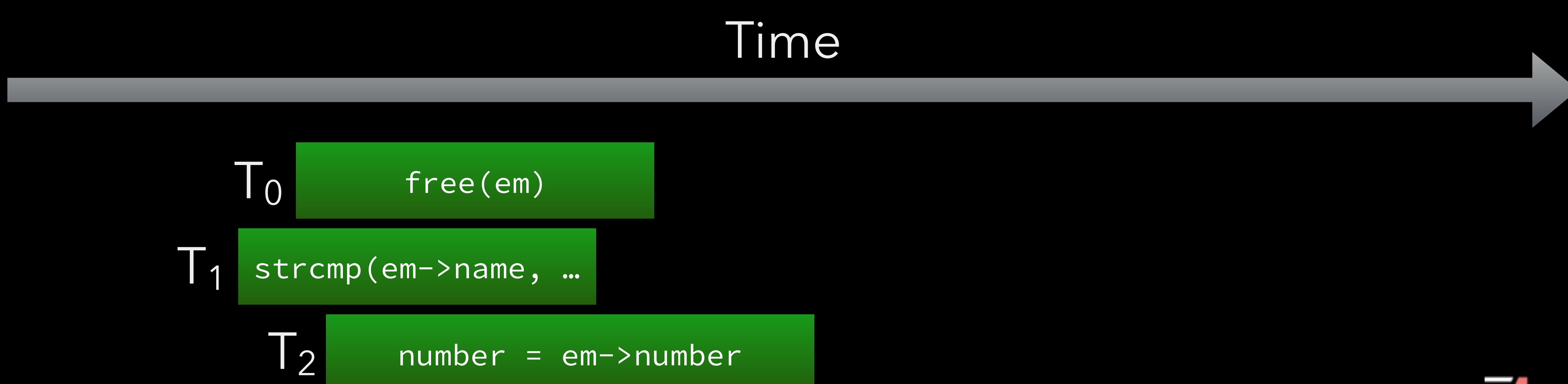
Schemes such as QSBR and EBR do so without affecting reader progress but without guaranteeing writer progress.



Safe Memory Reclamation

Techniques such as **hazard pointers**, **quiescent-state-based reclamation** and **epoch-based reclamation** protect against read-reclaim races.

Schemes such as QSBR and EBR do so without affecting reader progress but without guaranteeing writer progress.



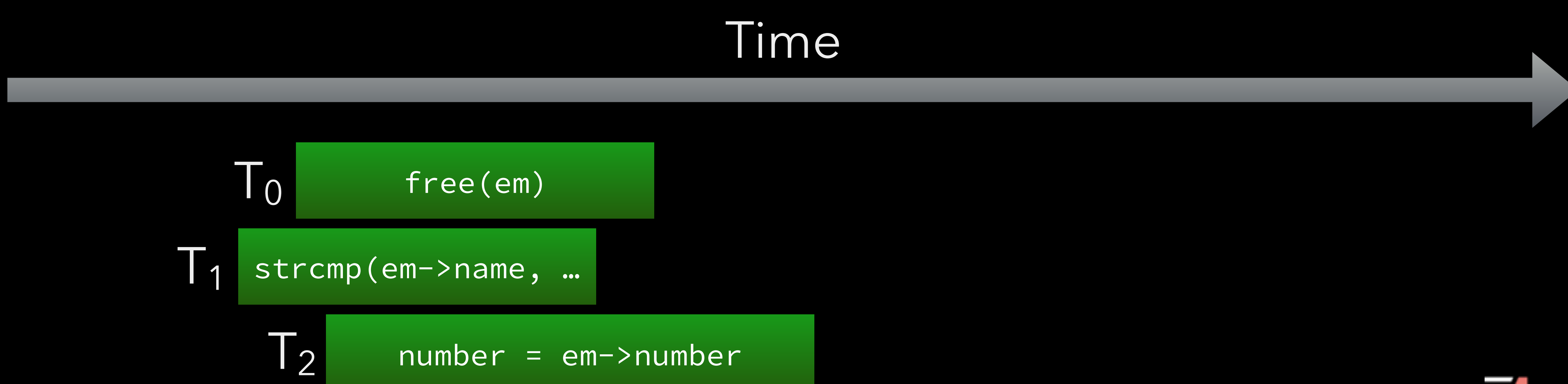
Backtrace

Safe Memory Reclamation

Techniques such as **hazard pointers**, **quiescent-state-based reclamation** and **epoch-based reclamation** protect against read-reclaim races.

Schemes such as QSBR and EBR do so without affecting reader progress but without guaranteeing writer progress.

Schemes provide strong guarantees on forward progress but require heavy-weight instructions and retry logic for readers.



Backtrace

BLOCKING SMR SCHEMES

- Read-side critical sections

```
smr_read_lock();

<protected section>

smr_read_unlock();
```

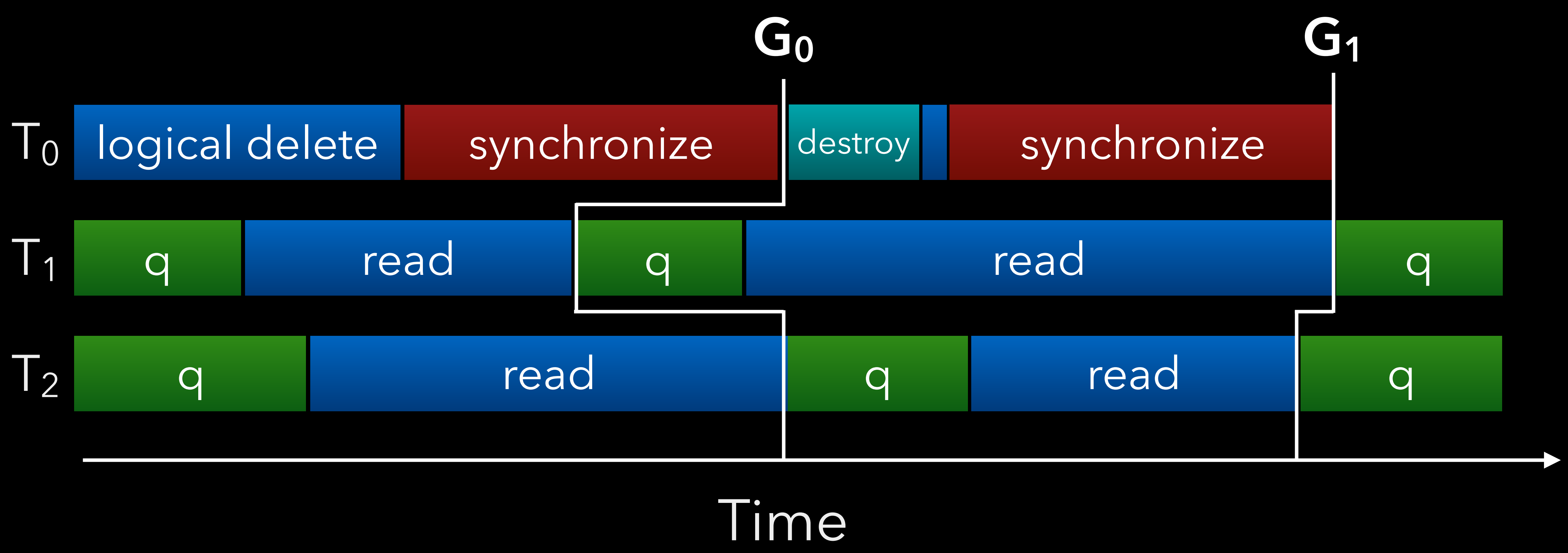
```
smr_read_begin();

<protected section>

smr_read_end();
```

- Explicit Reclamation

```
smr_synchronize();
```

Backtrace

QUIESCENT-STATE-BASED RECLAMATION



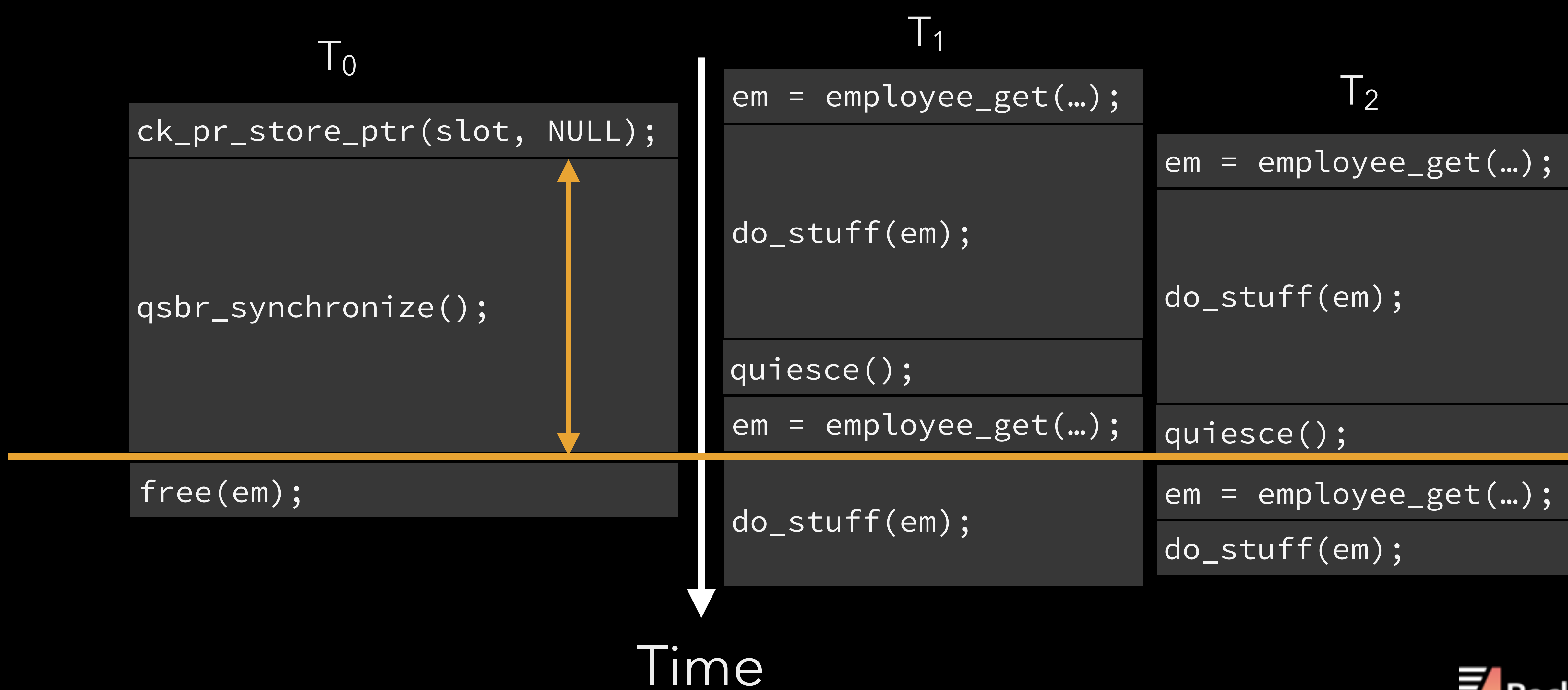
QUIESCENT-STATE-BASED RECLAMATION

Writer

```
employee_number_delete
[…]
    ck_pr_store_ptr(slot, NULL);
    qsbr_synchronize();
    free(em);
[…]
```

Readers

```
[…]
    for (;;) {
        em = employee_get(…);
        do_stuff(em);
        quiesce();
    }
[…]
```



QUIESCENT-STATE-BASED RECLAMATION

Writers

```
static void
qsbr_synchronize(void)
{
        int i;
        uint64_t goal;

        ck_pr_fence_memory();
        goal = ck_pr_faa_64(&global.value, 1) + 1;

        for (i = 0; i < n_reader; i++) {
                uint64_t *c =
                  &threads.readers[i].counter.value;

                while (ck_pr_load_64(c) < goal)
                        ck_pr_stall();
        }

        return;
}
```

Readers

```
static void
qsbr_quiesce(struct thread *th)
{
        uint64_t v;

        ck_pr_fence_memory();
        v =  ck_pr_load_64(&global.value);
        ck_pr_store_64(&th->counter.value, v);
        ck_pr_fence_memory();
        return;
}
```

```
static void
qsbr_read_lock(struct thread *th)
{

        ck_pr_barrier(); /* Compiler barrier. */
        return;
}

static void
qsbr_read_unlock(struct thread *th)
{

        ck_pr_barrier(); /* Compiler barrier. */
        return;
}
```

Conclusion

There are no silver bullets in multicore synchronization, but a deep understanding of both your workload and your underlying environment may allow you to extract phenomenal performance and reliability increases.

The End

@0xF390

http://concurrencykit.org

http://backtrace.io/

A lot of the content can be found on https://queue.acm.org/detail.cfm?id=2492433 - along with references.