



**HAL**  
open science

# Remote Core Locking: Migrating Critical-Section Execution to Improve the Performance of Multithreaded Applications

Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia Lawall, Gilles Muller

## ► To cite this version:

Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia Lawall, Gilles Muller. Remote Core Locking: Migrating Critical-Section Execution to Improve the Performance of Multithreaded Applications. USENIX ATC '12, Jun 2012, Boston, United States. hal-00991709v1

**HAL Id: hal-00991709**

**<https://inria.hal.science/hal-00991709v1>**

Submitted on 16 May 2014 (v1), last revised 21 Dec 2022 (v2)

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Remote Core Locking

Migrating Critical-Section Execution to Improve the Performance of Multithreaded Applications

**Jean-Pierre Lozi**  
LIP6/INRIA

Florian David  
LIP6/INRIA

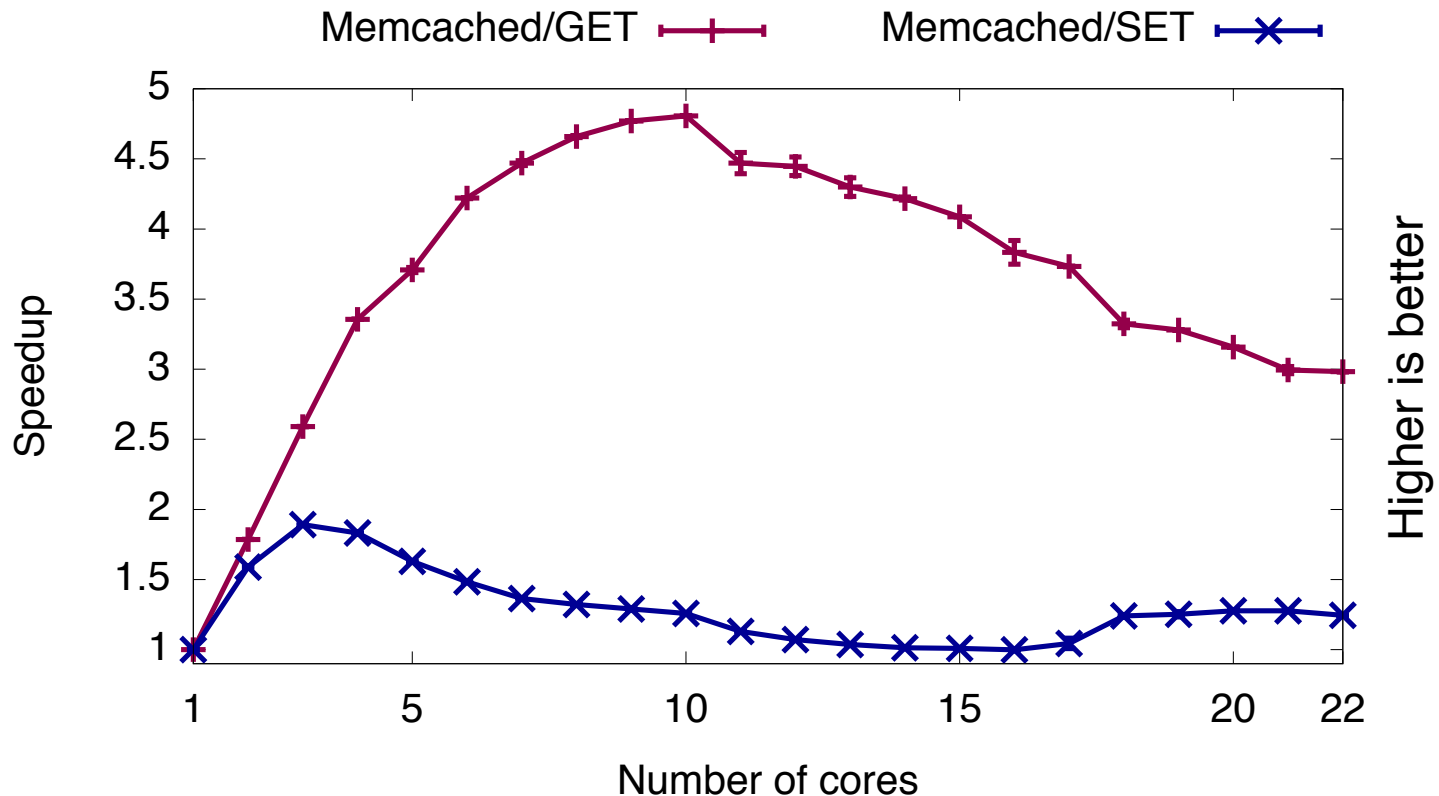
Gaël Thomas  
LIP6/INRIA

Julia Lawall  
LIP6/INRIA

Gilles Muller  
LIP6/INRIA

# Problem: scalability

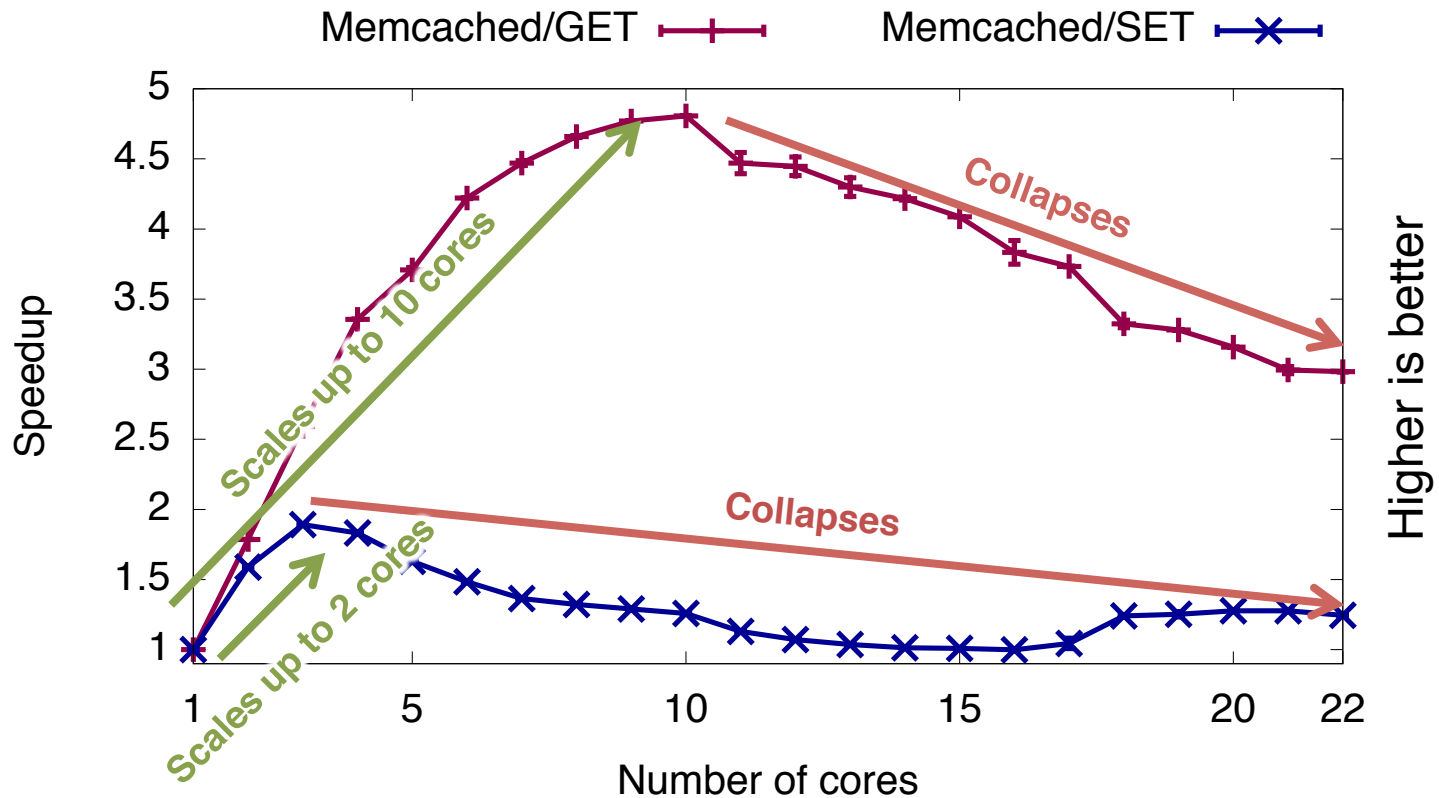
- Many legacy applications don't scale well on multicore architectures
- For instance, Memcached (GET/SET requests):



Experiments run on a 48-core, "magny-cours" x86 AMD machine

# Problem: scalability

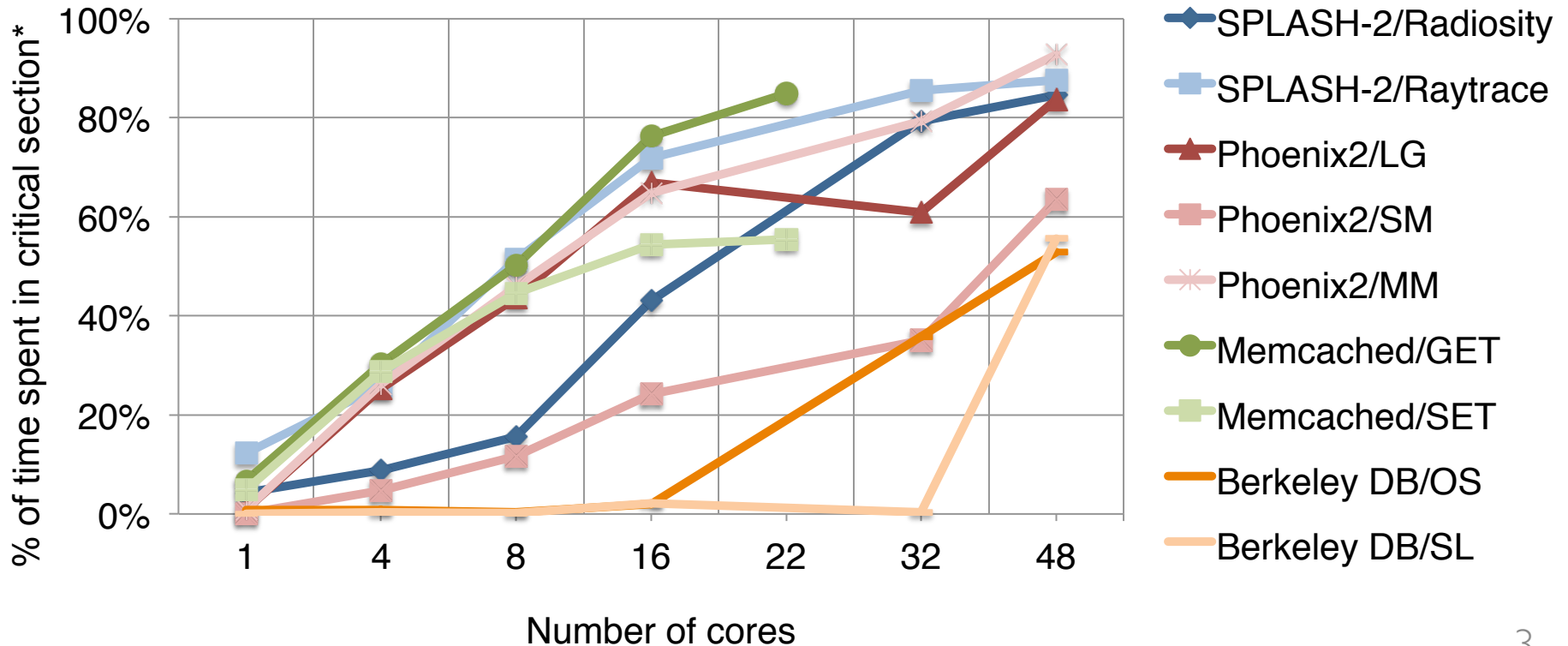
- Many legacy applications don't scale well on multicore architectures
- For instance, Memcached (GET/SET requests):



Experiments run on a 48-core, "magny-cours" x86 AMD machine

# Why?

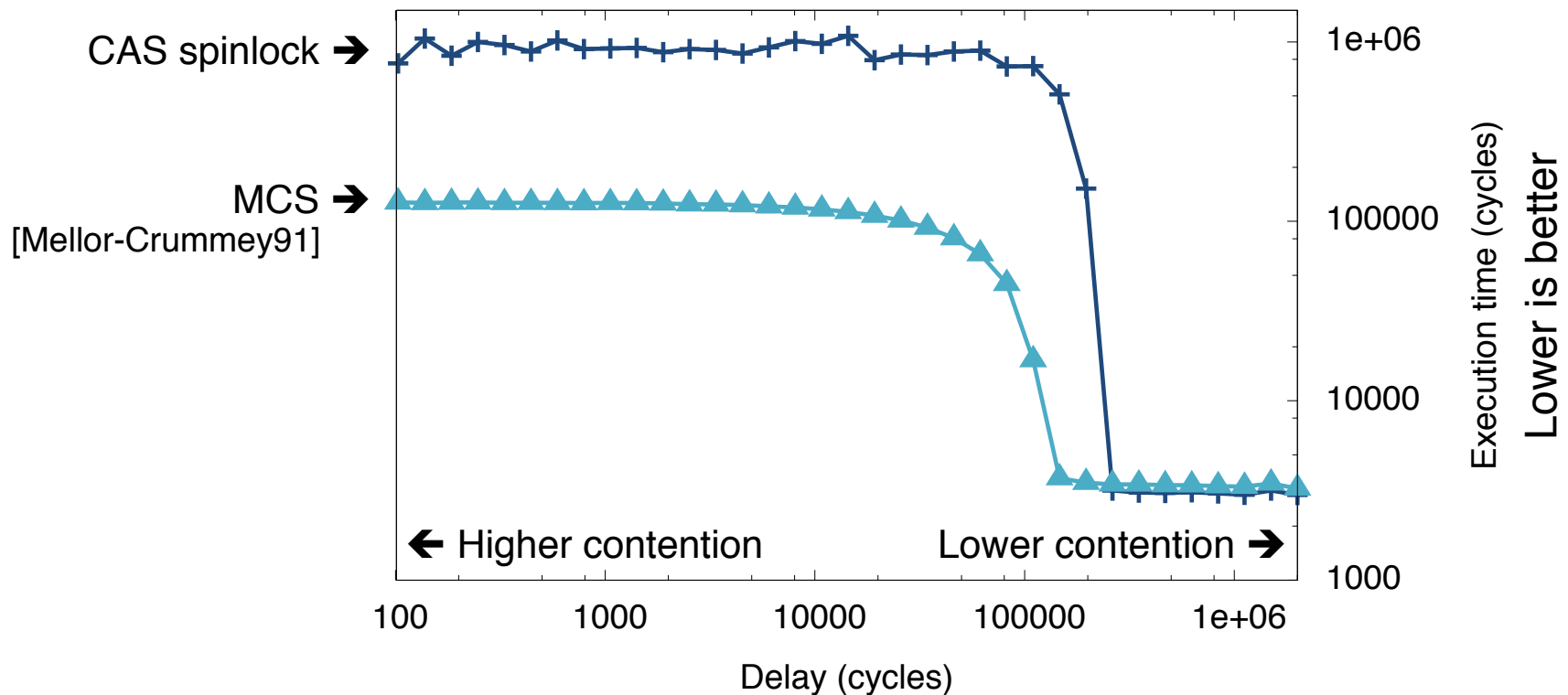
- Critical sections = bottleneck on multicore architectures
- High contention  $\Rightarrow$  lock acquisition is costly
  - More cores  $\Rightarrow$  more contention



\* Including lock acquisition time

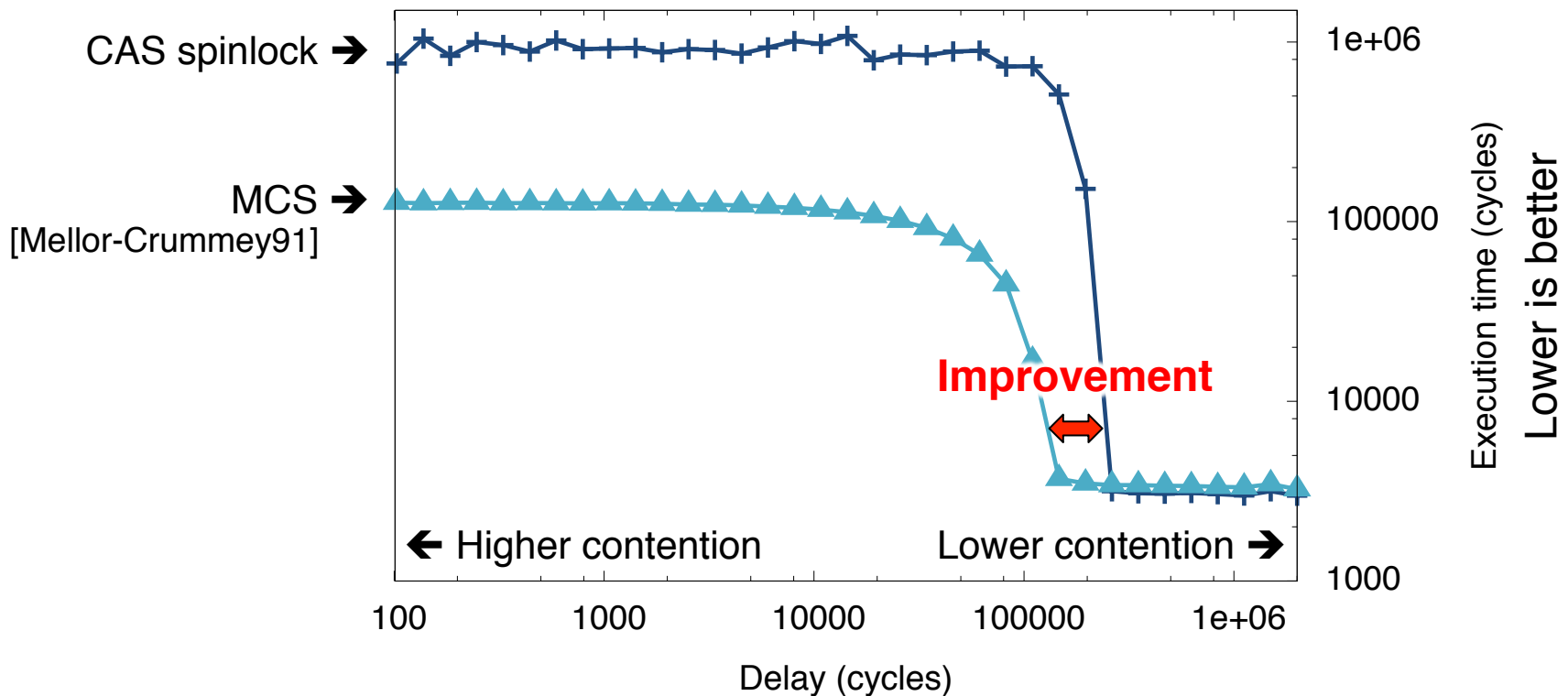
# Solution: designing better locks

- Better resistance to contention
- No need to redesign the application
- Custom microbenchmark to compare locks:



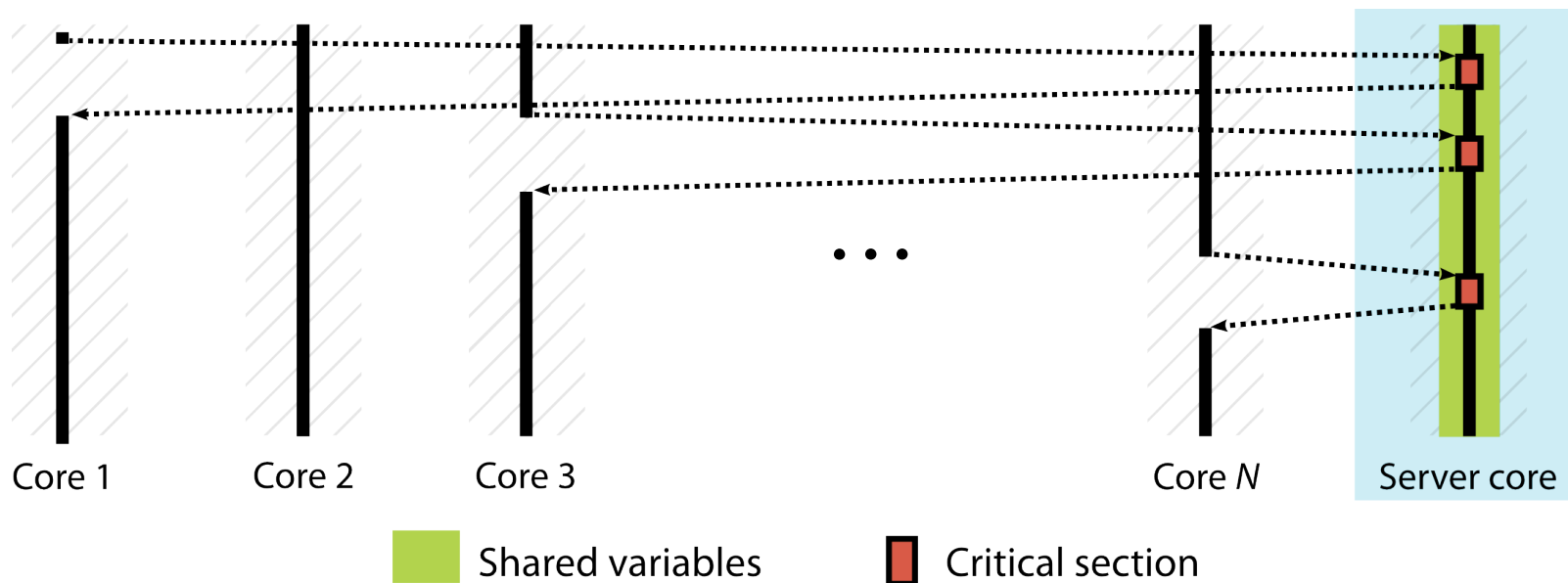
# Solution: designing better locks

- Better resistance to contention
- No need to redesign the application
- Custom microbenchmark to compare locks:



# Remote Core Locking

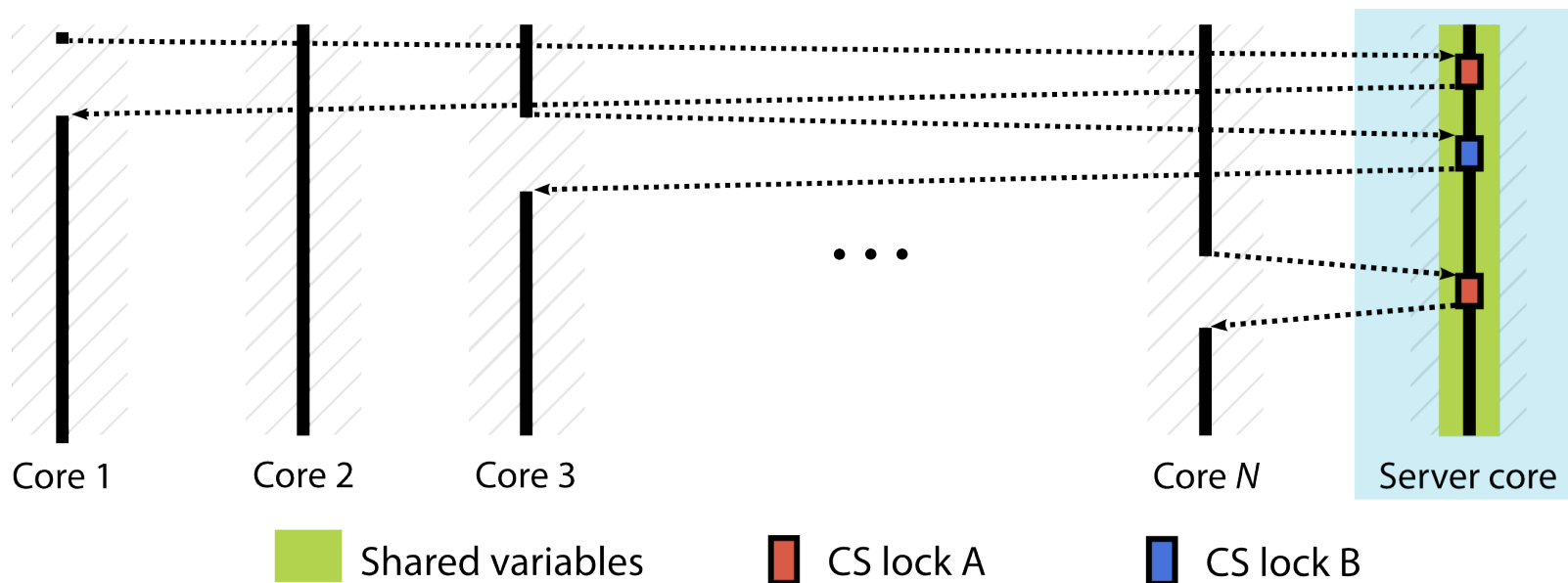
- Objective:** remove atomic instructions and reduce cache misses
- Execute contended critical sections on a dedicated server core
  - Very fast transfer of control, no sync on global variable
    - Faster than lock acquisitions when contention is high
  - Shared data remains on server core  $\Rightarrow$  fewer cache misses





# Remote Core Locking

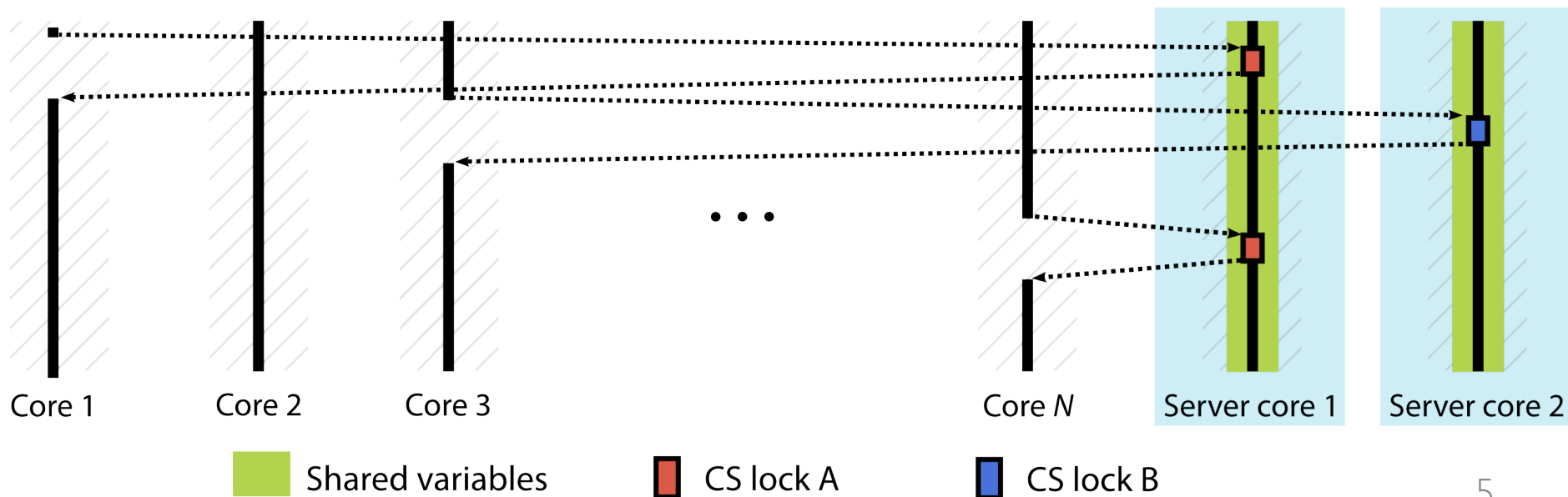
- Objective:** remove atomic instructions and reduce cache misses
- Execute contended critical sections on a dedicated server core
  - Very fast transfer of control, no sync on global variable
    - Faster than lock acquisitions when contention is high
  - Shared data remains on server core  $\Rightarrow$  fewer cache misses



# Remote Core Locking

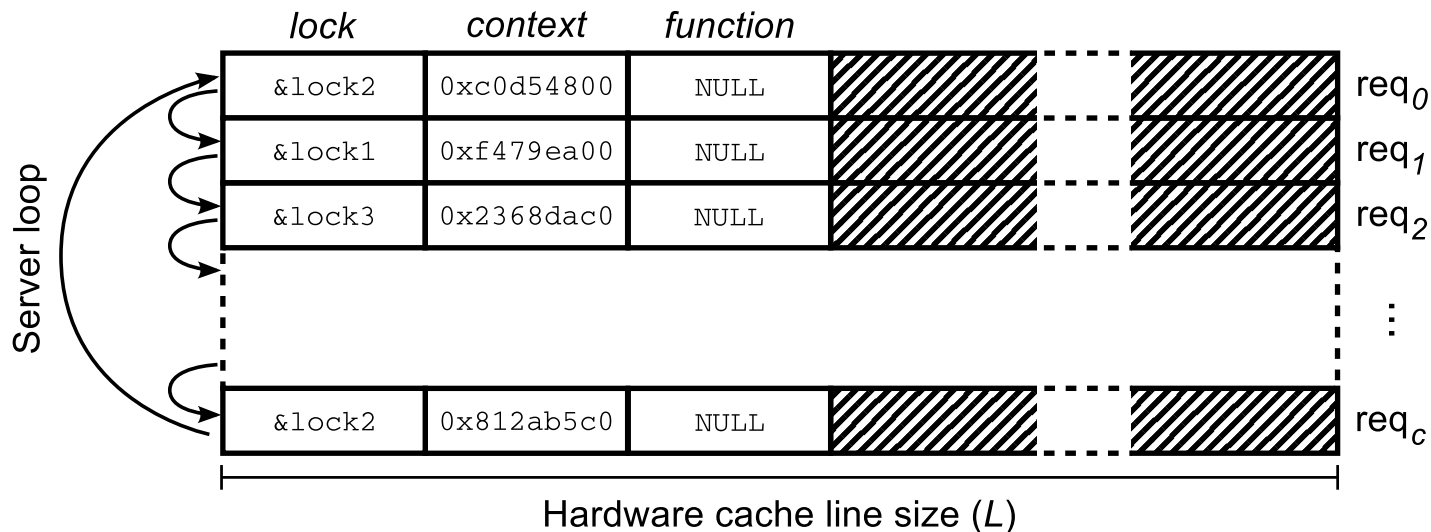
**Objective:** remove atomic instructions and reduce cache misses

- Execute contended critical sections on a dedicated server core
- Very fast transfer of control, no sync on global variable
  - Faster than lock acquisitions when contention is high
- Shared data remains on server core  $\Rightarrow$  fewer cache misses



# Implementation: general idea

- Implementation based on cache line-sized mailboxes
- Three fields: lock, context, function

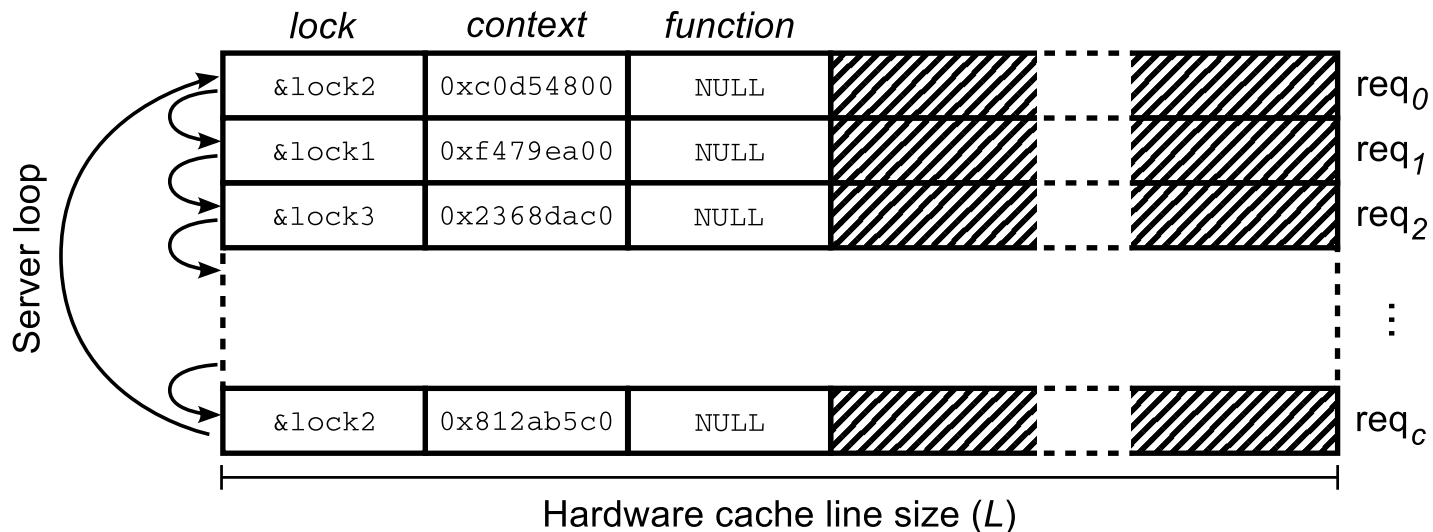


- Client fills the field and waits for the function to be reset
- Server loops across the fields

# Implementation: general idea

- Implementation based on cache line-sized mailboxes
- Three fields: lock, context, function

**Client thread 2 wants to execute a critical section protected by “lock4”**

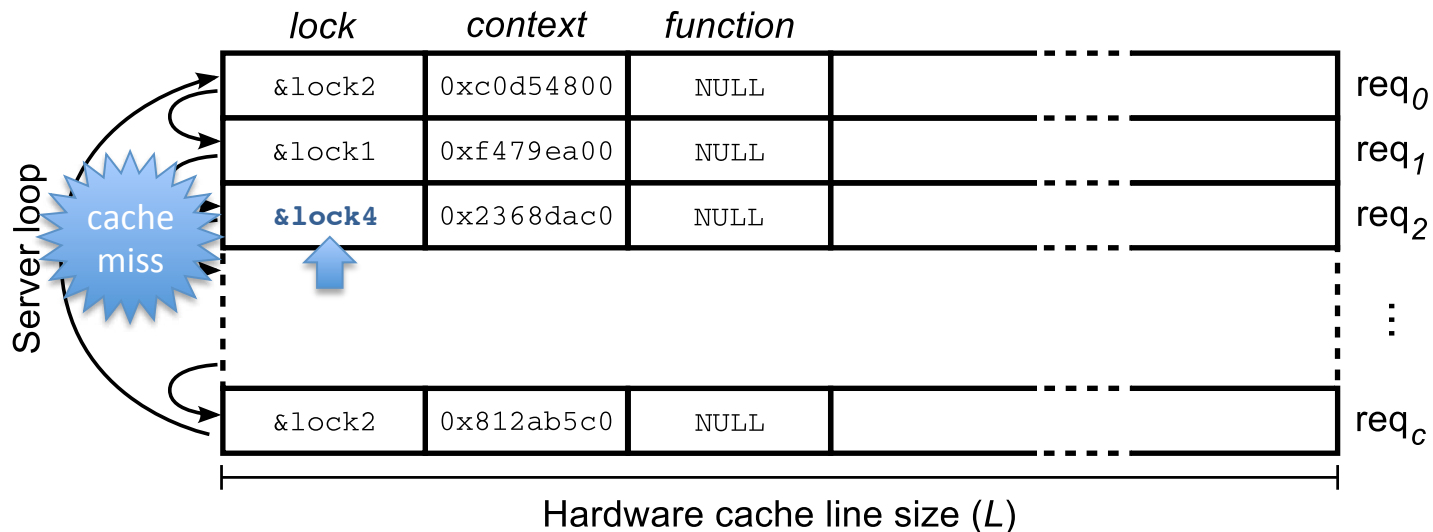


- Client fills the field and waits for the function to be reset
- Server loops across the fields

# Implementation: general idea

- Implementation based on cache line-sized mailboxes
- Three fields: lock, context, function

Client thread 2 wants to execute a critical section protected by “lock4”

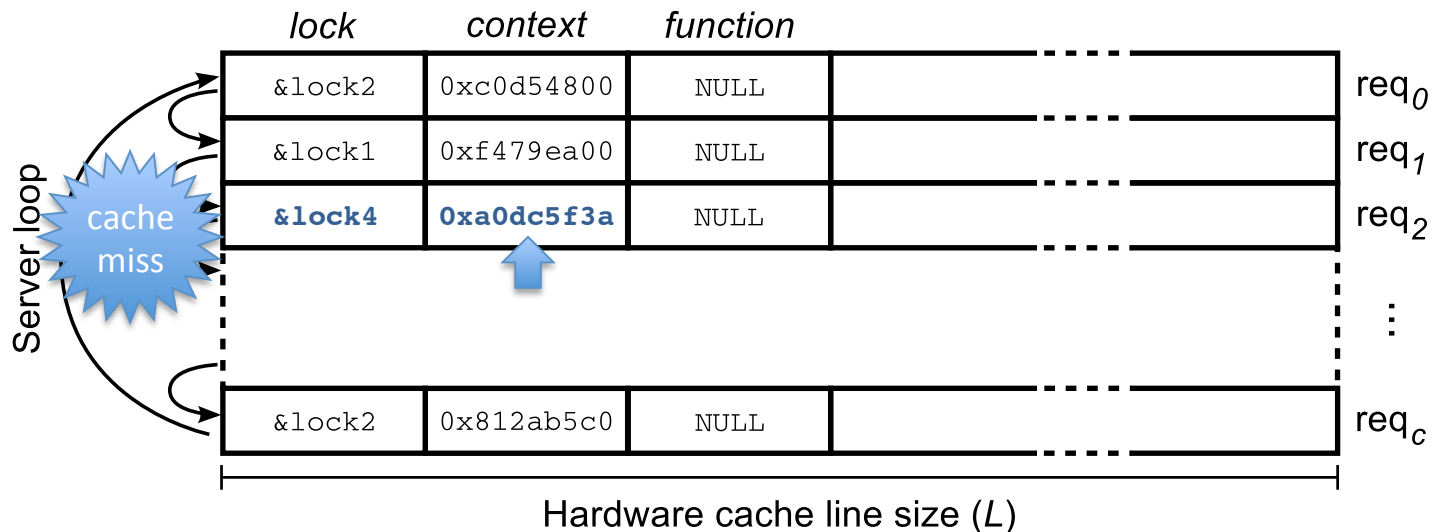


- Client fills the field and waits for the function to be reset
- Server loops across the fields

# Implementation: general idea

- Implementation based on cache line-sized mailboxes
- Three fields: lock, context, function

**Client thread 2 wants to execute a critical section protected by “lock4”**

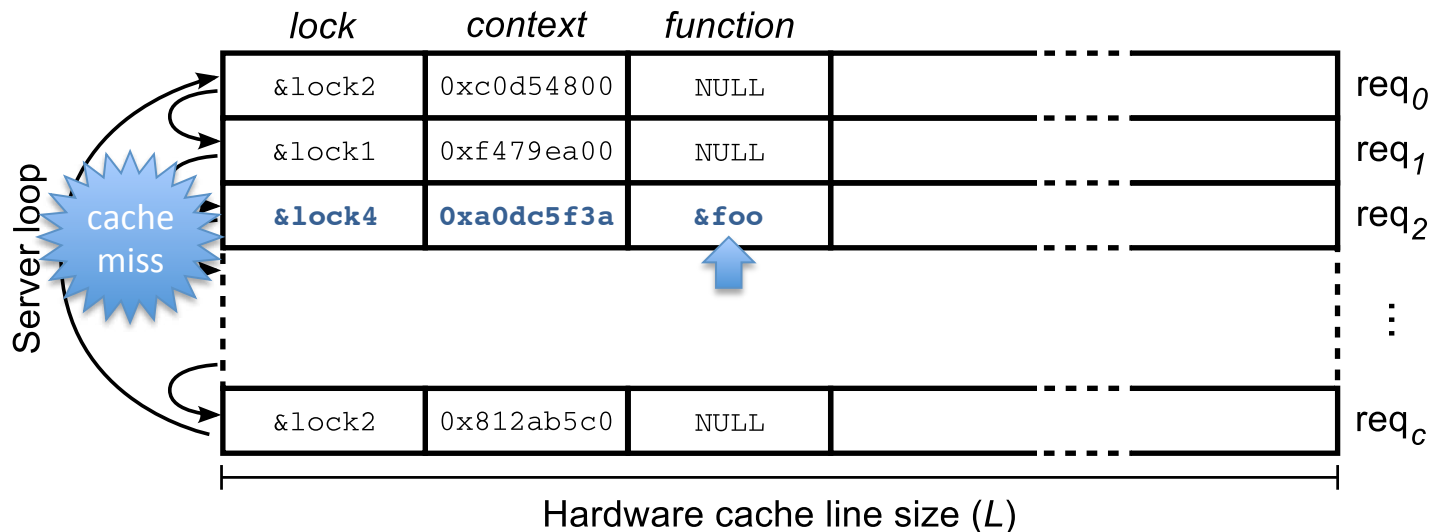


- Client fills the field and waits for the function to be reset
- Server loops across the fields

# Implementation: general idea

- Implementation based on cache line-sized mailboxes
- Three fields: lock, context, function

**Client thread 2 wants to execute a critical section protected by “lock4”**

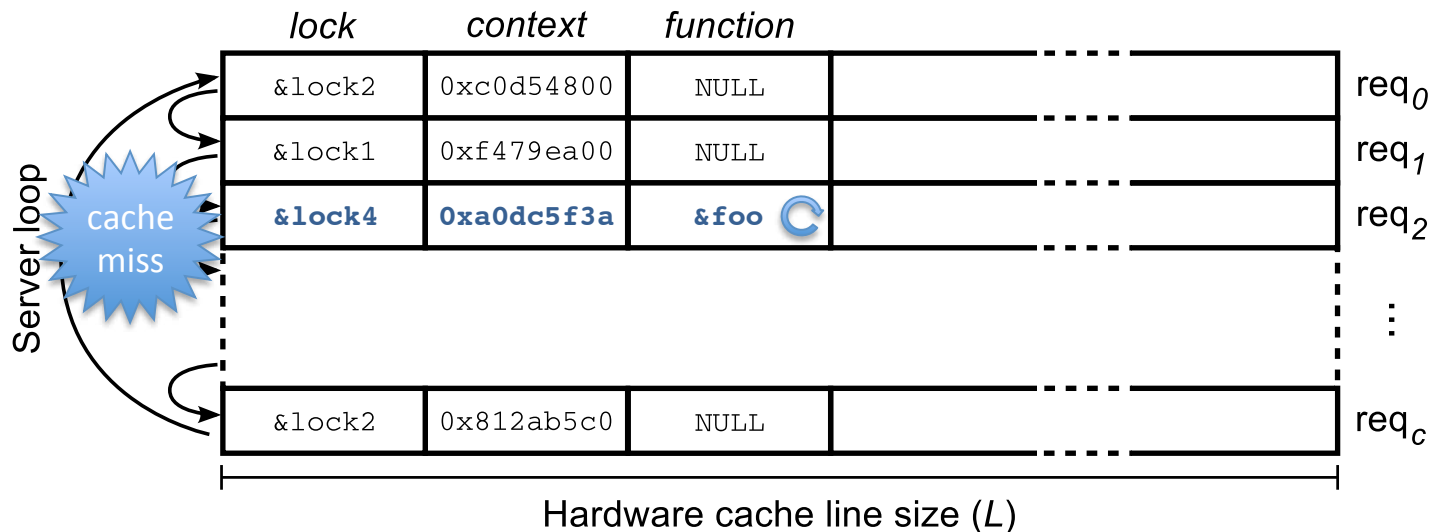


- Client fills the field and waits for the function to be reset
- Server loops across the fields

# Implementation: general idea

- Implementation based on cache line-sized mailboxes
- Three fields: lock, context, function

**Client thread 2 wants to execute a critical section protected by “lock4”**



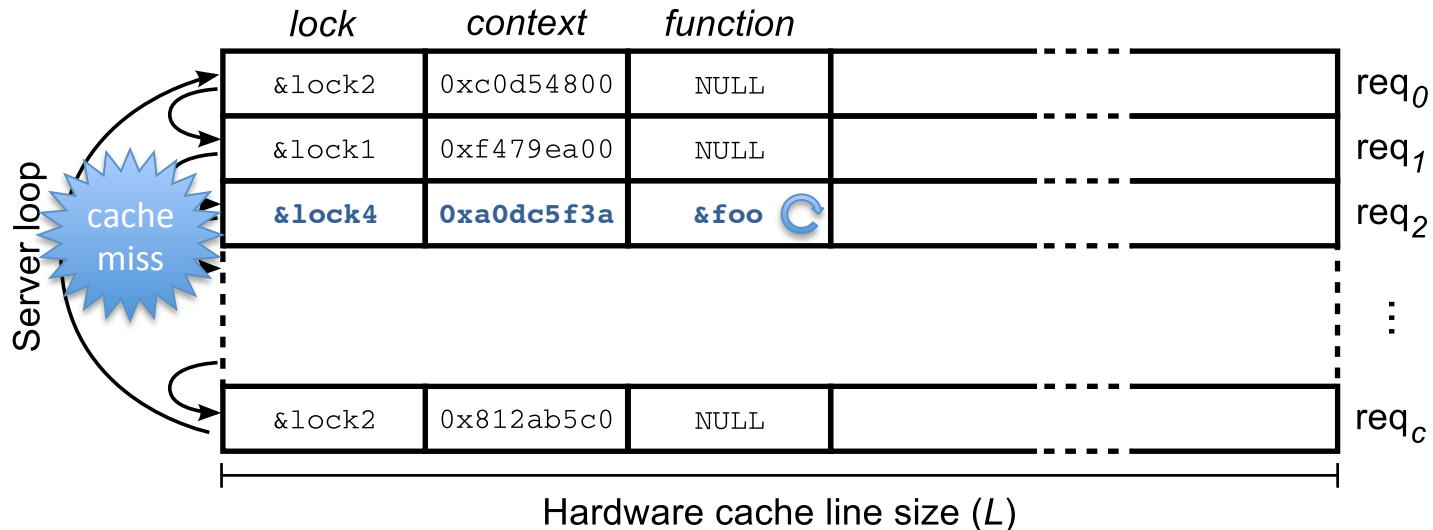
- Client fills the field and waits for the function to be reset
- Server loops across the fields



# Implementation: general idea

- Implementation based on cache line-sized mailboxes
- Three fields: lock, context, function

**Client thread 2 wants to execute a critical section protected by “lock4”**  
**Server continuously checks mailboxes and executes critical sections**

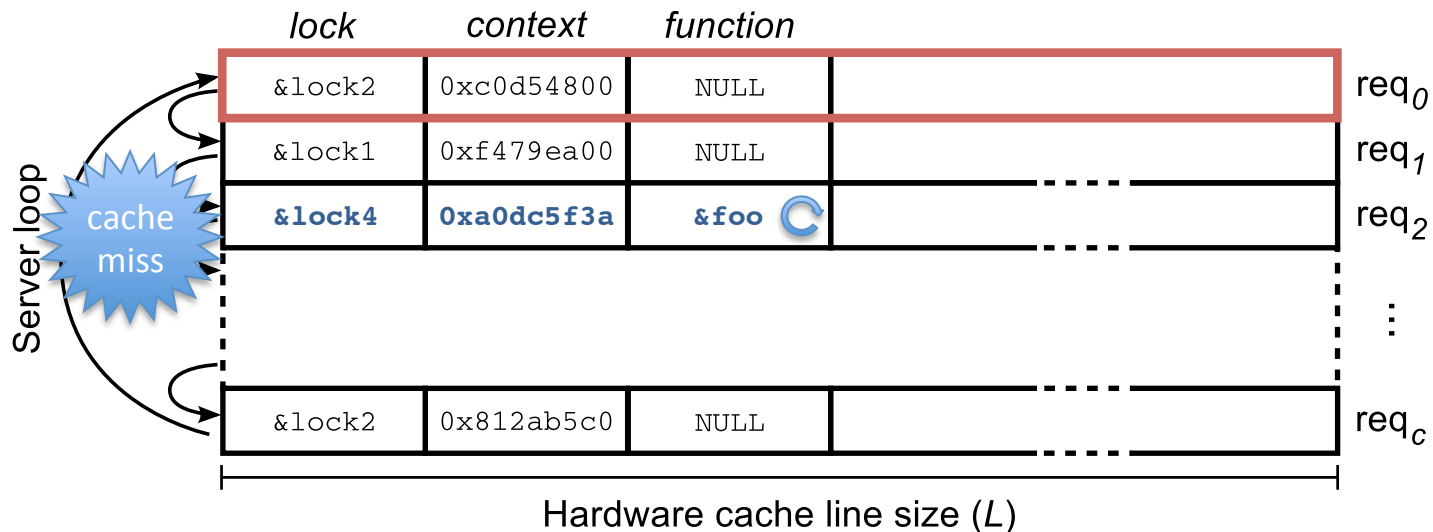


- Client fills the field and waits for the function to be reset
- Server loops across the fields

# Implementation: general idea

- Implementation based on cache line-sized mailboxes
- Three fields: lock, context, function

**Client thread 2 wants to execute a critical section protected by “lock4”**  
**Server continuously checks mailboxes and executes critical sections**

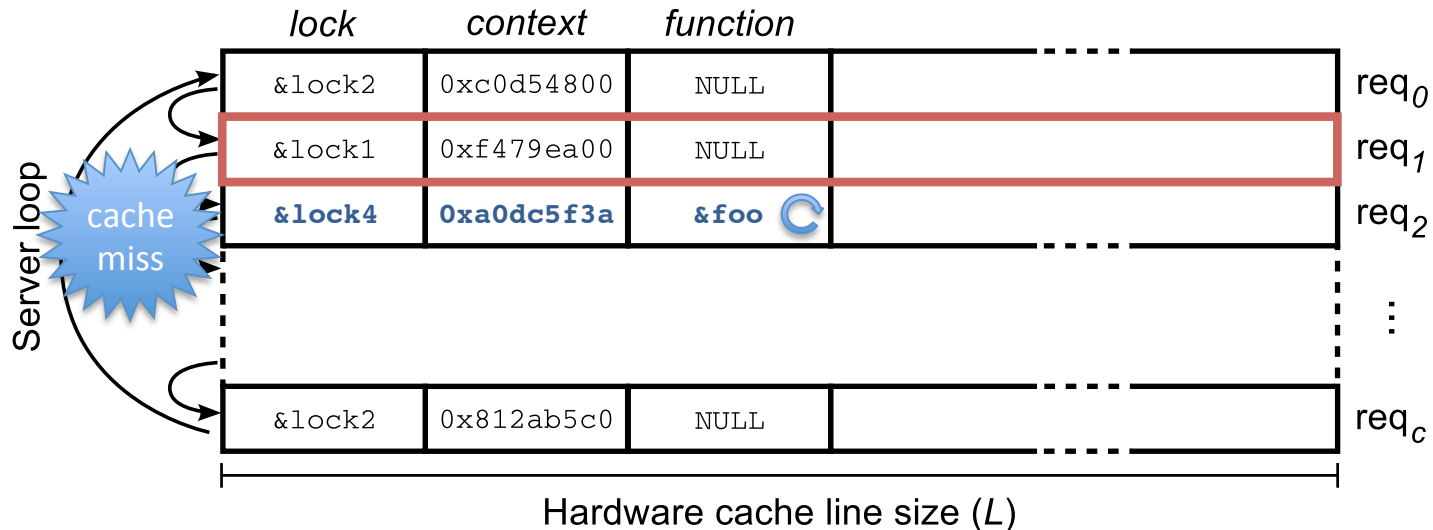


- Client fills the field and waits for the function to be reset
- Server loops across the fields

# Implementation: general idea

- Implementation based on cache line-sized mailboxes
- Three fields: lock, context, function

**Client thread 2 wants to execute a critical section protected by “lock4”**  
**Server continuously checks mailboxes and executes critical sections**

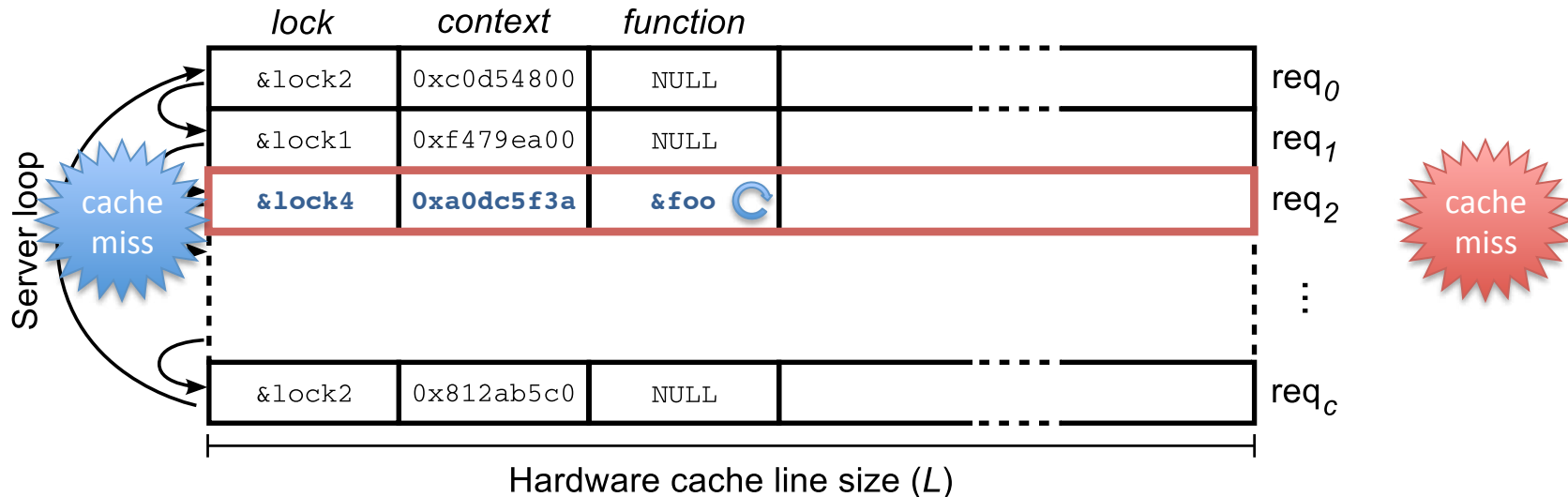


- Client fills the field and waits for the function to be reset
- Server loops across the fields

# Implementation: general idea

- Implementation based on cache line-sized mailboxes
- Three fields: lock, context, function

**Client thread 2 wants to execute a critical section protected by “lock4”**  
**Server continuously checks mailboxes and executes critical sections**

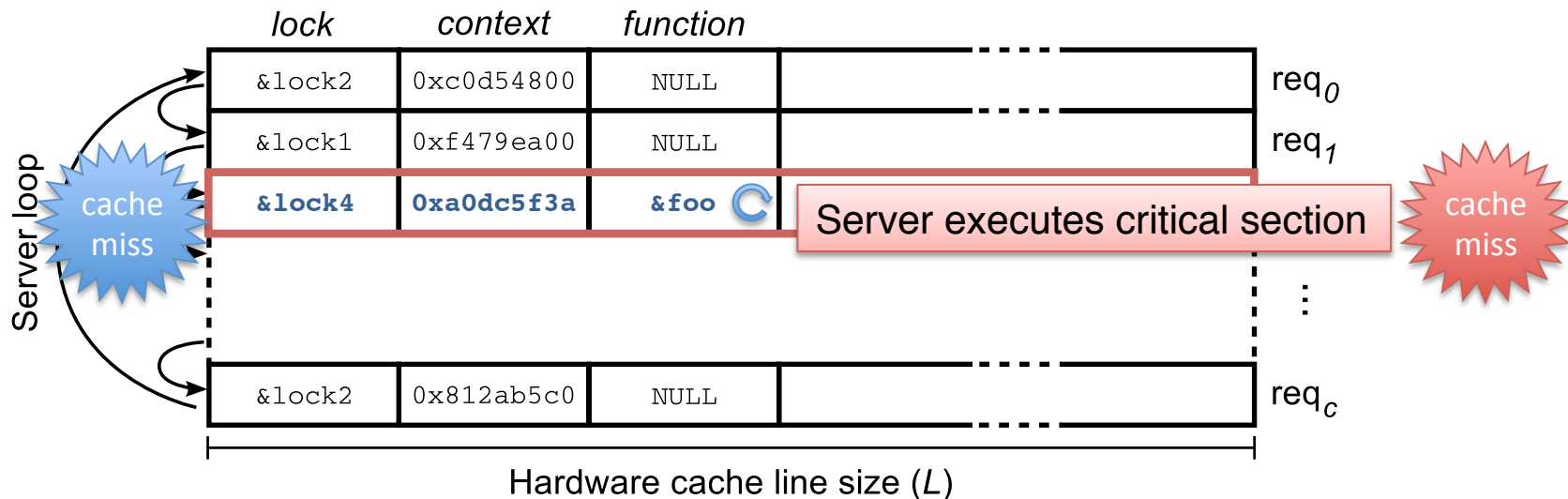


- Client fills the field and waits for the function to be reset
- Server loops across the fields

# Implementation: general idea

- Implementation based on cache line-sized mailboxes
- Three fields: lock, context, function

**Client thread 2 wants to execute a critical section protected by “lock4”**  
**Server continuously checks mailboxes and executes critical sections**

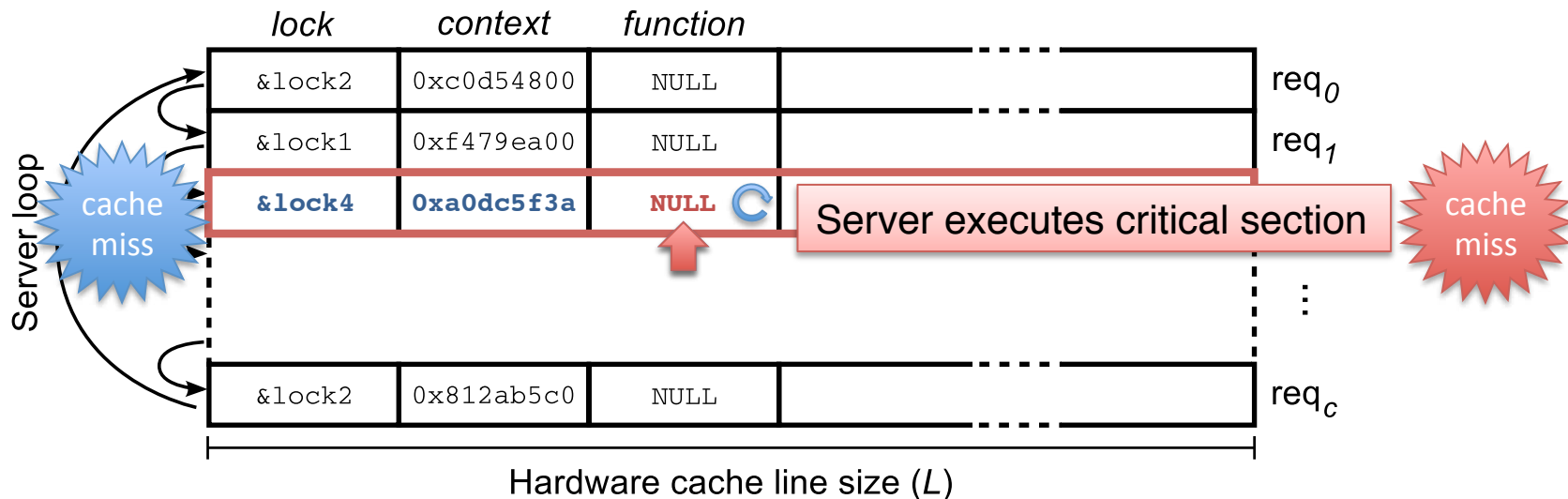


- Client fills the field and waits for the function to be reset
- Server loops across the fields

# Implementation: general idea

- Implementation based on cache line-sized mailboxes
- Three fields: lock, context, function

**Client thread 2 wants to execute a critical section protected by “lock4”**  
**Server continuously checks mailboxes and executes critical sections**

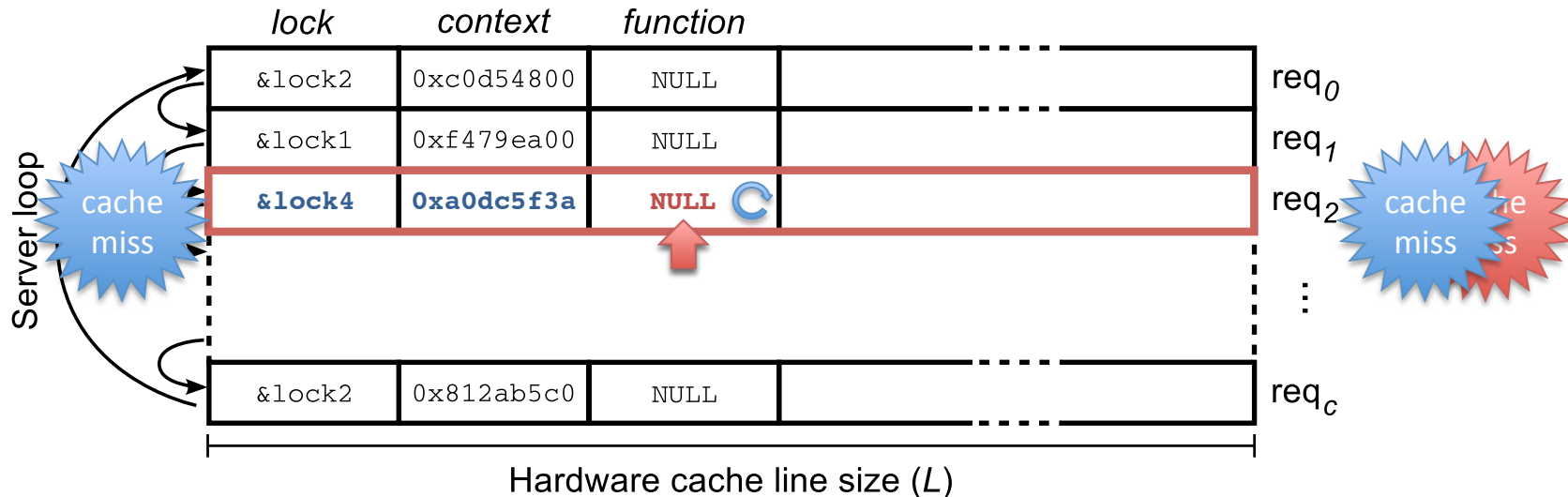


- Client fills the field and waits for the function to be reset
- Server loops across the fields

# Implementation: general idea

- Implementation based on cache line-sized mailboxes
- Three fields: lock, context, function

**Client thread 2 wants to execute a critical section protected by “lock4”**  
**Server continuously checks mailboxes and executes critical sections**

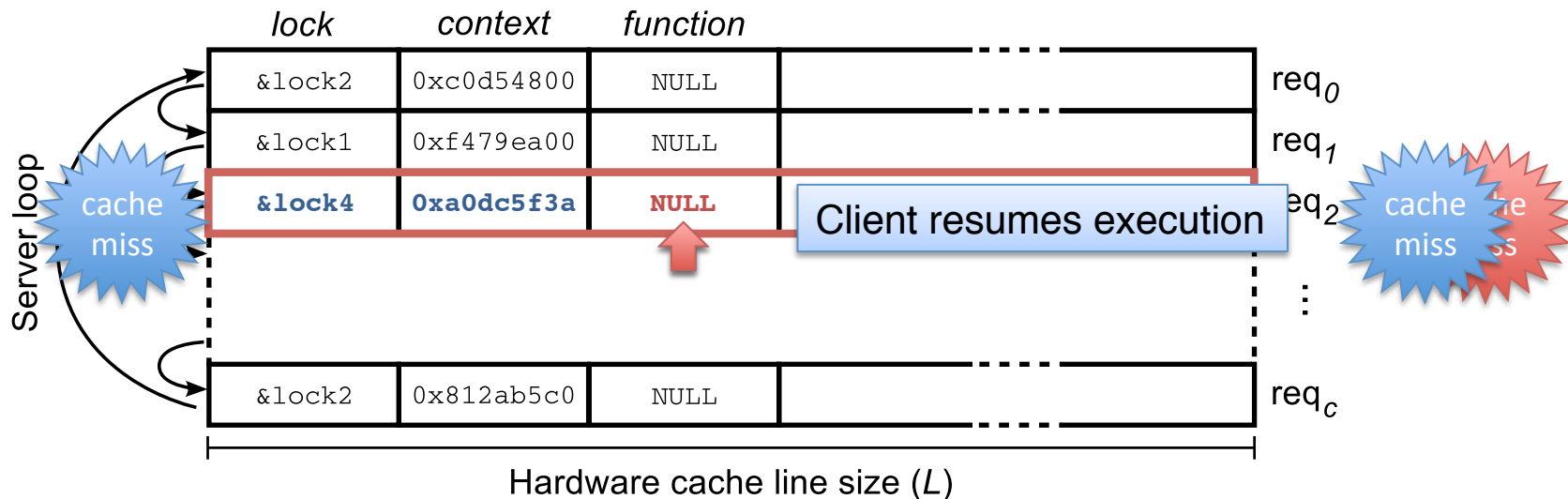


- Client fills the field and waits for the function to be reset
- Server loops across the fields

# Implementation: general idea

- Implementation based on cache line-sized mailboxes
- Three fields: lock, context, function

**Client thread 2 wants to execute a critical section protected by “lock4”**  
**Server continuously checks mailboxes and executes critical sections**



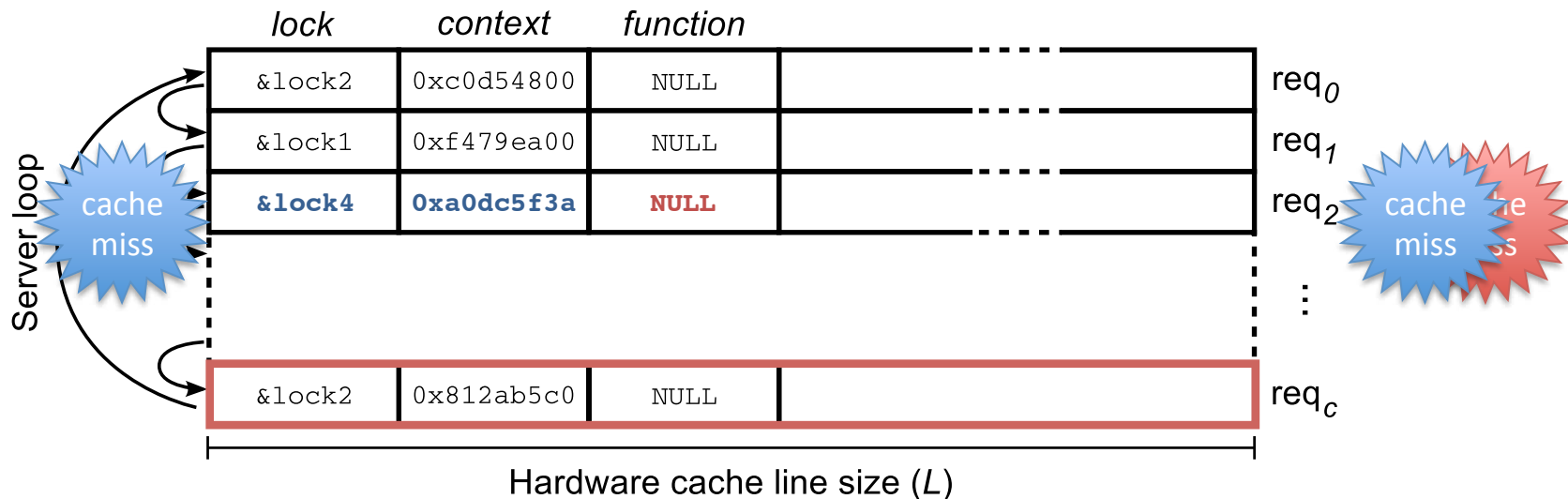
- Client fills the field and waits for the function to be reset
- Server loops across the fields



# Implementation: general idea

- Implementation based on cache line-sized mailboxes
- Three fields: lock, context, function

**Client thread 2 wants to execute a critical section protected by “lock4”**  
**Server continuously checks mailboxes and executes critical sections**

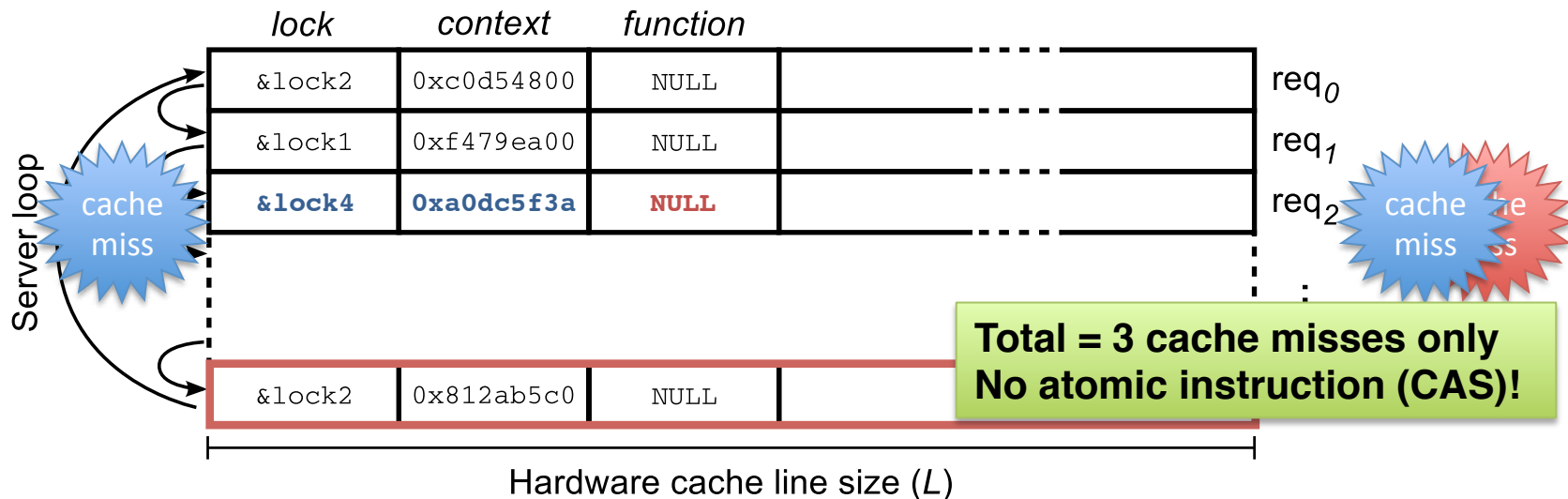


- Client fills the field and waits for the function to be reset
- Server loops across the fields

# Implementation: general idea

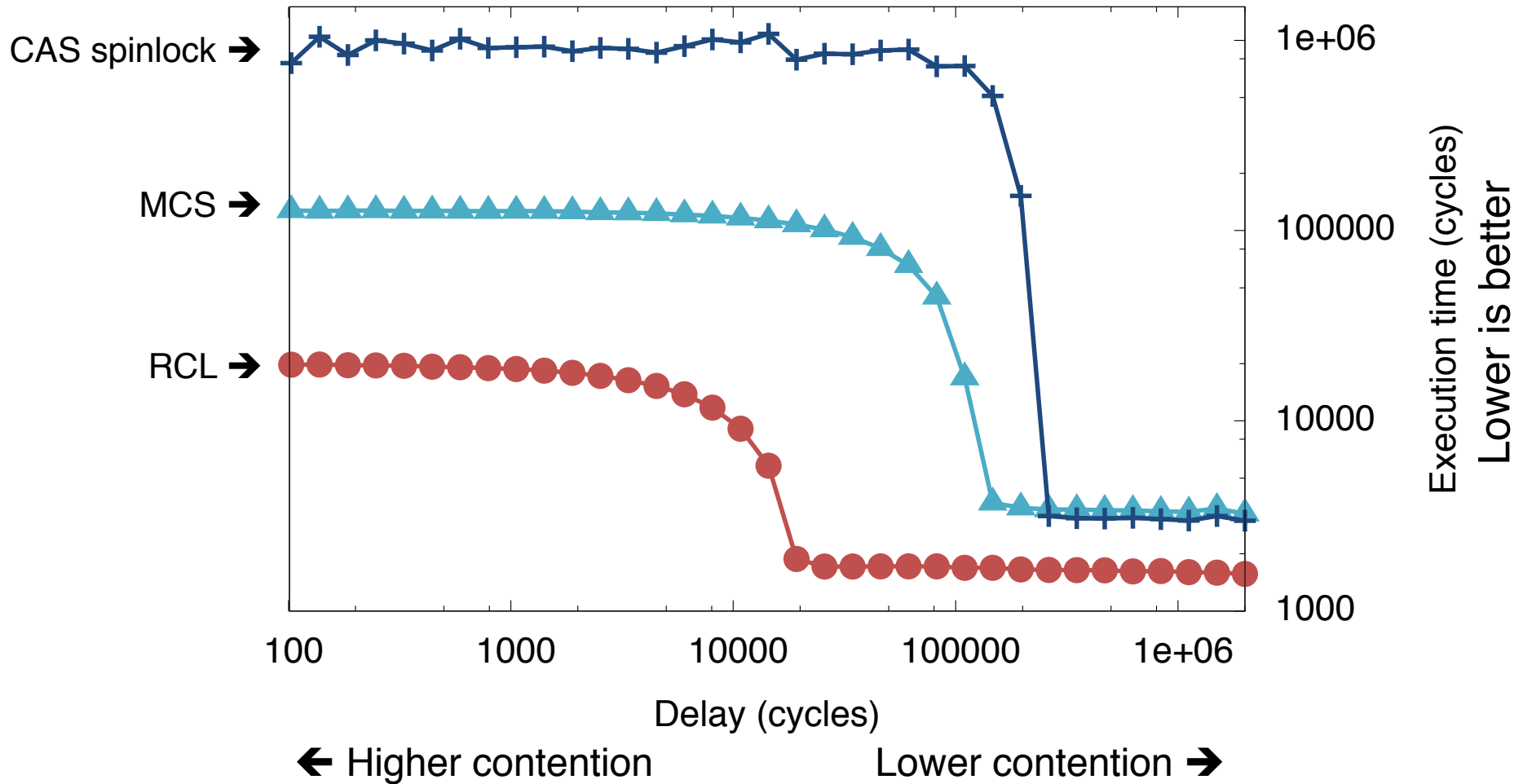
- Implementation based on cache line-sized mailboxes
- Three fields: lock, context, function

**Client thread 2 wants to execute a critical section protected by “lock4”**  
**Server continuously checks mailboxes and executes critical sections**

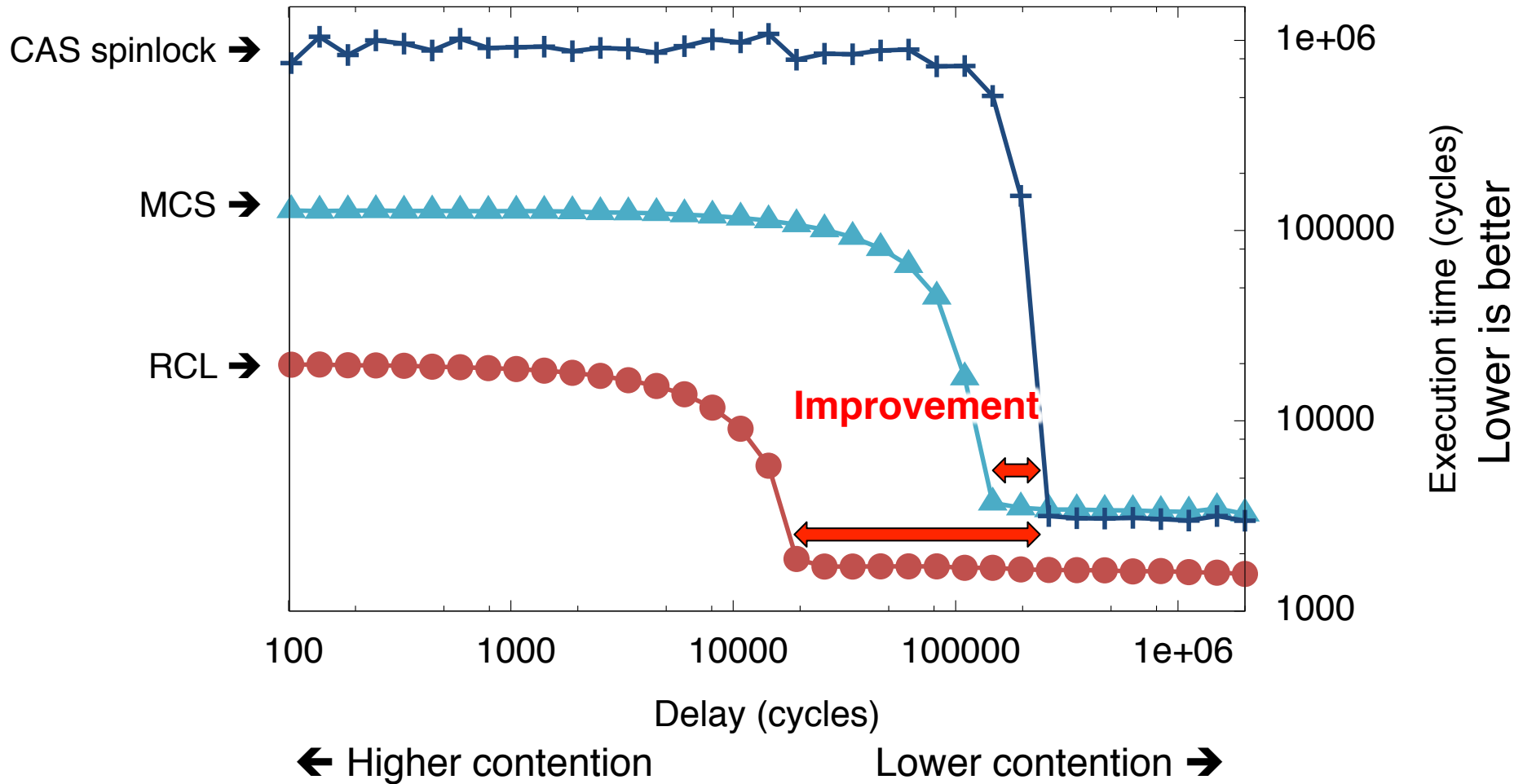


- Client fills the field and waits for the function to be reset
- Server loops across the fields

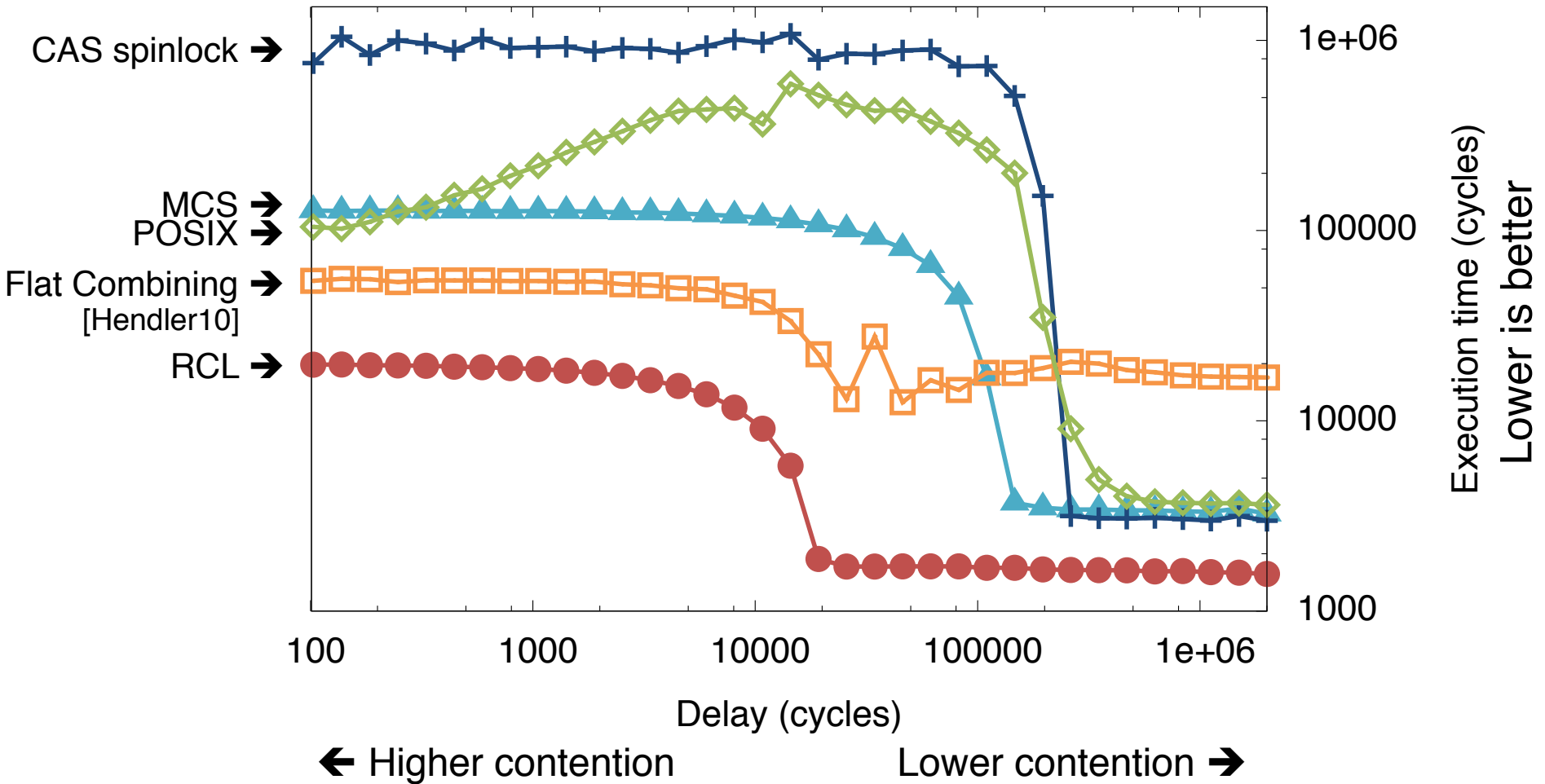
# Performance



# Performance



# Performance



# Using RCL in legacy applications (I)

## **RCL Runtime :**

- Handles blocking in critical sections (I/O, page faults...)
  - Pool of servicing threads on server
  - Able to service other (independent) critical sections when blocked
- Makes it possible to use condition variables (cond/wait)
  - Used by ~50% of applications that use POSIX locks in Debian 6.0.3

# Using RCL in legacy applications (2)

## **Reengineering:**

- Critical sections must be encapsulated into functions
  - Local variables sent as parameters (context)

# Using RCL in legacy applications (2)

## Reengineering:

```
void func(void) {  
    int a, b, x;  
    ...  
    a = ...;  
    ...  
    pthread_mutex_lock();  
    a = f(a);  
    f(b);  
    pthread_mutex_unlock();  
    ...  
}
```

```
struct context { int a, b };
```

```
void func(void) {  
    struct context c;  
    int x;  
    ...  
    c.a = ...;  
    ...  
    execute_rcl(__cs, &c);  
    ...  
}
```

```
void __cs(struct context *c) {  
    c->a = f(c->a)  
    f(c->b)  
}
```



# Using RCL in legacy applications (2)

## Reengineering:

```
void func(void) {  
    int a, b, x;  
    ...  
    a = ...;  
    ...  
    pthread_mutex_lock();  
    a = f(a);  
    f(b);  
    pthread_mutex_unlock();  
    ...  
}
```

```
struct context { int a, b };  
  
void func(void) {  
    struct context c;  
    int x;  
    ...  
    c.a = ...;  
    ...  
    execute_rcl(__cs, &c);  
    ...  
}  
  
void __cs(struct context *c) {  
    c->a = f(c->a);  
    f(c->b);  
}
```

# Using RCL in legacy applications (2)

## Reengineering:

```
void func(void) {  
    int a, b, x;  
    ...  
    a = ...;  
    ...  
    pthread_mutex_lock();  
    a = f(a);  
    f(b);  
    pthread_mutex_unlock();  
    ...  
}
```

```
struct context { int a, b };  
  
void func(void) {  
    struct context c;  
    int x;  
    ...  
    c.a = ...;  
    ...  
    execute_rcl(__cs, &c);  
    ...  
}  
  
void __cs(struct context *c) {  
    c->a = f(c->a);  
    f(c->b);  
}
```

# Using RCL in legacy applications (2)

## Reengineering:

```
void func(void) {  
    int a, b, x;  
    ...  
    a = ...;  
    ...  
    pthread_mutex_lock();  
    a = f(a);  
    f(b);  
    pthread_mutex_unlock();  
    ...  
}
```

```
struct context { int a, b };  
  
void func(void) {  
    struct context c;  
    int x;  
    ...  
    c.a = ...;  
    ...  
    execute_rcl(__cs, &c);  
    ...  
}  
  
void __cs(struct context *c) {  
    c->a = f(c->a);  
    f(c->b);  
}
```

# Using RCL in legacy applications (2)

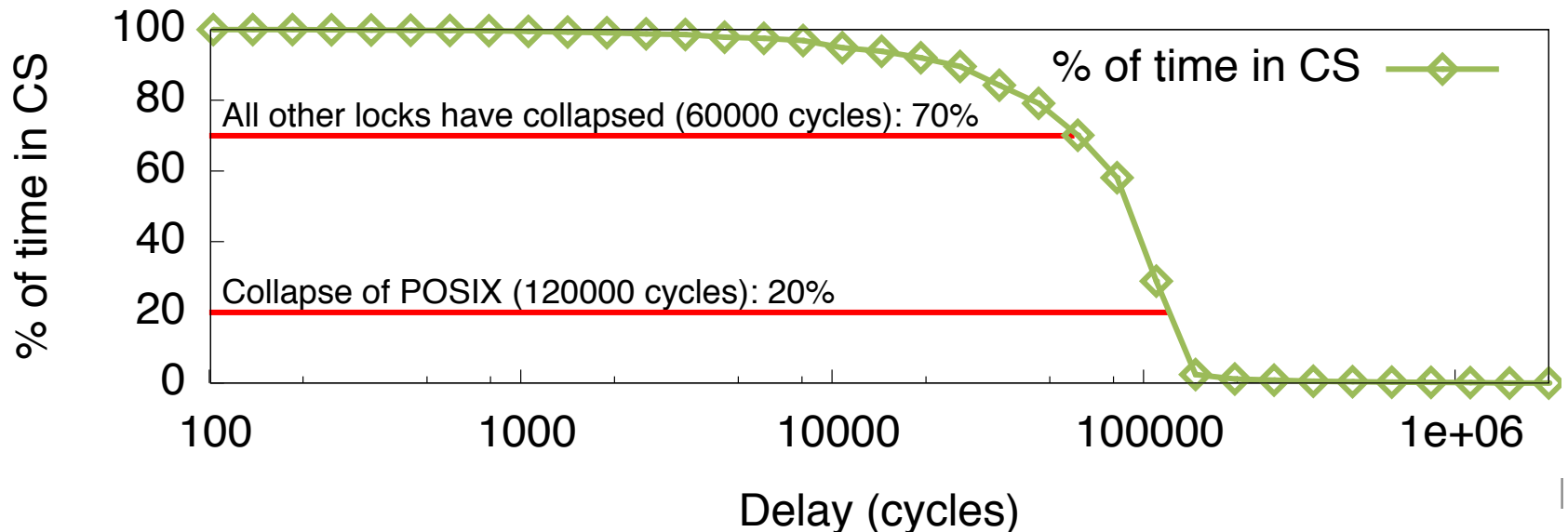
## **Reengineering:**

- Critical sections must be encapsulated into functions
  - Local variables sent as parameters (context)
- Tool to reengineer applications automatically
  - Possible to pick which locks use RCL
  - To avoid false serialization:  
Possible to pick which server(s) handle which lock(s).

# Using RCL in legacy applications (3)

## Profiling:

- Custom profiler to find good candidates
- Metric: time spent in critical sections
- Running the profiler on the microbenchmark shows that:
  - If time spent in CS > 20%, RCL is more efficient than POSIX locks
  - If time spent in CS > 70%, RCL is more efficient than all other locks

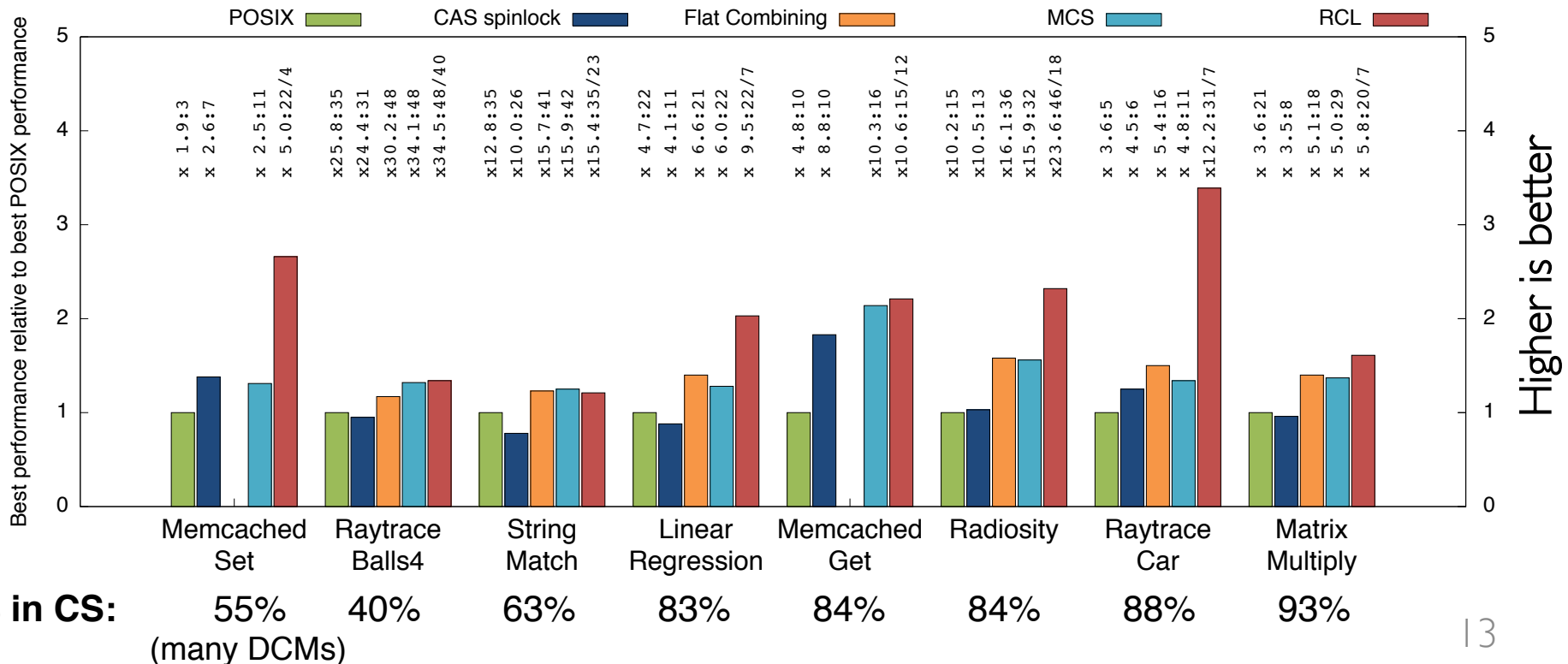


# Experiments

- Benchmarks (highly contended  $\Rightarrow$  70% time spent in CS):
  - **SPLASH-2 benchmark suite**
    - 3 applications out of 10 are highly contended
  - **Phoenix2 benchmark suite**
    - 3 applications out of 7 are highly contended
  - **Memcached**
    - Highly contended with the GET workload
  - **Berkeley DB / TPC-C**
    - Highly contended with 2 workloads (Order Status, Stock Level)

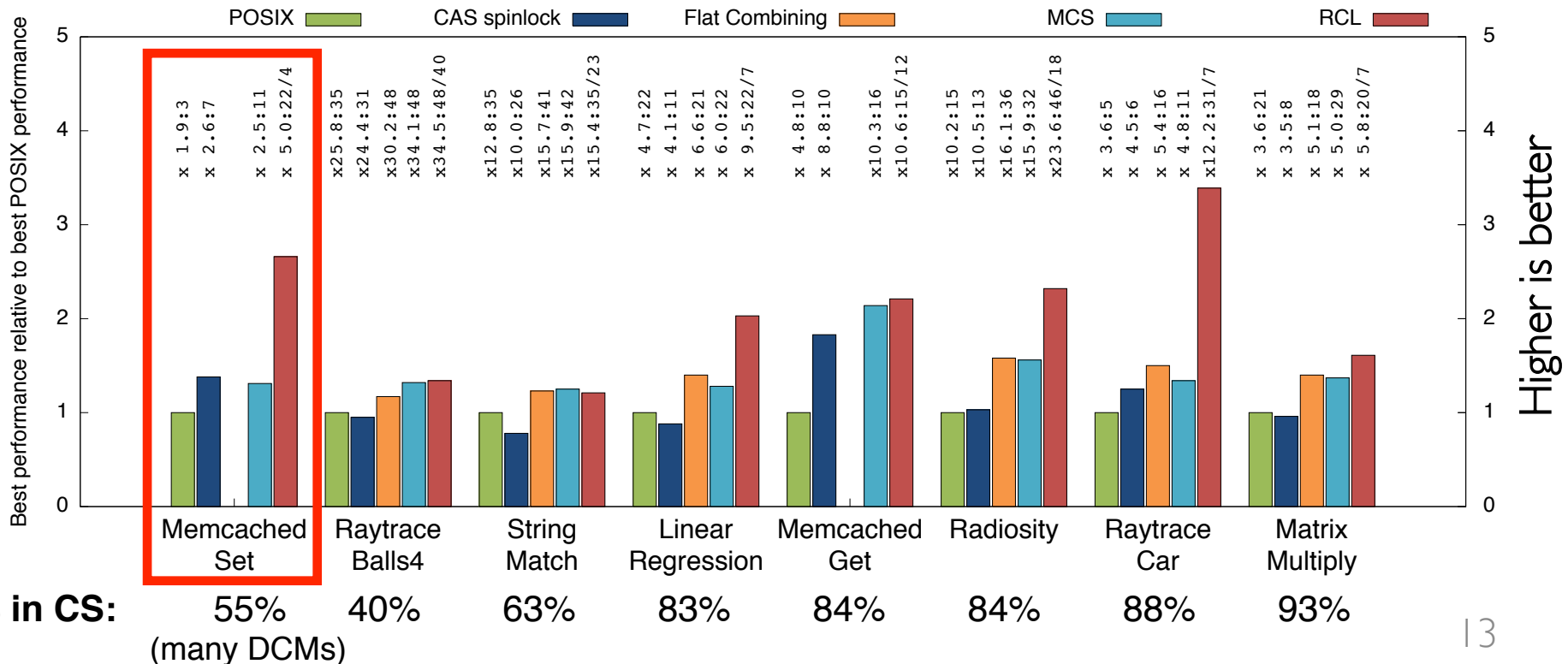
# Evaluation results (I)

- Better performance and scalability when time in CS > 70%
  - Performance improvement correlated with time in CS
- Only one or two locks replaced each time



# Evaluation results (I)

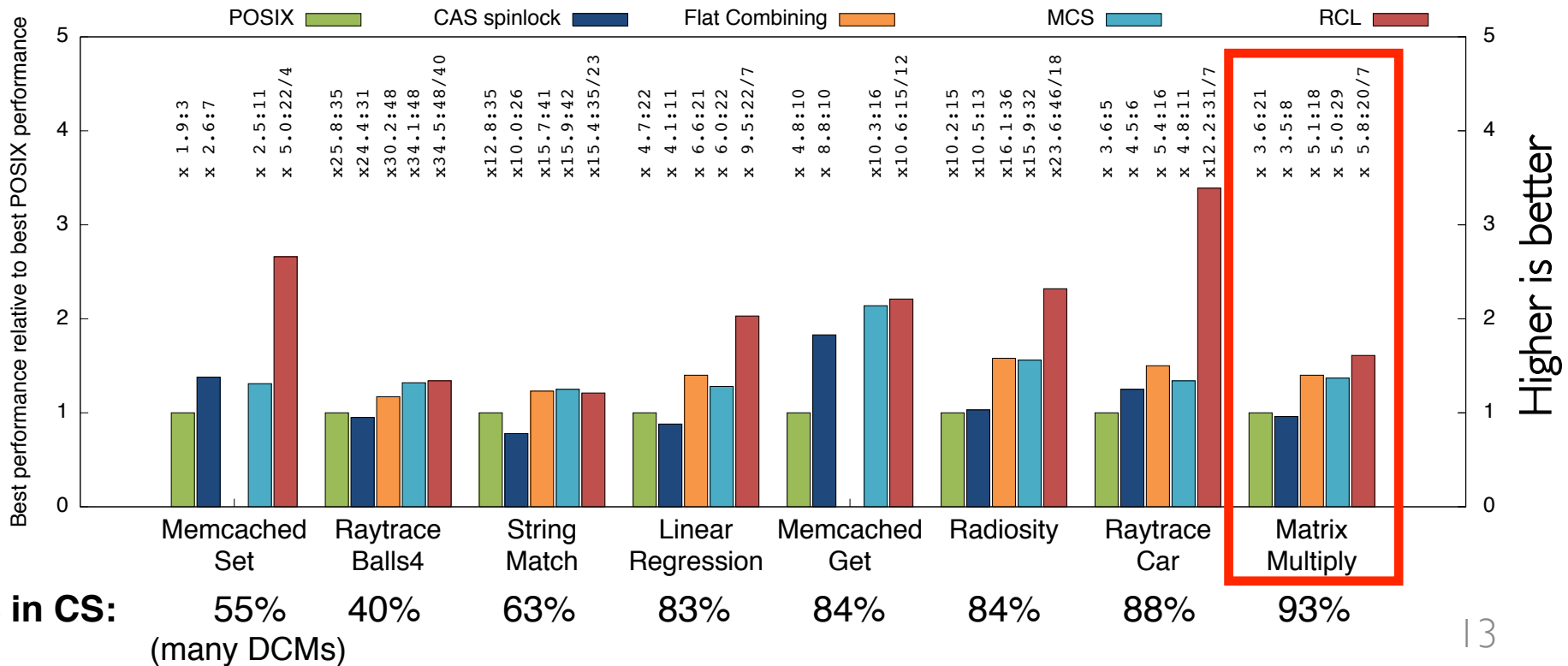
- Better performance and scalability when time in CS > 70%
  - Performance improvement correlated with time in CS
- Only one or two locks replaced each time





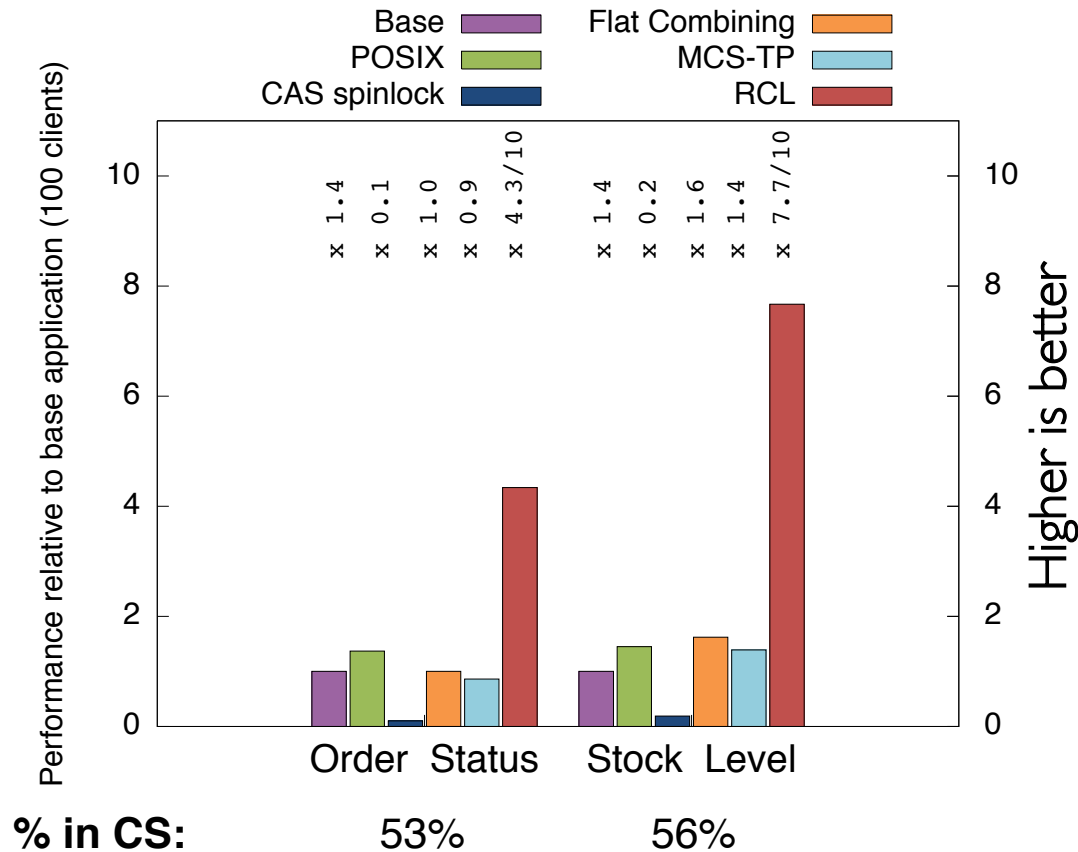
# Evaluation results (I)

- Better performance and scalability when time in CS > 70%
  - Performance improvement correlated with time in CS
- Only one or two locks replaced each time



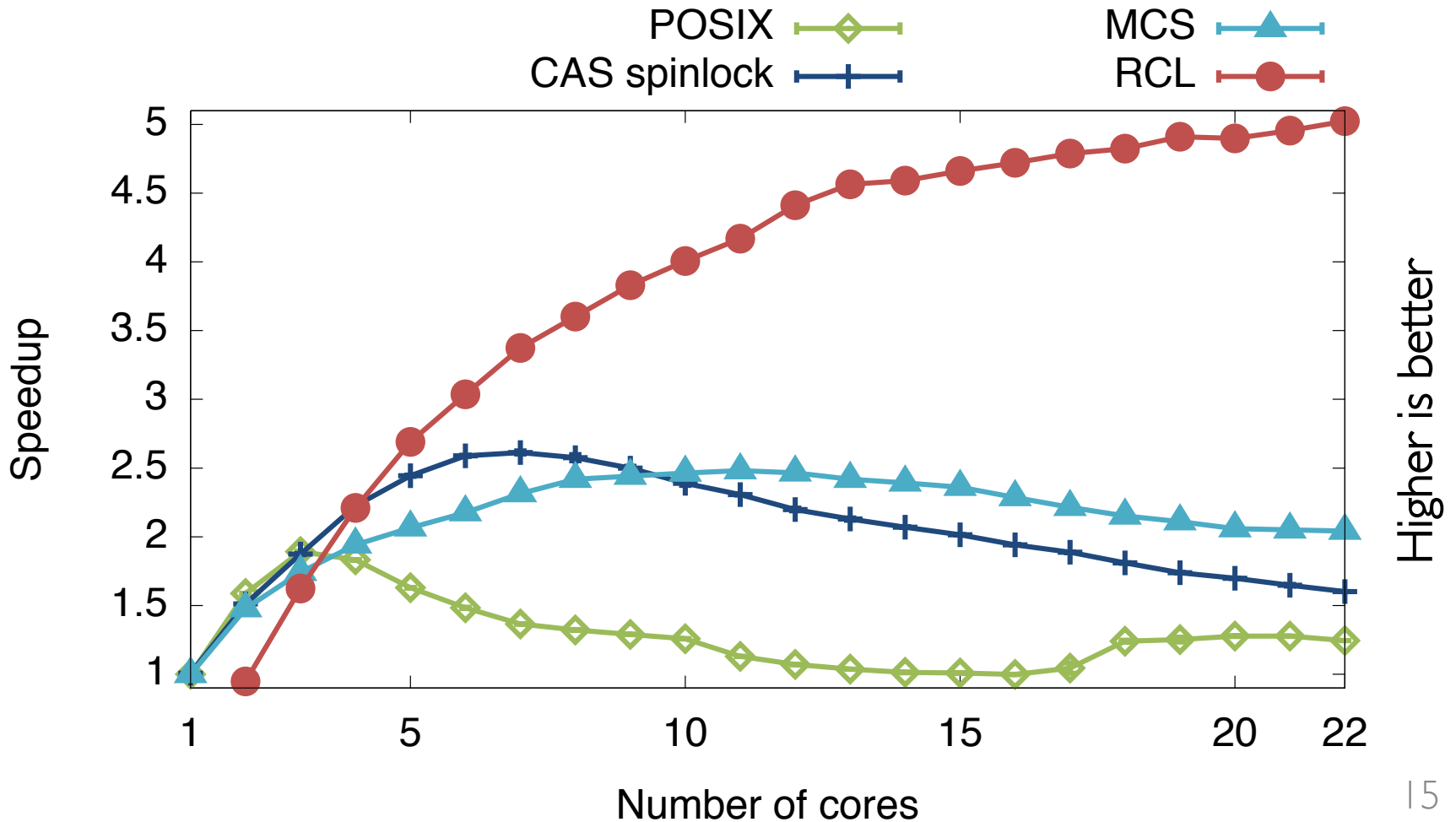
# Evaluation results (2)

- Berkeley DB with TPC-C (100 clients)
- Large gains, % in CS underestimated



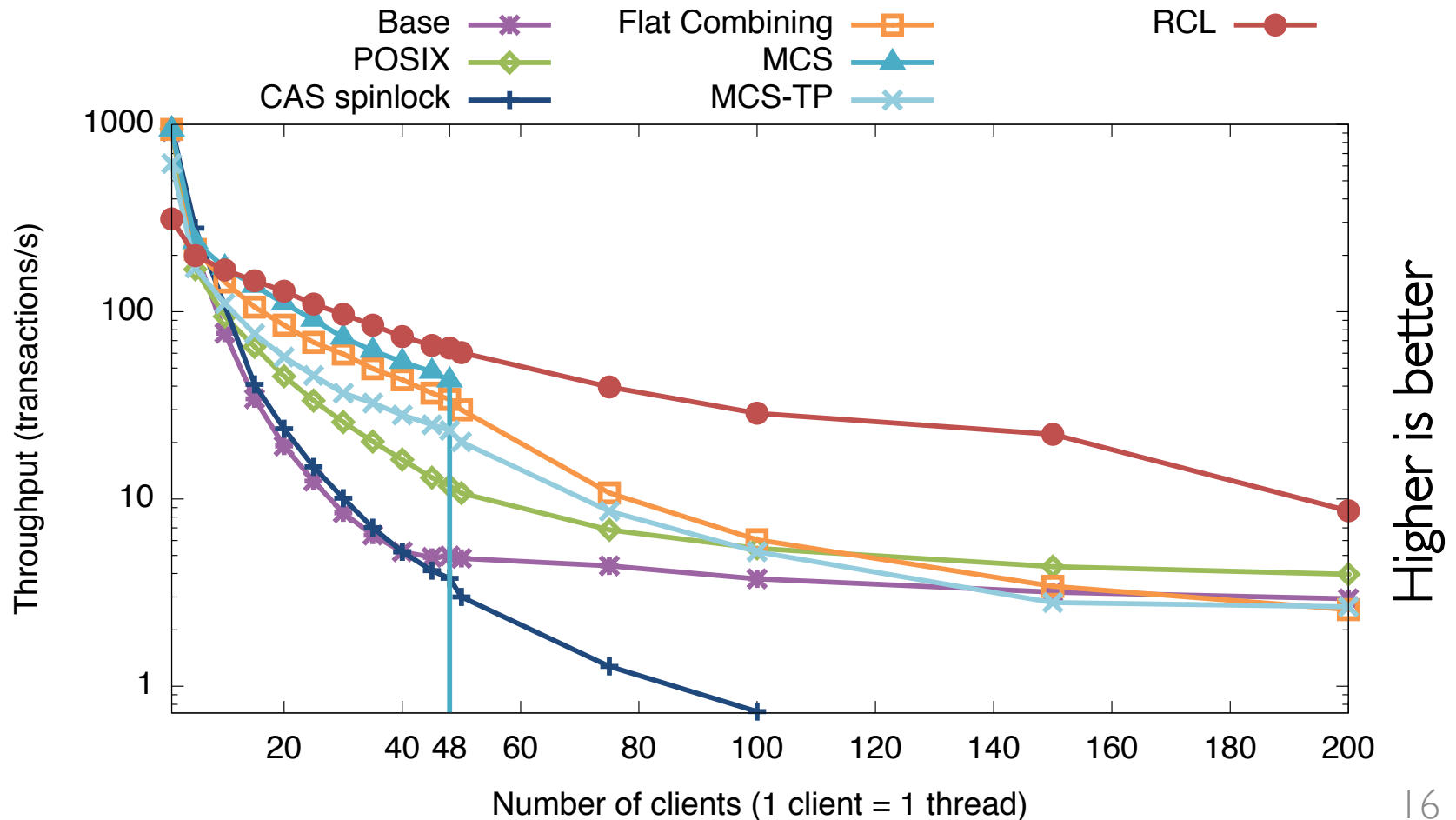
# RCL Scalability (I)

- Memcached, SET requests:



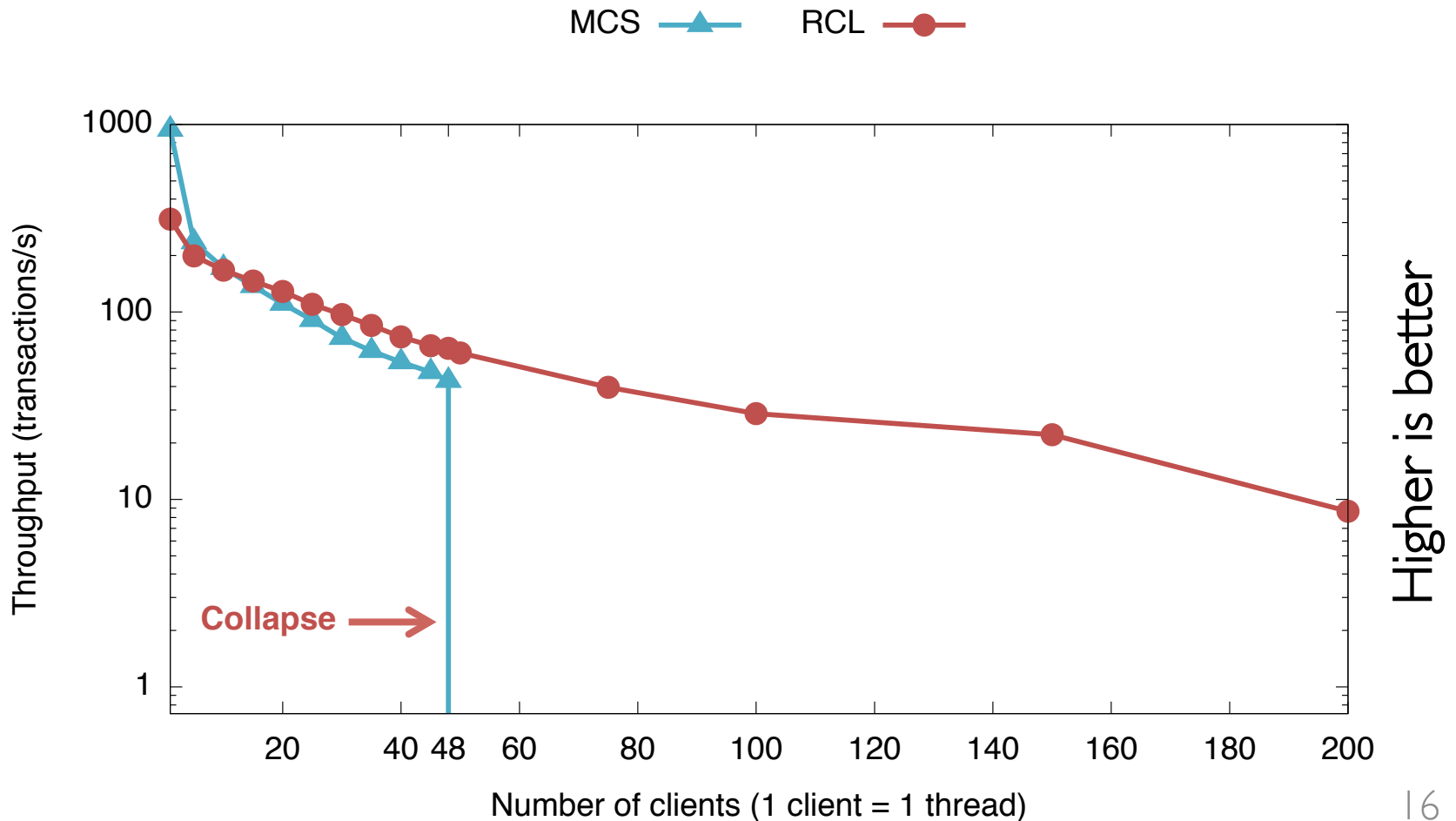
# RCL Scalability (2)

- Berkeley DB / TPC-C, Stock Level requests:



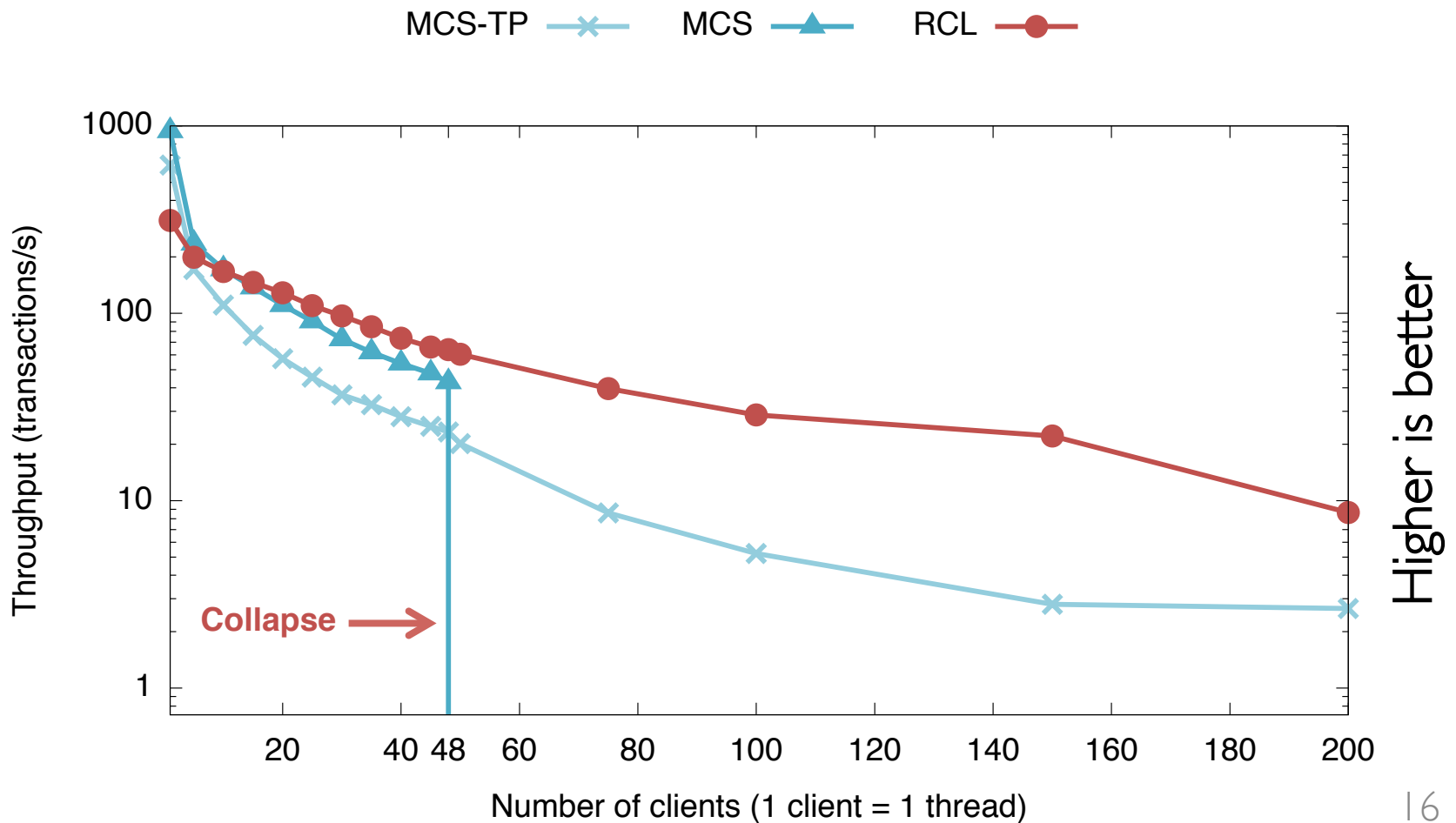
# RCL Scalability (2)

- Berkeley DB / TPC-C, Stock Level requests:



# RCL Scalability (2)

- Berkeley DB / TPC-C, Stock Level requests:



# Conclusion

- RCL reduces lock acquisition time and improves data locality
- Profiler to detect when RCL can be useful
- Tool to ease the transformation of legacy code
- Future work: adaptive RCL runtime
  - Dynamically switch between locking strategies
  - Load balancing between servers