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):

![Graph showing speedup vs. number of cores for Memcached/GET and Memcached/SET with collapses at certain points.](image)

Experiments run on a 48-core, “magny-cours” x86 AMD machine
Why?

• Critical sections = bottleneck on multicore architectures
• High contention ⇒ lock acquisition is costly
  – More cores ⇒ 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:

Critical sections access 5 cache lines each
Solution: designing better locks

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

![Graph showing improvements between CAS spinlock and MCS]

Critical sections access 5 cache lines each
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

![Diagram of Core Locking](image)
**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 ⇒ fewer cache misses

![Diagram of Remote Core Locking](image)
**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 ⇒ 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

<table>
<thead>
<tr>
<th>lock</th>
<th>context</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;lock2</td>
<td>0xc0d54800</td>
<td>NULL</td>
</tr>
<tr>
<td>&amp;lock1</td>
<td>0xf479ea00</td>
<td>NULL</td>
</tr>
<tr>
<td>&amp;lock4</td>
<td>0xa0dc5f3a</td>
<td>&amp;foo</td>
</tr>
<tr>
<td>&amp;lock2</td>
<td>0x812ab5c0</td>
<td>NULL</td>
</tr>
</tbody>
</table>

Hardware cache line size (L)

req₀  req₁  req₂  reqₙ
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

<table>
<thead>
<tr>
<th>lock</th>
<th>context</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;lock2</td>
<td>0xc0d54800</td>
<td>NULL</td>
</tr>
<tr>
<td>&amp;lock1</td>
<td>0xf479ea00</td>
<td>NULL</td>
</tr>
<tr>
<td>&amp;lock4</td>
<td>0xa0dc5f3a</td>
<td>&amp;foo</td>
</tr>
</tbody>
</table>

- 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

<table>
<thead>
<tr>
<th>lock</th>
<th>context</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;lock2</td>
<td>0xc0d54800</td>
<td>NULL</td>
</tr>
<tr>
<td>&amp;lock1</td>
<td>0xf479ea00</td>
<td>NULL</td>
</tr>
<tr>
<td>&amp;lock4</td>
<td>0xa0dc5f3a</td>
<td>&amp;foo</td>
</tr>
</tbody>
</table>

Client fills the field and waits for the function to be reset

Server loops across the fields

Hardware cache line size \((L)\)
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

<table>
<thead>
<tr>
<th>lock</th>
<th>context</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;lock2</td>
<td>0xd0d54800</td>
<td>NULL</td>
</tr>
<tr>
<td>&amp;lock1</td>
<td>0xf479ea00</td>
<td>NULL</td>
</tr>
<tr>
<td>&amp;lock4</td>
<td>0x0dc5f3a</td>
<td>NULL</td>
</tr>
<tr>
<td>&amp;lock2</td>
<td>0x812ab5c0</td>
<td>NULL</td>
</tr>
</tbody>
</table>

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

Server loop across the fields.

- 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

![Diagram showing cache lines and mailboxes with lock, context, and function fields.](image-url)
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

<table>
<thead>
<tr>
<th>lock</th>
<th>context</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;lock2</td>
<td>0xc0d54800</td>
<td>NULL</td>
</tr>
<tr>
<td>&amp;lock1</td>
<td>0xf479ea00</td>
<td>NULL</td>
</tr>
<tr>
<td>&amp;lock4</td>
<td>0xa0dc5f3a</td>
<td>NULL</td>
</tr>
</tbody>
</table>

- 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**

<table>
<thead>
<tr>
<th>lock</th>
<th>context</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;lock2</td>
<td>0x0d54800</td>
<td>NULL</td>
</tr>
<tr>
<td>&amp;lock1</td>
<td>0xf479ea00</td>
<td>NULL</td>
</tr>
<tr>
<td>&amp;lock4</td>
<td>0xa0dc5f3a</td>
<td>NULL</td>
</tr>
</tbody>
</table>

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

**Total = 3 cache misses only**

**No atomic instruction (CAS)!**
Performance

CAS spinlock ➔
MCS ➔
RCL ➔

Execution time (cycles)
Lower is better

Delay (cycles)
Higher contention ➔ Lower contention ➔
Performance

CAS spinlock ➔
MCS ➔
RCL ➔

Execution time (cycles)
Lower is better

Delay (cycles)
Higher contention ➜ Lower contention ➔

Improvement
Performance

CAS spinlock ➔
MCS ➔
POSIX ➔
Flat Combining ➔
[Hendler10]
RCL ➔

Execution time (cycles)
Lower is better

Delay (cycles)
Higher contention ➔
Lower contention ➔
Using RCL in legacy applications (1)

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)
Reengineering:

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

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

```c
void __cs(struct context *c) {
    c->a = f(c->a)
    f(c->b)
}
```
Using RCL in legacy applications (2)

Reengineering:

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

```c
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:

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

```c
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:

```c
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 (1)

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

% in CS:

<table>
<thead>
<tr>
<th>Application</th>
<th>In CS (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memcached Set</td>
<td>55%</td>
</tr>
<tr>
<td>Raytrace Balls4</td>
<td>40%</td>
</tr>
<tr>
<td>String Match</td>
<td>63%</td>
</tr>
<tr>
<td>Linear Regression</td>
<td>83%</td>
</tr>
<tr>
<td>Memcached Get</td>
<td>84%</td>
</tr>
<tr>
<td>Radiosity</td>
<td>84%</td>
</tr>
<tr>
<td>Raytrace Car</td>
<td>88%</td>
</tr>
<tr>
<td>Matrix Multiply</td>
<td>93%</td>
</tr>
</tbody>
</table>
Evaluation results (1)

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

<table>
<thead>
<tr>
<th>% in CS:</th>
<th>55%</th>
<th>40%</th>
<th>63%</th>
<th>83%</th>
<th>84%</th>
<th>84%</th>
<th>88%</th>
<th>93%</th>
</tr>
</thead>
<tbody>
<tr>
<td>(many DCMs)</td>
<td>Memcached Set</td>
<td>Raytrace Balls4</td>
<td>String Match</td>
<td>Linear Regression</td>
<td>Memcached Get</td>
<td>Radiosity</td>
<td>Raytrace Car</td>
<td>Matrix Multiply</td>
</tr>
<tr>
<td>Higher is better</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Graph showing performance relative to best POSIX performance](image)
**Evaluation results (1)**

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

% in CS:  
- 55% (many DCMs)  
- 40%  
- 63%  
- 83%  
- 84%  
- 84%  
- 88%  
- 93%
Evaluation results (2)

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

Performance relative to base application (100 clients)

<table>
<thead>
<tr>
<th></th>
<th>Base</th>
<th>POSIX</th>
<th>CAS spinlock</th>
<th>Flat Combining</th>
<th>MCS-TP</th>
<th>RCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Order</td>
<td>x 1.4</td>
<td>x 0.1</td>
<td>x 1.0</td>
<td>x 4.3/10</td>
<td>x 1.4</td>
<td>x 0.2</td>
</tr>
<tr>
<td>Status</td>
<td>x 1.0</td>
<td>x 0.9</td>
<td>x 0.9</td>
<td>x 0.2</td>
<td>x 1.0</td>
<td>x 1.0</td>
</tr>
<tr>
<td>Stock</td>
<td>x 1.4</td>
<td>x 0.2</td>
<td>x 1.6</td>
<td>x 1.4</td>
<td>x 7.7/10</td>
<td>x 7.7/10</td>
</tr>
<tr>
<td>Level</td>
<td>x 1.4</td>
<td>x 0.2</td>
<td>x 1.6</td>
<td>x 1.4</td>
<td>x 7.7/10</td>
<td>x 7.7/10</td>
</tr>
</tbody>
</table>

% in CS: 53% 56%
RCL Scalability (1)

• Memcached, SET requests:

![Graph showing speedup vs. number of cores for different locks.](image-url)
**RCL Scalability (2)**

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

![Graph showing throughput (transactions/s) against number of clients (1 client = 1 thread). The x-axis represents the number of clients ranging from 20 to 200, while the y-axis represents throughput ranging from 1000 to 0. The graph compares different locking mechanisms: Base, POSIX, CAS spinlock, Flat Combining, MCS, MCS-TP, and RCL. Higher is better. The RCL Scalability (2) indicates the performance improvement.]

---

Higher is better
RCL Scalability (2)

- Berkeley DB / TPC-C, Stock Level requests:
RCL Scalability (2)

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

![Graph showing throughput vs number of clients]

Higher is better

Throughput (transactions/s)

Number of clients (1 client = 1 thread)
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