Towards More Scalable Mutual Exclusion for Multicore Architectures

Vers des mécanismes d’exclusion mutuelle plus efficaces pour les architectures multi-cœur

Jean-Pierre Lozi

PhD thesis defense, Université Pierre et Marie Curie
Regal/Whisper team, LIP6/INRIA
Outline

- **Context:** multicore architectures
- State of the art: locks
- Contribution: Remote Core Locking
- Evaluation
- Perspectives and conclusion
Context: multicore architectures

- Decades of increasing CPU clock speeds
- Since early 2000’s, problems with power consumption/dissipation
- Increasing numbers of cores to keep increasing processing power
  - Possible because number of transistors keeps increasing

![Intel CPU Trends](image)

- # transistors
- Clock speed
- Power Consumption
- Ratio power/speed
Problem with multicore: 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 with multicore: 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?

- Bottleneck = critical sections, protected by locks
- High contention ⇒ lock acquisition is costly
  - More cores ⇒ higher contention

* Including lock acquisition time
Why?

- Bottleneck = critical sections, protected by locks
- High contention $\Rightarrow$ lock acquisition is costly
  - More cores $\Rightarrow$ higher contention

- Two possible solutions:
  - Redesign applications (fine-grained locking)
    - Costly (millions of lines of legacy code)
  - Design better locks
Why?

- Bottleneck = critical sections, protected by locks
- High contention $\Rightarrow$ lock acquisition is costly
  - More cores $\Rightarrow$ higher contention

- Two possible solutions:
  - Redesign applications (fine-grained locking)
    - Costly (millions of lines of legacy code)
  - Design better locks
Designing better locks

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

Critical sections access 5 cache lines each

[Mellor-Crummey ASPLOS’91]
Designing better locks

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

  Critical sections access 5 cache lines each

[Mellor-Crummey ASPLOS’91]
Outline

• Context: multicore architectures
• **State of the art: locks**
• Contribution: Remote Core Locking
• Evaluation
• Perspectives and conclusion
State of the art

• Spinlocks
• Blocking locks
• Queue locks (MCS, CLH)  
  [Mellor-Crummey ASPLOS’91, Craig TR’93, Hagersten IPPS’94]
• Flat combining  
  [Hendler SPAA’10]
Spinlocks

- Spinlocks
  - Busy-wait, trying to set a lock variable with an atomic instruction
  - Contention when all threads try to set that variable concurrently!

```plaintext
function lock(boolean *lock)
    while !compare_and_swap(lock, false, true) do
        ;

function unlock(boolean *lock)
    *lock = false;
```
Spinlocks

- Busy-wait, trying to set a lock variable with an atomic instruction
- Contention when all threads try to set that variable concurrently!

```python
function lock(boolean *lock)
    while !compare_and_swap(lock, false, true) do
    ;

function unlock(boolean *lock)
    *lock = false;
```
Spinlocks

- Spinlocks
  - Busy-wait, trying to set a lock variable with an atomic instruction
  - Contention when all threads try to set that variable concurrently!

```c
function lock(boolean *lock) {
    while !compare_and_swap(lock, false, true) do ;
}

function unlock(boolean *lock) {
    *lock = false;
}
```
Spinlocks

- Spinlocks
  - Busy-wait, trying to set a lock variable with an atomic instruction
  - Contention when all threads try to set that variable concurrently!
- Cost of all threads concurrently writing to a single variable:
  - Up to 125 times slower when all hardware threads used!
Blocking locks

• Try to acquire lock; in case of failure, sleep
• Does not waste CPU resources
• Context switches needed between each acquisition:
  not very reactive

```c
function lock(boolean *lock)
    while !compare_and_swap(lock, false, true) do
        yield();

function unlock(boolean *lock)
    *lock = false;
```
Blocking locks

• Try to acquire lock; in case of failure, sleep
• Does not waste CPU resources
• Context switches needed between each acquisition: not very reactive
• Very frequently used because works with only one core
  – The “legacy” lock
  – POSIX locks are blocking locks
Queue locks

• Example: MCS [Mellor-Crummey ASPLOS'91]
• Idea: threads enqueue themselves in a list
  – One synchronization variable per thread instead of global tail
Queue locks

- Example: MCS [Mellor-Crummey ASPLOS'91]
- Idea: threads enqueue themselves in a list
  - One synchronization variable per thread instead of global

```
spn?  next
Ignored
Thread 1’s node
```

Atomic

Tail
Queue locks

- Example: MCS [Mellor-Crummey ASPLOS'91]
- Idea: threads enqueue themselves in a list
  - One synchronization variable per thread instead of global

Thread 1 executes "critical section"
Queue locks

- Example: MCS [Mellor-Crummey ASPLOS'91]
- Idea: threads enqueue themselves in a list
  - One synchronization variable per thread instead of global

Thread 1 executes "critical section"
Queue locks

- **Example:** MCS [Mellor-Crummey ASPLOS'91]
- **Idea:** threads enqueue themselves in a list
  - One synchronization variable per thread instead of global

Thread 1 executes "critical section"

Thread 1's node

Thread 2's node
Queue locks

- Example: MCS [Mellor-Crummey ASPLOS'91]
- Idea: threads enqueue themselves in a list
  - One synchronization variable per thread instead of global

Thread 1 executes "critical section"
Queue locks

- Example: MCS [Mellor-Crummey ASPLOS'91]
- Idea: threads enqueue themselves in a list
  - One synchronization variable per thread instead of global

Thread 1 executes "critical section"

- Thread 1’s node: spin? Ignored, next
- Thread 2’s node: spin? True, next
- Thread 3’s node: spin? True, next (tail)
Queue locks

- Example: MCS [Mellor-Crummey ASPLOS’91]
- Idea: threads enqueue themselves in a list
  – One synchronization variable per thread instead of global

Thread 2 executes "critical section"
Queue locks

- Example: MCS [Mellor-Crummey ASPLOS'91]
- Idea: threads enqueue themselves in a list
  - One synchronization variable per thread instead of global

```
Thread 3 executes critical section
```

```
Thread 3's node

<table>
<thead>
<tr>
<th>spin?</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td></td>
</tr>
</tbody>
</table>
```

tail
Queue locks

- Example: MCS [Mellor-Crummey ASPLOS'91]
- Idea: threads enqueue themselves in a list
  - One synchronization variable per thread instead of global
Flat Combining [Hendler SPAA’10]

- Threads enqueue critical sections (functions) in list
- Occasionally, a thread becomes a “combiner”
  - Executes all pending critical sections
  - Possibly merging critical sections with fast sequential algorithm
- Uses a global spinlock, need to clean up the list
Flat Combining [Hendler SPAA’10]

- Threads enqueue critical sections (functions) in list
- Occasionally, a thread becomes a “combiner”
  - Executes all pending critical sections
  - Possibly merging critical sections with fast sequential algorithm
- Uses a global spinlock, need to clean up the list

T3 acquires the global lock
Flat Combining [Hendler SPAA’10]

- Threads enqueue critical sections (functions) in list
- Occasionally, a thread becomes a “combiner”
  - Executes all pending critical sections
  - Possibly merging critical sections with fast sequential algorithm
- Uses a global spinlock, need to clean up the list

T3 executes its CS and the following ones
Flat Combining [Hendler SPAA’10]

- Threads enqueue critical sections (functions) in list
- Occasionally, a thread becomes a “combiner”
  - Executes all pending critical sections
  - Possibly merging critical sections with fast sequential algorithm
- Uses a global spinlock, need to clean up the list

T3 executes its CS and the following ones
Flat Combining [Hendler SPAA’10]

- Threads enqueue critical sections (functions) in list
- Occasionally, a thread becomes a “combiner”
  - Executes all pending critical sections
  - Possibly merging critical sections with fast sequential algorithm
- Uses a global spinlock, need to clean up the list

T3 executes its CS and the following ones
Flat Combining [Hendler SPAA’10]

- Threads enqueue critical sections (functions) in list
- Occasionally, a thread becomes a “combiner”
  - Executes all pending critical sections
  - Possibly merging critical sections with fast sequential algorithm
- Uses a global spinlock, need to clean up the list
Lock performance assessment

Execution time (cycles)

- POSIX
- MCS
- Spinlock
- Flat Comb.

CAS spinlock ➔

MCS ➔

Blocking locks ➔

Flat Combining ➔

Lower is better

Higher contention ➔

Delay

Lower contention ➔
Outline

• Context: multicore architectures
• State of the art: locks
• **Contribution: Remote Core Locking**
• Evaluation
• Perspectives and conclusion
Contribution: Remote Core Locking

Objective: create the fastest possible lock algorithm under contention
Contribution: Remote Core Locking

Objective: create the fastest possible lock algorithm under contention

How?
**Contribution: Remote Core Locking**

**Objective:** create the fastest possible lock algorithm under contention

*How?*

![Diagram](image)
Objective: create the fastest possible lock algorithm under contention

How?
**Objective:** create the fastest possible lock algorithm under contention

**How?**

**Contribution: Remote Core Locking**

<table>
<thead>
<tr>
<th>Time</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CS2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CS3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Critical path:**

Transfers of lock ownership
Contribution: Remote Core Locking

Objective: create the fastest possible lock algorithm under contention

How? By shortening the critical path as much as possible
Contribution: Remote Core Locking

What makes the critical path longer than needed?
Contribution: Remote Core Locking

What lengthens the critical path?

1) Long transfers of lock ownership

<table>
<thead>
<tr>
<th>Time</th>
<th>CS1</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>CS2</th>
<th>CS3</th>
</tr>
</thead>
</table>

Transfers of lock ownership
Contribution: Remote Core Locking

What lengthens the critical path?

1) Long transfers of lock ownership

Global spin (Spinlock), context switch (Blocking lock), remote thread wakeup (MCS), global lock acq. (Flat Comb.),…
Contribution: Remote Core Locking

What lengthens the critical path?

2) Poor data locality in critical sections
Contribution: Remote Core Locking

What lengthens the critical path?

2) Poor data locality in critical sections
Contribution: Remote Core Locking

Solution: Remote Core Locking

Dedicate a core for executing critical sections
**Contribution: Remote Core Locking**

**Solution:** Remote Core Locking

*Dedicate a core for executing critical sections*

T1  T2  T3

CS1  CS2  CS3

*Time*

Server core
**Contribution: Remote Core Locking**

**Solution:** Remote Core Locking

*Dedicate a core for executing critical sections*
Solution: Remote Core Locking

Dedicate a core for executing critical sections
Problem: what to do when using several locks?
- False serialization, bad for performance

If too much contention: simply add more servers
- Not a problem, because RCL only targets contended locks
- Typically only a handful of them
False serialization

• Problem: what to do when using several locks?
  – False serialization, bad for performance

• If too much contention: simply add more servers
  – Not a problem, because RCL only targets contended locks
  – Typically only a handful of them
Implementation: general idea

- Communication 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 (fair)

<table>
<thead>
<tr>
<th>lock</th>
<th>context</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

Server loop

Hardware cache line size ($L$)

[Diagram showing server loops and client requests]
Implementation: general idea

- Communication 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 (fair)
Implementation: general idea

- Communication 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 (fair)
Implementation: general idea

- Communication 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 (fair)
Implementation: general idea

- Communication 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 (fair)
Implementation: general idea

- Communication 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 (fair)
Implementation: general idea

- Communication 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 (fair)
Implementation: general idea

- Communication 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 (fair)
### Implementation: general idea

- Communication 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>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</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 (fair)
Implementation: general idea

- Communication 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

Server loop

- Client fills the field and waits for the function to be reset
- Server loops across the fields (fair)
Implementation: general idea

- Communication 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 (fair)
Implementation: general idea

- Communication 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 (fair)
Implementation: general idea

- Communication 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>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>&amp;lock4</td>
<td>0xa0dc5f3a</td>
<td>NULL</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

Client resumes execution

- Client fills the field and waits for the function to be reset
- Server loops across the fields (fair)
Implementation: general idea

- Communication 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>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</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 (fair)
Implementation: general idea

- Communication 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 (fair)
RCL Performance

CAS spinlock

MCS

Blocking locks

Flat Combining

RCL

Execution time (cycles)

Lower is better

Higher contention

Delay

Lower contention
RCL Performance

Execution time (cycles)

POSIX  Spinlock  MCS  Flat Comb.  CC-Synch  DSM-Synch  RCL

CAS spinlock

MCS

Combining locks { RCL

Blocking locks

Lower is better

Higher contention

CC/DSM-Synch = Improved Flat Combining, without global lock or queue cleanup, developed in parallel with RCL [Fatourou PPoPP’12]
Using RCL in legacy applications

Three components:

• RCL runtime
  – Library that makes it possible to write RCL applications

• Profiler
  – To find out which applications / locks can potentially benefit from RCL

• Reengineering
  – To transform code for traditional locks into code that can use RCL
Using RCL in legacy applications

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
  – Not possible with combining locks
Using RCL in legacy applications

Profiler:

• Detects which applications / locks benefit from RCL

• Uses two metrics:
  – % of time spent in critical sections (measures contention)
  – Avg. # of cache misses in critical sections (measures data locality)
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:

```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);
    ...
}
```

```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:

• 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).
Outline

• Context: multicore architectures
• State of the art: locks
• Contribution: Remote Core Locking
• **Evaluation in legacy applications**
  – Methodology
  – Main results
  – Scalability
  – More software threads than hardware threads
• Perspectives and conclusion
Methodology

• Evaluation on two different machines
  – Different architectures and OSes

• Different application types
  – Parallel computing
    • Scientific computations (SPLASH-2), MapReduce (Phoenix 2)
  – Server applications (Memcached, Berkeley DB)

• Different configurations
  – One software thread per hardware thread
  – More software threads than hardware threads (Berkeley DB)
Magnycours-48

- Four Opteron 6172, two dies per CPU, six cores per die
  - No hardware multithreading: 48 hardware threads
- Non-complete interconnect graph
  - Asymmetrical access times
Niagara2-128

- Two UltraSPARC-T2+ CPUs, each with 8 cores
- Simultaneous hyperthreading: 8 hardware threads per core (!)
  - 128 hardware threads
- Less representative of current multicore machines
Differences

• Magnycours-48 has much faster sequential speed
• Niagara2-128 has faster communication speed / sequential speed
• On SPLASH-2, parallel scientific applications:
Differences

- Magnycours-48 has much faster sequential speed
- Niagara2-128 has faster communication speed / sequential speed
- On SPLASH-2, parallel scientific applications:
Differences

- Magnycours-48 has much faster sequential speed
- Niagara2-128 has faster communication speed / sequential speed
- On SPLASH-2, parallel scientific applications:
Outline

• Context: multicore architectures
• State of the art: locks
• Contribution: Remote Core Locking
• **Evaluation in legacy applications**
  – Methodology
  – **Main results**
  – Scalability
  – More software threads than hardware threads
• Perspectives and conclusion
Main results

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 > 15%, RCL is more efficient than POSIX locks
  – If time spent in CS > 60%, RCL is more efficient than all other locks
Collapse of POSIX (105,000 cycles): 15%

Collapse of MCS (60,000 cycles): 60%

% of time in CS

Delay (cycles)

Execution time (cycles)

- POSIX
- Spinlock
- MCS
- MCS-TP
- Flat Comb.
- CC-Synch
- DSM-Synch
- RCL
Main results

- Better performance when time in CS > 60%
  - Performance improvement correlated with time in CS

- Only one or two locks replaced each time

![Bar chart showing performance improvements](chart.png)

- **% in CS:**
  - Memcached: Set, Many DCMs: 44.7%
  - String Match: 63.9%
  - Raytrace: Balls4: 65.7%
  - Memcached: Get: 79.0%
  - Linear Regression: 81.6%
  - Radiosity: 87.7%
  - Raytrace: Car: 90.2%
  - Matrix Multiply: 92.2%
Main results

- Better performance when time in CS > 60%
  - Performance improvement correlated with time in CS

- Only one or two locks replaced each time

![Bar chart showing performance results for various workloads and synchronization methods. The x-axis represents different workloads and the y-axis represents the best performance fraction of the best POSIX perf. Higher is better. The chart includes workloads such as Memcached: Set and String Match, Raytrace: Balls4 and Memcached: Get, Linear Regression, Radiosity, Raytrace: Car, and Matrix Multiply. The synchronization methods include POSIX Spinlock, MCS Flat Combining, CC-Synch, and DSM-Synch. The graph highlights that RCL shows better performance when time in CS > 60%, with performance improvement correlated with time in CS. Only one or two locks are replaced each time. The % in CS for Memcached: Set is 44.7% (many DCMs), String Match 63.9%, Raytrace: Balls4 65.7%, Memcached: Get 79.0%, Linear Regression 81.6%, Radiosity 87.7%, Raytrace: Car 90.2%, and Matrix Multiply 92.2%.]
Main results

• Better performance when time in CS > 60%
  – Performance improvement correlated with time in CS

• Only one or two locks replaced each time

% in CS:
- Memcached: Set: 44.7%
- String Match: 63.9%
- Raytrace: Balls4: 65.7%
- Memcached: Get: 79.0%
- Linear Regression: 81.6%
- Radiosity: 87.7%
- Raytrace: Car: 90.2%
- Matrix Multiply: 92.2%

Higher is better

(Many DCMs)
Main results

- On Niagara2-128: profiler thresholds = 15% / 85%
Main results

• On Niagara2-128, no bench > 85%
  – Faster communication / sequential speed, less issues with contention

• Still some performance gains when time in CS > 15%

% in CS:  20.2%  38.7%  69.2%  79.1%
(many DCMs)
Main results

• On Niagara2-128, no bench > 85%
  – Faster communication / sequential speed, less issues with contention

• Still some performance gains when time in CS > 15%

% in CS:  20.2%  38.7%  69.2%  79.1%
(many DCMs)
Main results

- On Niagara2-128, no bench > 85%
  - Faster communication / sequential speed, less issues with contention

- Still some performance gains when time in CS > 15%


% in CS:
- Memcached: Set: 20.2% (many DCMs)
- Radiosity: 38.7%
- Memcached: Get: 69.2%
- Raytrace: Car: 79.1%
Outline

• Context: multicore architectures
• State of the art: locks
• Contribution: Remote Core Locking

• Evaluation in legacy applications
  – Methodology
  – Main results
  – Scalability
  – More software threads than hardware threads

• Perspectives and conclusion
Scalability of RCL

- RCL not only improves performance, it also improves scalability
- Example: Memcached with Set requests
  - On Magnycours-48 and Niagara2-128
- Memcached uses condition variables
  - No results for combining locks
Scalability of RCL

- Memcached, Set requests:
Scalability of RCL

- Memcached, Set requests:

![Graph showing scalability of RCL, Spinlock, MCS, POSIX under varying number of threads.](image)
Outline

• Context: multicore architectures
• State of the art: locks
• Contribution: Remote Core Locking
• Evaluation in legacy applications
  – Methodology
  – Main results
  – Scalability
  – More software threads than hardware threads
• Perspectives and conclusion
More sw threads than hw threads

• Many locks perform poorly when many software threads
  – Some spinning threads get woken up
  – Possible interference with scheduling: convoy effect (very slow)

• RCL dedicates a core: it always makes progress on the critical path
More SW threads than HW threads

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

![Graph showing performance comparison between different locking mechanisms and thread counts.]
More SW threads than HW threads

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

![Graph showing performance comparison between different thread models.](image)

- Quick collapse
More SW threads than HW threads

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

![Graph showing performance comparison between different thread configurations.](image-url)
More SW threads than HW threads

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

![Graph showing performance results for different client numbers and global request rates.](image)

- Original
- POSIX
- Spinlock
- MCS
- Flat Combining
- MCS-TP
- CC-Synch
- DSM-Synch
- RCL

Higher is better.
Yielding the processor

Was that a fair comparison?
Yielding the processor

Was that a fair comparison?

• What if locks yield the CPU instead of spinning?

• Less reactive, but threads no longer woken up just to spin?

• Added calls to yield() in MCS, MCS-TP, Combining Locks
  – …and RCL clients
Yielding the processor

- Berkeley DB / TPC-C, Stock Level requests, yield():

![Graph showing performance trends](image)
Yielding the processor

- Berkeley DB / TPC-C, Stock Level requests, yield():

![Graph showing performance comparison between different locking mechanisms]

- Yields the processor

- Berkeley DB / TPC-C, Stock Level requests, yield():

- Higher is better!
Outline

• Context: multicore architectures
• State of the art: locks
• Contribution: Remote Core Locking
• Evaluation in legacy applications
• Perspectives and conclusion
• Modified RCL implementations
  – Dynamic RCL runtime
  – Hierarchical RCL
  – RCL for embedded architectures

• HTMs: supported by Haswell
  – What can RCL do for transactional memories?
  – Hassan et al. [IPDPS ’14] propose a STM algorithm…
    • …that runs commit and invalidation on dedicated remote server threads
    • …with cache-aligned communication
    • …and uses RCL for locks
Non-cache-coherent architectures

- Could RCL provide performance improvements on non-cache-coherent architectures?
- Petrović et al. [PPoPP ’14] propose an algorithm inspired by RCL for partially cache-coherent architectures
- Major performance improvements on TILE-Gx CPUs.
Conclusion

• RCL reduces lock acquisition time and improves data locality
  – Cost: uses a few cores and may perform worse with few threads

•Profiler to detect when RCL can be useful

• Tool to ease the transformation of legacy code

• Future work:
  – Modified RCL implementations
  – Applying ideas from RCL to HTMs and NCC architectures
  – Started by others