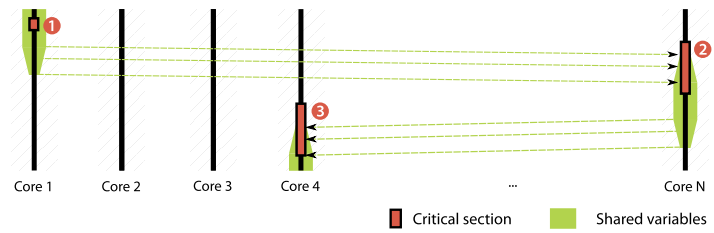
Remote Core Locking (RCL): Migrating Critical Section Execution To Improve Performance

Remote Core Locking (RCL) aims to improve the performance of multithreaded applications on multicore architectures. The general idea of RCL is to replace, in legacy applications, some performance-critical lock acquisitions by optimized remote procedure calls to a dedicated core. By doing so, RCL reduces both (i) the time needed to acquire a lock, by increasing data locality in critical sections and (ii) the time needed to acquire a lock under high contention, thanks to its client-server architecture.

Problem: on multicore architectures, executing critical sections can be slow

Issue #1: cache misses when executing critical sections

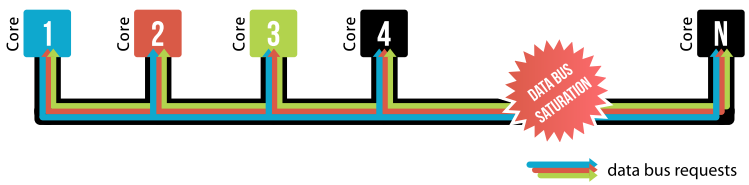
- A critical section mainly accesses shared data
- Shared data has to be fetched from the previous core's cache



Cache faults slow down the execution of critical sections ❷ and ❸.

Issue #2: data bus saturation

- Traditional spinlocks use atomic instructions to actively wait on a shared variable
- This technique induces frequent broadcasts on the data bus
- When contention is high, the data bus can become saturated

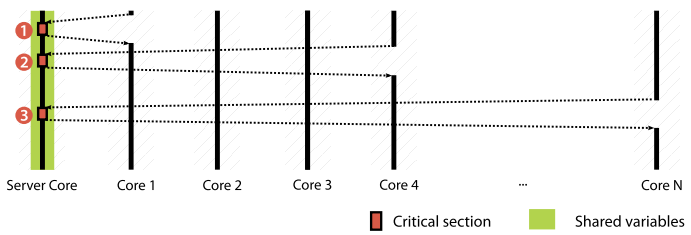


Core 1, 2 and 3 all try to acquire a spinlock, broadcasting write requests on the synchronization cache lines in a busy-wait loop. This leads to data bus saturation.

Our solution: migrating the execution of critical sections to a remote core

Solution to issue #1: all shared data remains cached

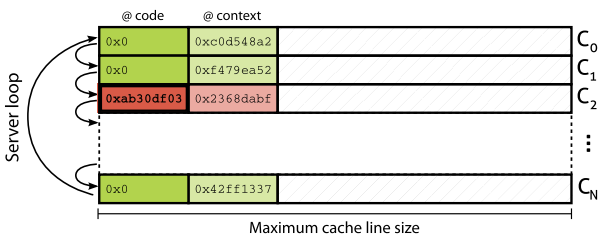
- Shared variables remain in the cache of the server core: critical sections execute faster
- RCL can be seen as transferring control instead of data



Critical sections ❷ and ❸ execute faster because shared variables are cached.

Solution to issue #2: one communication channel per client

- One cache line per client is used to communicate with the server: no saturation

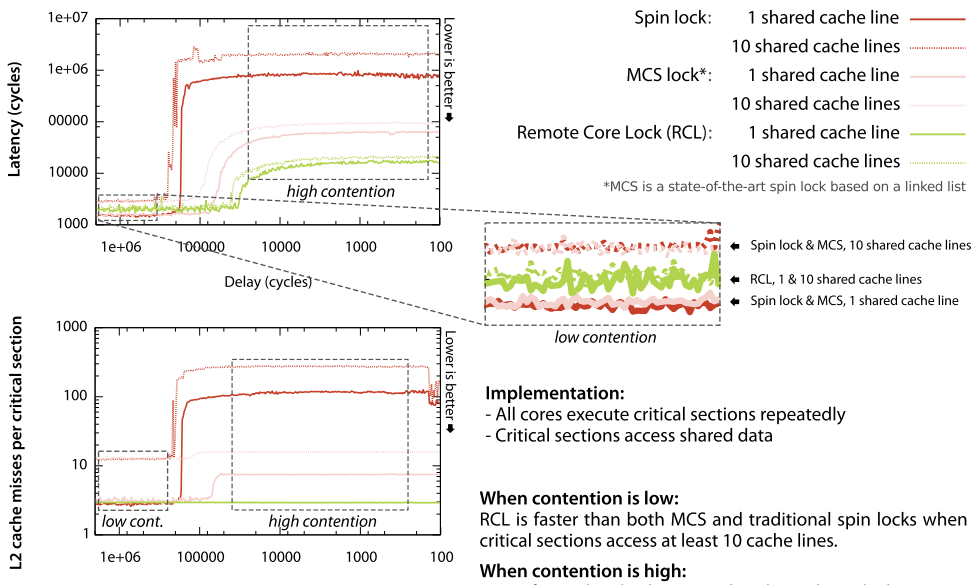


Client #2 (C_2) requested the execution of a critical section.

Algorithm:
- To request the execution of a critical section, a client writes in his block the addresses of (i) the code of the critical section and (ii) a struct containing context variables
- The server actively waits for a request and executes them

Evaluation

Microbenchmark

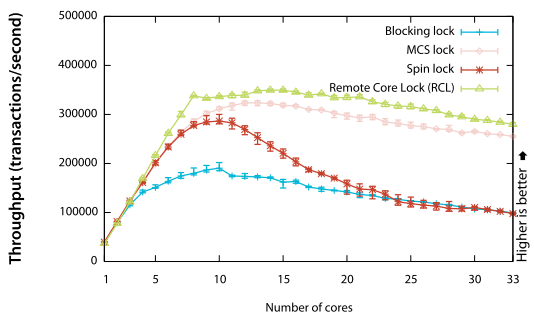


Implementation:
- All cores execute critical sections repeatedly
- Critical sections access shared data

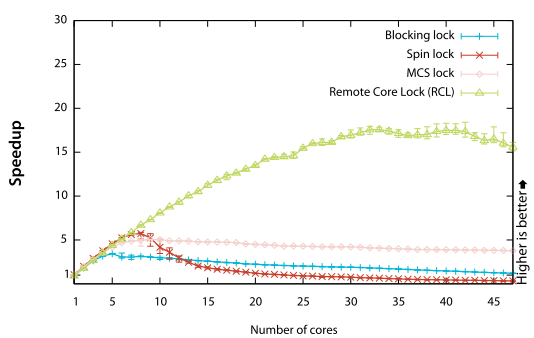
When contention is low:
RCL is faster than both MCS and traditional spin locks when critical sections access at least 10 cache lines.

When contention is high:
RCL is faster than both MCS and traditional spin locks.

Macrobenchmarks, *preliminary results*



Memcached:
Frequent cache misses in critical sections, performance gain.



SPLASH-2/Raytrace:
High contention, scalability gain.

Jean-Pierre Lozi* Gaël Thomas* Julia Lawall† Gilles Muller*

* *LIP6, University Pierre and Marie Curie, Paris, France*
† *DIKU, University of Copenhagen, Denmark*

The Remote Core Lock (RCL): Can Migrating The Execution Of Critical Sections To Remote Cores Improve Performance?

Jean-Pierre Lozi (Student) Gaël Thomas Gilles Muller
LIP6
University of Paris 6
{jean-pierre.lozi, gael.thomas, gilles.muller}@lip6.fr

Julia Lawall
DIKU
University of Copenhagen
julia@diku.dk

No demo will be presented with the poster.

Multicore hardware is now commonplace in most computer systems, ranging from powerful servers to desktop environments and even embedded systems. NUMA technology makes it possible to use a large number of processing cores at the cost of a complex hierarchy of caches and buses. Nevertheless, current systems and applications are unable to take advantage of these new architectures: performance stalls, even though the number of cores keeps increasing.

The main obstacle hampering the scalability of multithreaded applications over a large number of cores is the cost of executing blocks of code that manipulate shared data in mutual exclusion. These blocks of code are protected by locks and called *critical sections*. When the number of cores is high, critical sections often cause major bottlenecks, for two main reasons. First, the cost of acquiring a lock can be prohibitive when a large number of threads try to enter a critical section simultaneously, because of the resulting load on the memory bus. Second, executing the code of critical sections can be costly because critical sections often access shared variables that have most recently been accessed by another core: this data must therefore be transfered from the other core's cache to the local core's cache. The resulting overhead is often the cause of bottlenecks on modern architectures in which communication between a large number of cores is handled by costly cache coherency algorithms.

We propose a new locking technique, Remote Core Locking (RCL), which aims to improve the performance of multithreaded applications on multicore architectures. The general idea of RCL is to replace, in legacy applications, some performance-critical lock acquisitions by optimized remote procedure calls to a dedicated "server" core. RCL has two advantages. First, by replacing a lock acquisition by a fast transfer of control, implemented with a single exchange of a cache line between the client and the server, RCL does not suffer from the performance collapse observed with regular locks when a large number of threads try to acquire the same lock simultaneously. Second, since all critical sections are executed on the server core, shared variables accessed by critical sections remain in the server core's cache: no costly transfer of shared data between cores is needed.

Related work Several approaches have tried to reduce the cost of lock acquisitions under high contention. First, various types of locks that remain efficient even when contention is high have been proposed [Bacon 1998, Johnson 2010, Scott 2001]. However, these solutions do not take data locality into account: shared data still needs to be transfered between cores when critical sections are executed. Other authors have proposed to reduce the use of locks by identifying common locking patterns (readers-writer pattern, read concurrent update [Guniguntala 2008]). These techniques limit the use of locks in certain cases without completely avoiding them: they are therefore complementary to our approach. Finally, algorithms avoiding locks completely have been proposed [Herlihy 2008, Michael 1996], but their use is limited to very restricted cases: therefore, improving locking mechanisms on multicore architectures remains fundamental.

Other work has focused on limiting the cost of transfering shared data between cores. Several experimental operating systems optimized for multicore architectures replicate shared data structures on each core to avoid data transfers [Baumann 2009a, Wickizer 2008]. Albeit efficient, these solutions require a complete overhaul of operating systems and applications. It has also been proposed to avoid relying on cache coherency algorithms to transfer data between cores by using message-passing techniques exclusively [Baumann 2009b]. However, this solution also requires major code rewrites and is more suited to the still uncommon cache-incoherent architectures.

Contributions In the context of this preliminary work, we propose three following contributions: (1) an efficient implementation of RCL, (2) a methodology to replace classical lock acquisitions with RCL, and (3) a first set of evaluations of RCL.

An efficient implementation of RCL. Each client communicates with the server using a single "synchronization" cache line. To execute a critical section, a client writes into the two first words of this cache line: (i) the address of the code of the critical section and (ii) the address of a data structure containing the local variables that are read and written by the critical section (a.k.a., the "context"). It then actively waits for the server to set the address of the code to `NULL`, which means that the critical section has been executed. The server checks each synchronization cache line in a loop, until it finds a non-`NULL` code address. When such an address is

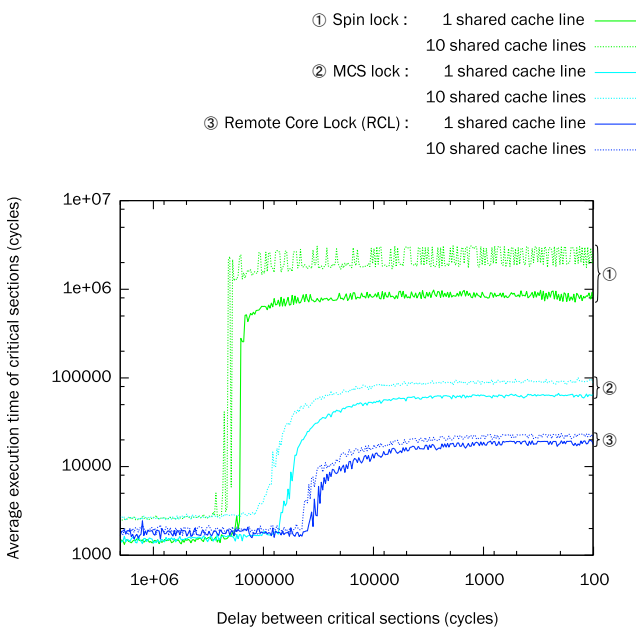


Figure 1. Latency of RCL relative to classical spinlocks and to the state-of-the-art MCS spinlock. A low delay between critical sections (x-axis) means that contention is high.

found, the server uses it as well as the address of the context to execute the critical section. It then resets the address of the code to `NULL`. This implementation is efficient for two reasons : first, its use of active waiting allows for high reactivity, and second, using one synchronization variable per client instead of a single global synchronization variable avoids overloading the data bus when a lot of threads try to acquire a lock concurrently.

A methodology to replace classical lock acquisitions with RCL. Since the code and context of critical sections have to be isolated in order to use RCL, using a single pair of lock/unlock functions is not sufficient. However, a classical lock can be replaced with RCL using an algorithm that can either be used manually by a developer or implemented in a compiler to replace any lock with RCL automatically.

A first set of evaluations of RCL. First, measurements with a custom microbenchmark show that RCL is more efficient than both classical spinlocks and than the state-of-the-art MCS spinlock when contention is high, as shown on Figure 1. If five cache lines or more are accessed in critical sections, RCL is more efficient than spinlocks even when contention is low. Second, using RCL in the Raytrace application from the SPLASH-2 benchmark suite increases performance up to 70% and scalability up to 400% relative to `pthread` locks. Finally, using RCL in the memcached cache server increases performance up to 70%.

RCL is a novel way of managing mutual exclusion in multithreaded programs : not only does it reduce the time needed to acquire a lock when contention is high, it also executes the code of critical sections faster thanks to improved data locality. Experiments with a microbenchmark show that when enough shared data is accessed in critical sections, RCL is always more efficient than state-of-the-art spinlocks. Preliminary experiments with macrobenchmarks show the usefulness of RCL in real-world applications. Future work will consist in writing tools allowing to locate locks that can be efficiently replaced by RCL, in order to provide a comprehensive solution to improve the performance of legacy applications on multicore architectures.

References

[Bacon 1998] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: featherweight synchronization for Java. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 258–268, New York, NY, USA, 1998. ACM Press.

[Baumann 2009a] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schuepbach, and A. Singhania. The Multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 29–44. ACM Press, 2009.

[Baumann 2009b] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schuepbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the 22nd ACM symposium on Operating systems principles*, SOSP '09, pages 29–44, Big Sky, MO, USA, October 2009. ACM Press.

[Guniguntala 2008] D. Guniguntala, P. E. McKenney, J. Triplett, and J. Walpole. The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with Linux. *IBM Systems Journal*, 47(2):221–236, April 2008.

[Herlihy 2008] M. Herlihy and N. Shavit. *The Art Of Multiprocessor Programming*. Elsevier, 2008.

[Johnson 2010] F. R. Johnson, R. Stoica, A. Ailamaki, and T. C. Mowry. Decoupling contention management from scheduling. In *Proceedings of the 15th edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '10, pages 117–128, New York, NY, USA, 2010. ACM Press.

[Michael 1996] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, PODC '96, pages 267–275, New York, NY, USA, 1996. ACM Press.

[Scott 2001] M. L. Scott and W. N. Scherer. Scalable queue-based spin locks with timeout. In *Scalable queue-based spin locks with timeout*, PPoPP '01, pages 44–52, New York, NY, USA, 2001. ACM Press.

[Wickizer 2008] S. B. Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: an operating system for many cores. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2008.