Fast and Portable Locking for Multicore Architectures

JEAN-PIERRE LOZI, Université Nice Sophia Antipolis, CNRS
FLORIAN DAVID, Sorbonne Universités, Inria, CNRS, UPMC
GAËL THOMAS, SAMOVAR, CNRS, Télécom ParisSud, Université Paris-Saclay
JULIA LAWALL and GILLES MULLER, Sorbonne Universités, Inria, CNRS, UPMC

The scalability of multithreaded applications on current multicore systems is hampered by the performance of lock algorithms, due to the costs of access contention and cache misses. The main contribution presented in this article is a new locking technique, Remote Core Locking (RCL), that aims to accelerate the execution of critical sections in legacy applications on multicore architectures. The idea of RCL is to replace lock acquisitions by optimized remote procedure calls to a dedicated *server* hardware thread. RCL limits the performance collapse observed with other lock algorithms when many threads try to acquire a lock concurrently and removes the need to transfer lock-protected shared data to the hardware thread acquiring the lock, because such data can typically remain in the server's cache. Other contributions presented in this article include a profiler that identifies the locks that are the bottlenecks in multithreaded applications and that can thus benefit from RCL, and a reengineering tool that transforms POSIX lock acquisitions into RCL locks.

Eighteen applications were used to evaluate RCL: the nine applications of the SPLASH-2 benchmark suite, the seven applications of the Phoenix 2 benchmark suite, Memcached, and Berkeley DB with a TPC-C client. Eight of these applications are unable to scale because of locks and benefit from RCL on an ×86 machine with four AMD Opteron processors and 48 hardware threads. By using RCL instead of Linux POSIX locks, performance is improved by up to 2.5 times on Memcached, and up to 11.6 times on Berkeley DB with the TPC-C client. On a SPARC machine with two Sun Ultrasparc T2+ processors and 128 hardware threads, three applications benefit from RCL. In particular, performance is improved by up to 1.3 times with respect to Solaris POSIX locks on Memcached, and up to 7.9 times on Berkeley DB with the TPC-C client.

Categories and Subject Descriptors: D.4.1 [**Operating Systems**]: Process Management—*Mutual exclusion*

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Multicore, synchronization, locks, RPC, locality, busy-waiting, memory contention, profiling, reengineering

ACM Reference Format:
Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia Lawall, and Gilles Muller. 2016. Fast and portable locking for multicore architectures. ACM Trans. Comput. Syst. 33, 4, Article 13 (January 2016), 62 pages.
DOI: http://dx.doi.org/10.1145/2845079

Authors' addresses: J.-P. Lozi, I3S, Templiers 1 building, office 446, 930 route des Colles, BP 145, 06903 Sophia Antipolis Cedex, France; email: jean-pierre.lozi@unice.fr; F. David, UPMC - LIP6, Boîte courrier 169, Couloir 25-26, Étage 3, Bureau 331, 4 place Jussieu, 75252 Paris Cedex 05; email: florian.david@lip6.fr; G. Thomas, Département d'informatique, Bureau B306, Telecom SudParis, 9, rue Charles Fourier, F-91011 Evry Cedex, France; email: gael.thomas@telecom-sudparis.eu; J. Lawall, UPMC - LIP6, Boîte courrier 169, Couloir 25-26, Étage 3, Bureau 318, 4 place Jussieu, 75252 Paris Cedex 05; email: julia.lawall@lip6.fr; G. Muller, UPMC - LIP6, Boîte courrier 318, Couloir 25-26, Étage 3, Bureau 318, 4 place Jussieu, 75252 Paris Cedex 05; email: gilles.muller@lip6.fr.

DOI: http://dx.doi.org/10.1145/2845079

1. INTRODUCTION

Over the past decades up to the early 2000s, the performance of Central Processing Units (CPUs) was steadily improving thanks to rising hardware clock frequencies and improved superscalar, pipelined designs. However, due to physical limitations, CPU manufacturers have now found it impossible to keep increasing clock frequencies, and the performance improvements obtained by pipelining and superscalar designs has peaked. Consequently, manufacturers now mainly focus on embedding many execution units into the same CPU instead to improve performance: the number of hardware threads in consumer CPUs keeps increasing in all computing devices, from servers to mobile phones. It is not uncommon for multiprocessor servers to include more than 100 hardware threads nowadays.

In order to take advantage of these available hardware threads, programs have to be parallelized, that is, they have to define multiple threads to perform the processing concurrently. However, most programs cannot be fully parallelized because accesses to shared data structures have to be synchronized in order to ensure that they remain consistent [Herlihy and Shavit 2008]. One of the most commonly used techniques to synchronize accesses to shared data structures consists of defining *critical sections*, that is, sections of code that are executed in mutual exclusion. Current mechanisms used to implement critical sections [Mellor-Crummey and Scott 1991a; He et al. 2005a; Hendler et al. 2010a; Fatourou and Kallimanis 2012], however, use many costly operations that access remote data on the critical path of the execution. These operations include atomic instructions or read and write operations on shared variables, and they involve costly communication schemes that are handled by the cache-coherency protocol, such as cache line invalidations or cache misses. As stated by Amdahl's Law [Amdahl 1967], the critical path ends up being the limiting factor for performance as the number of threads increases, because the time spent in the parallel portion of the code decreases until it becomes negligible as compared to the critical path. As a result, costly communication schemes end up consuming a large part of the execution time of the application as the number of hardware threads increases. We have observed that the costly operations that access remote data on the critical path by current mechanisms used to implement critical sections can be classified in two categories:

- *Lock management*. Mutual exclusion is implemented with *locks*, that is, objects that can only be held by a single thread at any given time: a thread needs to hold a lock in order to execute a critical section. Locks are implemented with shared data structures, such as queues or Boolean variables. Acquiring or releasing a lock typically requires intercore communication to write to these shared data structures in order to change the state of the lock and to maintain their consistency.
- *Accessing shared variables*. A critical section is typically used to ensure mutual exclusion on a resource, which is often represented by a shared data structure. When different threads that are scheduled on different hardware threads execute the critical section, the cache lines that hold the shared data structure bounce between the hardware threads, causing cache misses.

In this article, we propose a new locking mechanism, Remote Core Locking (RCL), that aims to improve the performance of legacy multithreaded applications on multicore hardware by decreasing the amount of intercore communication on the critical path triggered by the execution of critical sections.[1] The principle of RCL is to reserve a hardware thread, called the *server*, to execute all the critical sections associated with

[1] RCL was initially presented in the paper "Remote Core Locking: Migrating Critical-Section Execution to Improve the Performance of Multithreaded Applications," which was published at USENIX ATC 2012 [Lozi et al. 2012]. Beyond the contents of the USENIX ATC paper, this article includes extended evaluation results, including results on two different architectures instead of one, more evaluated lock algorithms (including

a lock. When an application thread, called a *client*, has to execute a critical section, it posts a message to the server, which subsequently executes the critical section and signals the client when the critical section has been executed. RCL naturally ensures mutual exclusion, since the execution of critical sections is serialized on the server. RCL reduces the cost of lock management because its message-based communication between the client and server uses less intercore communication on the critical path than existing lock implementations. RCL also optimizes data locality because shared data structures protected by the lock remain in the caches of the hardware thread of the server, which reduces the amount of cache misses on the critical path.

Replacing locks of legacy C applications by RCL, however, raises two issues. First, RCL artificially serializes the critical sections associated with different locks managed by the same server hardware thread: transforming many locks into RCLs on a smaller number of servers can induce *false serialization*, which decreases parallelism. Therefore, the programmer must first decide which locks should be transformed into RCLs and on which hardware thread(s) to run the server(s). For this, we have designed a profiler to identify which locks are frequently used by the application and how much time is spent on locking. Based on this information, we propose a set of simple heuristics to help the programmer decide which locks should be transformed into RCLs, and a methodology to identify on which hardware thread to run each such lock's server. Second, in legacy C applications, the code of the critical sections, lock acquisitions included, is often embedded inside a longer function. As the control flow has to migrate to a server and then return to the client during the execution of the critical section, the code of each critical section that is associated with a lock transformed into a RCL has to be extracted as a function. To facilitate this transformation, we have designed an automated reengineering tool for C programs to transform the code of each critical section so that it can be executed as a remote procedure call on the server hardware thread: the code within the critical section is extracted as a function and any variables referenced or updated by the critical section that are declared by the function containing the critical section code are sent as arguments, amounting to a *context*, to the server hardware thread.

We have developed a runtime for both Linux and Solaris that is compatible with POSIX threads, and that supports a mixture of RCL and POSIX locks in a single application. Based on this runtime, we have evaluated RCL on two machines: (i) Magnycours-48, an ×86 machine with four AMD Opteron CPUs and 48 hardware threads running Linux 3.9.7, and (ii) Niagara2-128, a SPARC machine with two Ultrasparc T2 CPUs and 128 hardware threads running Solaris 10. We compare the performance of RCL to that of other locks using a custom microbenchmark that measures the execution time of critical sections that access a varying number of shared memory locations. Furthermore, based on the results of our profiler, three applications from the SPLASH-2 suite [University of Delaware 2007; Singh et al. 1992], three applications in the Phoenix 2 suite [Stanford University 2011; Talbot et al. 2011; Yoo et al. 2009; Ranger et al. 2007], Memcached [Danga Interactive 2003; Fitzpatrick 2004], and Berkeley DB [Oracle Corporation 2004; Olson et al. 1999] with a TPC-C benchmark developed at Simon Fraser University were identified as applications that could benefit from RCL. For each of these applications, we compare RCL against the standard system implementation of the POSIX lock, MCS [Mellor-Crummey and Scott 1991a], Flat Combining [Hendler et al. 2010a], CC-Synch [Fatourou and Kallimanis 2012], and DSM-Synch [Fatourou and Kallimanis 2012]. Comparisons are made for a same number of hardware threads, which means that there are fewer application threads in the RCL case, since one or more hardware threads are dedicated to RCL servers.

the recently proposed CC-Synch and DSM-Synch lock algorithms [Fatourou and Kallimanis 2012]), and additional experiments.

The key highlights of our results are as follows:

—On our custom microbenchmark, under high contention, RCL is faster than all other evaluated lock algorithms: on Magnycours-48 (Niagara2-128, respectively), RCL is 3.2 (1.8, respectively) times faster than the second best approach, CC-Synch, and 5.0 (7.2, respectively) times faster than the operating system's POSIX lock.
—On application benchmarks, contexts are small, and thus the need to pass a context to the server has only a marginal performance impact.
—On most benchmarks, only one lock is frequently used, and therefore only one RCL server is needed. The only exception is Berkeley DB with the TPC-C client, which requires two or three RCL servers to reach optimal performance by reducing false serialization.
—On Magnycours-48 (Niagara2-128, respectively), RCL performs better on five (one, respectively) application(s) from the SPLASH-2 and Phoenix 2 benchmark than all other evaluated locks.
—For Memcached with Set requests, on Magnycours-48 (Niagara2-128, respectively), RCL yields a speedup of 2.5 (1.3, respectively) times over the operating system's POSIX lock, 1.9 (1.2, respectively) times over the basic spinlock, and 2.0 (1.2, respectively) times over MCS. The number of cache misses in critical sections is divided by 2.9 (2.3, respectively) by RCL, which shows that it can greatly improve locality. Flat Combining, CC-Synch, and DSM-Synch were not evaluated in this experiment because they do not implement condition variables, which are used by Memcached.
—For Berkeley DB with the TPC-C client, when using Stock Level transactions, on Magnycours-48 (Niagara2-128, respectively), RCL yields a speedup of up to 11.6 (7.6, respectively) times over the original Berkeley DB locks for 48 (384, respectively) simultaneous clients. RCL resists better than other locks when the number of simultaneous clients increases. In particular, RCL performs much better than other locks when the application uses more client threads than there are available hardware threads on the machine, even when other locks are modified to yield the processor in their busy-wait loops.

The article is structured as follows. Section 2 describes existing lock algorithms. Section 3 presents our main contribution, RCL. Section 4 presents the profiler and the reengineering tool that are provided with RCL to facilitate its use in legacy applications. Section 5 evaluates the performance of RCL and compares its performance to that of existing lock algorithms. Finally, Section 6 concludes.

2. RELATED WORK

The lock, one of the oldest synchronization mechanisms [Dijkstra 1965; Hoare 1974], is still extensively used in modern applications. A lock makes it possible for multiple threads to execute sections of code in mutual exclusion in order to ensure exclusive access to shared resources. In this section, we study several lock algorithms, focusing on the following five criteria:

—*Cost of lock management.* The cost of acquiring and releasing the lock on the critical path, that is, lock management operations that are serialized with the critical sections protected by the lock.
—*Data locality.* The cost, in terms of number of cache misses, of accessing the shared data protected by the lock.
—*Performance with many threads.* How well the lock algorithm performs when more threads use the lock than there are hardware threads on the machine. The performance of some lock algorithms collapses in this scenario.
—*Support for condition variables.* Not all lock algorithms provide support for condition variables, and supporting them is not always trivial (notably with combining locks, as

will be explained in Section 2.3). Supporting condition variables is crucial for legacy support, since they are very commonly used in existing applications. For instance, roughly half of the multithreaded applications that use POSIX locks in Debian 6.0.3 also use them.

—*Fairness*. The ability of a lock algorithm to ensure that all threads make progress.

In the rest of this section, we provide an in-depth analysis of the families of locks used in most of our evaluation in Section 5, with examples. We then present other notable lock algorithms that will be evaluated in Section 5.6.

2.1. Centralized Locks

With centralized locks, the ownership of a lock is determined by a single Boolean variable that we refer to as the *lock variable*. The thread that is able to change the lock variable from **`false`** to **`true`** with a compare-and-swap instruction becomes the owner of the lock. The compare-and-swap instruction ensures that a variable is atomically set to a new value if and only if it was equal to a given value. The lock is released by setting the lock variable to **`false`** using a regular assignment instruction. We now describe and analyze the properties of two widely used variants of centralized locks: spinlocks and blocking locks.

2.1.1. Spinlocks. With spinlocks [Herlihy and Shavit 2008], a thread continuously tries to set the lock variable to **`true`** in a busy-wait loop, until it manages to acquire the lock. Some variants use various policies to yield the processor in the busy-wait loop in order to improve resistance to contention and save energy, as will be seen in Section 2.4. In this article, we refer to the spinlock algorithm that never yields the processor as the *basic spinlock*.

2.1.2. Blocking Locks. With blocking locks, when a thread fails to acquire a lock, it sleeps via a system call, such as `futex_wait()` on Linux. When the lock holder releases the lock, it not only sets the lock variable to **`false`**, but also wakes up the threads waiting for the lock. On mainstream POSIX operating systems, such as Linux or Solaris, the POSIX lock is implemented with a blocking lock.

2.1.3. Analysis. We now analyze the properties of centralized locks.

Cost of Lock Management. With basic spinlocks, when many threads try to acquire the lock concurrently by repeatedly applying compare-and-swap instructions to the lock variable, the processor interconnect gets saturated by messages from the cache-coherence protocol. As will be shown in Section 5.1, the result is that even executing a single one of these compare-and-swap instructions takes a lot of time. Consequently, the critical path, which includes setting the lock variable to **`true`** at the beginning of a critical section and setting the lock variable to **`false`** at the end, becomes very long.

Blocking locks are also relatively inefficient under high contention, because threads sleep while they are waiting for the lock. Consequently, two lengthy lock management operations take place on the critical path before a thread can reacquire the lock: (i) a system call to unblock the thread, and (ii) a context switch to schedule that thread.

Data Locality. As stated in the Introduction, when a thread executes a critical section that was last executed on another hardware thread, the shared data protected by the critical section has to be loaded when it is accessed, which results in cache misses.

Performance with Many Threads. If the lock holder is preempted, none of the threads that are waiting for the lock can acquire it, consequently, no progress is made on the critical path. This is a major issue with the basic spinlock when there are more threads that use the lock than there are hardware threads on the machine: if all threads are busy (either busy-waiting or executing a critical section), the operating system's scheduler will have to preempt some of these threads without knowing which thread is executing a critical section and therefore progressing on the critical path.

Blocking locks, however, can perform much better than spinlocks when there are more threads that use the lock than there are hardware threads on the machine. This is due to the fact that threads sleep while they are waiting for the lock, which reduces the need for the scheduler to preempt threads in applications where many threads are waiting on locks: the scheduler never preempts the lock holder in order to run a busy-waiting thread on the same hardware thread.

Support for Condition Variables. Blocking locks are provided by all mainstream operating systems through standard lock implementations, such as the POSIX lock implementations on Linux and Solaris. These implementations also provide primitives to handle condition variables. Support for condition variables for spinlocks can trivially be implemented using the aforementioned primitives.

Fairness. None of the centralized locks presented in this section guarantee progress. A thread that tries to enter a critical section can indefinitely be delayed by other threads that succeed in acquiring the lock.

2.2. Queue Locks

Queue locks define the lock variable as a pointer to the tail of a queue of nodes, with one node for each thread that is either waiting for the lock or executing a critical section. Each node contains a synchronization variable named `wait` and a pointer to another node. A thread that waits for the lock busy-waits on one of the `wait` variables, and enters the critical section when the `wait` variable is set to **`false`**. Two variants of queue locks exist: CLH [Craig 2003; Magnusson et al. 1994] and MCS [Mellor-Crummey and Scott 1991b]. In both algorithms, each thread initially owns a preallocated node.[2]

2.2.1. CLH. In CLH, each node contains a pointer to its predecessor. The queue is never empty: it initially contains a dummy node whose `wait` variable is set to **`false`**. In order to acquire the lock, a thread sets the `wait` variable of its node to **`true`** and atomically enqueues its node at the end of the list. It then waits for the lock variable of its predecessor to be equal to **`false`** to enter the critical section. A lock owner sets the `wait` variable of its node to **`false`** to pass lock ownership to its successor in the list. It then takes ownership of its predecessor's node as its new node.

2.2.2. MCS. In MCS, a node contains a pointer to its successor. The queue does not contain a dummy node. When the queue is empty, the lock is free, and a thread acquires the lock by enqueueing its node at the head of the list. Otherwise, the lock is not free. In that case, the thread sets its node's `wait` variable to **`false`** and enqueues it at the end of the list. It then busy-waits on the `wait` variable of its newly enqueued node, instead of the previous node as is the case with CLH. To release the lock, the lock holder sets the `wait` variable of its successor, if it exists, to **`true`**. If no successor exists, the lock holder sets the tail pointer to **`null`**. As compared to CLH, MCS may have better memory locality on NUMA and non-cache-coherent architectures because a thread always busy-waits on the same node, and that node is allocated locally.

2.2.3. Analysis. We now analyze the properties of queue locks.

Lock Management. If a waiting thread does not get preempted, the node on which it busy-waits remains in the local cache until its predecessor releases the lock. No other thread busy-waits on the same variable. Consequently, queue locks are not subject to

[2] In practice, in order to allow for nested critical sections, each thread needs to own one node per lock. To simplify the presentation, we do not consider this case in this section.

the processor interconnect saturation issue that centralized locks suffer from. At high contention, on the critical path, between each critical section, the only overhead is the cost of the previous lock holder writing to the corresponding `wait` variable (invalidation), and the new lock holder fetching the new value of that variable (cache miss).

Data Locality. Like with centralized locks, shared variables protected by the lock have to be loaded in the cache of the lock holder.

Performance with Many Threads. Like with centralized locks, when there are more threads that use the lock than there are hardware threads on the machine, the scheduler is likely to preempt the lock holder. Consequently, threads waiting for the lock waste resources while no progress is made on the critical path because no critical section can be executed.

Another reason why queue locks perform poorly when more threads try to acquire a lock than there are hardware threads on the machine is the *convoy effect* [Koufaty et al. 2010]: the FIFO (First-In-First-Out) ordering of lock acquisitions and the FIFO scheduling policy of the operating system interact in such a way that critical sections take one or several of the scheduler's time quanta to be executed. To illustrate the problem, suppose we have a machine with two hardware threads and three threads, T_1, T_2, and T_3. We also suppose that both the queue of the lock and the queue of the system scheduler are $T_1 \leftarrow T_2 \leftarrow T_3$. During the first quantum, T_1 and T_2 are scheduled. T_1 acquires the lock since it is the first element of the lock queue. At the end of its critical section, it releases the lock and likewise enqueues itself at the end of the lock queue to execute a new critical section. T_2 then acquires the lock, executes the critical section, and enqueues itself at the end of the lock queue. At this point, T_1 and T_2 are still scheduled, but are unable to progress because T_1 waits for T_3 to release the lock and T_2 waits for T_1 to release the lock. At the end of the quantum, both the lock queue and the scheduling queue are $T_3 \leftarrow T_1 \leftarrow T_2$. In the same manner, during the next quantum, only T_3 and T_1 are able to execute a critical section: after the execution of their critical sections, they have to wait for T_2 to release the lock. As a result, during each quantum, only two critical sections are executed, leading to extremely low performance in the very likely case where executing two critical sections is orders of magnitude faster than a time quantum.

Time-published locks [He et al. 2005a] (MCS-TP) are modified versions of the MCS and CLH locks that aim to prevent the issues caused by preemption. They use a timestamp-based heuristic to solve the problems caused by interactions with the system scheduler. Each thread periodically writes the current system time (a high-resolution timestamp) in its node. If a thread fails to acquire the lock for a long amount of time, it checks the timestamp of the lock holder, and if that timestamp has not been recently updated, it assumes that the lock holder has been preempted. In this case, it yields the processor in order to increase the probability for the lock holder to be scheduled again: this technique helps solve the problem of a preempted lock owner. Moreover, if a thread waiting for the lock observes that the timestamp counter of its successor has not been recently updated, the thread assumes that its successor has been preempted; consequently, it removes it from the queue. This technique removes convoys, because it deschedules threads that needlessly busy-wait for the lock, leaving a chance for the lock holder to be scheduled again. However, time-published locks have a significant overhead relative to standard queue locks when fewer threads try to acquire a lock than there are hardware threads, due to the additional timestamp management operations on the critical path.

Support for Condition Variables. Like with spinlocks, support for condition variables for queue locks can trivially be implemented using the primitives that the operating system provides to manage condition variables in blocking locks.

Fairness. Queue locks are naturally fair: when a thread fails to acquire a lock, it enqueues itself in a FIFO queue, which also ensures that critical sections are executed in FIFO order. Time-publishing locks may be slightly less fair because a thread can be removed from the lock queue in order to avoid convoys.

2.3. Combining Locks

A combining lock [Oyama et al. 1999; Hendler et al. 2010a; Fatourou and Kallimanis 2012] is a lock algorithm that sometimes also aims to merge operations on a shared object in a way that decreases algorithmic complexity. Conceptually, each thread owns a node that contains a pointer to a function that encapsulates a critical section that can be executed remotely by another thread. A thread adds its node in a linked list (stack, queue, or unordered list) to request the execution of a critical section. One of the threads is designed as the *combiner thread* and will execute some of these operations, if possible merging them using an algorithm that is able to execute several critical sections together faster than if they were executed individually (such an algorithm does not always exist). A combining lock can also be used as a traditional lock algorithm if operations are not merged, but simply executed sequentially by the combiner: this is the only use of combining locks we consider in this article.

We describe three variants of combining locks: the Oyama lock [Oyama et al. 1999] (the oldest of the combining locks; its authors did not propose to merge operations), Flat Combining [Hendler et al. 2010a], and CC-Synch along with its variant for non-cache-coherent architectures, DSM-Synch [Fatourou and Kallimanis 2012].

2.3.1. The Oyama Lock. The Oyama lock [Oyama et al. 1999] (simply referred to as "Oyama" in the rest of this article) uses a synchronization variable that can take three values: (i) `FREE`, which means that the lock is free, (ii) `LOCKED`, which means that a critical section is being executed, but no other critical section needs to be executed after that, or (iii) a pointer to a stack, that is, a linked list that ensures LIFO (Last-In-First-Out) order. When a thread t needs to execute a critical section, it tries to atomically switch the value of the synchronization variable from `FREE` to `LOCKED`. In case of failure, another thread is executing a critical section. Consequently, t writes the address of the function that encapsulates it into its node, and pushes its node atomically onto the stack. Otherwise, if t succeeds in switching the value of the synchronization variable from `FREE` to `LOCKED`, it owns the lock and executes its critical section. It then tries to release the lock using a compare-and-swap instruction to switch its value from `LOCKED` to `FREE`. This operation can fail if other threads have pushed their nodes on the stack during the execution of the critical section. In this case, t becomes a combiner: it atomically detaches the stack, sets the value of the synchronization variable to `LOCKED`, and executes all critical sections from the stack. Thread t then tries to release the lock atomically again, and in case of failure, continues to act as a combiner.

2.3.2. Flat Combining. Flat Combining [Hendler et al. 2010a] uses a basic spinlock and an unordered linked list. Nodes can either be active or inactive. To execute a critical section, a node writes the address of the function that encapsulates it into its node and checks whether the node is active. If the node is inactive, it activates it and inserts it into the linked list, otherwise, no insertion is needed because the node was placed in the linked list for the execution of a previous critical section and has not yet been removed.

The thread then busy-waits until either (i) a combiner has executed its request, or (ii) the spinlock is free. In the latter case, the thread acquires the spinlock, becomes a combiner, and executes all critical sections in the linked list before releasing the spinlock. After having executed a critical section, the combiner sets the critical section's address to **`null`** in the node, which (i) indicates to the thread that owns the node that its critical section has been executed, and (ii) indicates to the next combiner that the node does not contain a pending request to execute a critical section.

Flat Combining performs linked list cleanup operations, which incurs a significant overhead: nodes that have not been used to execute critical sections in a long time are removed from the linked list regularly in order to ensure that the combiner will not traverse too many inactive nodes. Constants have to be tuned by the developer to determine (i) how long a node is allowed to stay in the list without requesting the execution of a critical section and (ii) how often the cleanup operation takes place.

2.3.3. CC-Synch and DSM-Synch. CC-Synch and DSM-Synch [Fatourou and Kallimanis 2012] use a queue instead of an unordered list. Nodes contain the address of the function that encapsulates the critical section, a `completed` Boolean variable that indicates whether the critical section has been executed, and a `wait` Boolean variable that indicates whether a thread should wait for the execution of its critical section.

ALGORITHM 1: CC-Synch

```
1  structures:
2    node_t      { request_t req, ret_val_t ret, boolean wait, boolean completed, node_t *next };

   // 'lock' represents the lock. It is the tail of a shared queue that initially contains a dummy node with the values
   // {null, null, false, false, null}. 'node' initially points to a thread-local node with the values {null, null, false, false,
   // null}.
3  function execute_cs(request_t req, node_t **lock, node_t **node) : ret_val_t
4      var node_t *next_node, *cur_node, *tmp_node, *tmp_node_next;
5      var int counter := 0;
6      next_node := *node;                          // The current thread uses a (possibly recycled) node
7      next_node → next := null;
8      next_node → wait := true;
9      next_node → completed := false;
   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
10     cur_node := atomic_swap(*lock, next_node);   // 'cur_node' is assigned to the current thread
11     cur_node → req := req;                       // The current thread announces its request
   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
12     cur_node → next := next_node;
13     *node := cur_node;
14     while cur_node → wait do                     // The current thread busy-waits until it is unlocked
15         pause();
   ................................................................................................
16     if cur_node → completed then
17         return cur_node → ret;                   // If the request is already applied, return its value
18     tmp_node := cur_node;                        // The current thread is the combiner
19     while tmp_node → next ≠ null and counter < MAX_COMBINER_OPERATIONS do
20         counter++;
21         tmp_node_next := tmp_node → next;
   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
           Critical section:
22         <apply tmp_node → req to object's state and store the return value to tmp_node → ret>
   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
23         tmp_node → completed := true;            // tmp_node's req is applied
   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
24         tmp_node → wait := false;                // Signal the client thread
25         tmp_node := tmp_node_next;               // Proceed to the next node
26     tmp_node → wait := false;
27     return cur_node → ret;
```

.... Acquire fence   - - - - - - - Release fence

The algorithm of CC-Synch is shown in Algorithm 1.[3] The queue initially contains a dummy node whose variables `wait` and `completed` are both set to **`false`** (line 2). To

[3] The memory fences in Algorithms 1, 2, and 4 are not required on architectures that ensure Total Store Order (TSO), but may be needed for weaker memory models. The ×86 and SPARC machines we used in the evaluation (Section 5) ensure TSO.

execute a critical section (lines 6–13), a thread sets its `wait` and `completed` variables to **`true`** and **`false`**, respectively. The thread then exchanges its node with the dummy node, which is always located at the end of the list: as a result, the thread's old node becomes the new dummy node. Finally, the thread writes the address of the function that encapsulates the critical section into its new node and makes it point to its old node (it may not be the dummy node anymore at that point), which ensures that its node has been enqueued in the list. Note that because the first thread that enqueues itself has both its `completed` and `wait` variables set to **`false`**, it will become a combiner (its critical section is not completed, but it will not wait for another combiner, therefore, it must become one).

Once a thread has enqueued its node in the list, it busy-waits while its `wait` variable equals **`true`** (lines 14 and 15). After this step, it checks whether its `completed` variable is set to **`true`** (lines 16 and 17): if so, the critical section has been executed by a combiner. Otherwise, the thread becomes a combiner, and goes through the queue, executing each critical section and setting the `completed` and `wait` flags of the corresponding node to **`true`** and **`false`**, respectively, afterward (lines 19–25). When the thread reaches the last element in the queue, or when it has executed a bounded number of critical sections, it sets the `wait` variable of the current element in the queue to **`false`**, and leaves the value of its `completed` variable as **`false`** (line 26). As the latter node will have both its `completed` and `wait` variables set to **`false`**, (i) the corresponding thread becomes the next combiner if the node belongs to a thread or (ii) the initial state of the dummy node is restored as the list contains a single node.

DSM-Synch is a variant of CC-Synch that aims to improve its memory locality. With CC-Synch, nodes are continuously exchanged between threads. Consequently, a thread's current node is allocated by another thread most of the time, and is therefore likely to be located on a remote memory bank on NUMA or non-cache-coherent architectures. DSM-Synch adds some complexity to ensure that a thread always reuses the same node. The result is that threads always busy-wait on variables from nodes that can be allocated in a local memory bank, which may improve memory locality at the price of a few more more atomic instructions on the critical path.

2.3.4. Analysis. We now analyze the properties of combining locks.

Lock Management. Combining locks have a shorter critical path than queue locks if combiners manage to execute long enough sequence of critical sections: a combiner can execute critical sections one after the other without synchronization between threads, as long as the hardware prefetcher manages to bring nodes to the local caches before they are accessed (otherwise cache misses may occur). Oyama, however, has significant additional overhead on the critical path, because all threads access the global synchronization variable concurrently, often using costly atomic instructions. Moreover, the combiner has to detach the stack regularly to execute a long sequence of critical sections if new threads keep enqueueing themselves while critical sections are being executed. For Flat Combining, when the combiner reaches the head of the linked list, it releases the lock, which is acquired again by a new thread that becomes the new combiner when there are still threads waiting on the lock (the fact that the combiner has reached the head of the linked list does not mean that there are no more pending critical sections because threads enqueue themselves at the tail). Releasing and acquiring this lock also slows down the critical path. Moreover, the cleanup operations also take place on the critical path because the combiner has to have exclusive access to the linked list in order to clean it up. For CC-Synch and DSM-Synch, if nodes are prefetched correctly, the only overhead on the critical path takes place when the role of combiner is handed over from one thread to the next, which results in an invalidation and a cache miss.[4]

Data Locality. Combining locks improve data locality: critical sections that are protected by a given lock often perform operations on the same set of shared variables and executing several of them consecutively on the same hardware thread makes it possible for these variables to remain in that hardware thread's caches. As a result, fewer cache misses occur on the critical path and performance is improved.

Performance with Many Threads. While Oyama and Flat Combining perform well when there are more threads that use the lock than there are hardware threads on the machine, CC-Synch and DSM-Synch do not, because of a design flaw in their algorithm that we illustrate with CC-Synch. Suppose that a thread t is preempted between lines 10 and 12 of Algorithm 1. The current combiner will only be able to execute critical sections up to that node and will hand over the role of combiner to the preempted thread t, because the list is split into two parts until the `next` pointer of t's `cur_node` is set. No progress will be made on the critical path until t wakes up. This issue occurs frequently when there are more threads that try to acquire the lock than there are hardware threads. It causes performance to collapse, as will be shown in Section 5.4.5.

Support for Condition Variables. Condition variables cannot easily be implemented for combining locks because by making the combiner sleep, they prevent it from executing the remaining critical sections in the queue. Moreover, since server threads are normal application threads, waiting on condition variables of other threads prevents them from making progress, which can cause undesirable unexpected effects such as deadlocks.

Fairness. Oyama uses a stack (LIFO order), which makes it especially unfair because a node's request can always be delayed by a more recent node insertion. Flat Combining uses an unordered linked list, which makes the algorithm more fair. CC-Synch and DSM-Synch, like queue locks, use a queue (FIFO order), which makes them perfectly fair.

2.4. Other Lock Algorithms

We have described the main families of locks and presented members of each family that were either notable due to their widespread use (spinlocks, blocking locks), historical reasons (CLH and Oyama were the first of their kind), or their efficiency (all other presented lock algorithms). We now give an overview of other existing lock algorithms.

Backoff locks improve on the basic spinlock, with the objective of achieving better performance when many threads perform concurrent lock acquisition attempts. The main idea of backoff locks is to make threads sleep for a *backoff delay* in the busy-wait loop. Doing so reduces contention and also has the advantage of saving power and CPU time. The delay typically increases at each iteration, often linearly (*linear backoff lock*) or exponentially (*exponential backoff lock*). According to Anderson [1990], increasing the delay exponentially is the most efficient strategy. The ticket lock [Reed and Kanodia 1979] also aims to improve the performance of spinlocks under high contention. It uses two shared counters: one containing the number of lock acquisition requests, and the other one containing the number of times the lock has been released. In order to acquire

[4] We consider that when `tmp_node→wait` is set **`false`**, the corresponding cache line is invalidated and the thread that is busy-waiting on that synchronization variable must fetch it again. The actual low-level behavior may differ depending on the cache-coherency protocol used.

the lock, a thread reads and increments the value of the first counter with an atomic fetch-and-add instruction and busy-waits until the second counter is equal to the value it has read from the first counter. The advantage the ticket lock has over the basic spinlock or backoff locks is that threads busy-wait on the second counter using an instruction that only reads the corresponding cache line instead of an instruction that attempts to write to it (e.g., a compare-and-swap instruction). Mellor-Crummey and Scott [1991b] show that both backoff locks and the ticket lock are slower than MCS under high contention. However, David et al. [2013] show that the ticket lock performs better than a wide range of other locks under low contention, and, given its small memory footprint, they recommend its use over more complex lock algorithms unless it is sure that a specific lock will be very highly contended.

Abellán et al. [2011] propose GLocks in order to provide fast, contention-resistant locking for multicore and manycore architectures. The key idea of GLocks is to use a token-based message passing protocol that uses a dedicated on-chip network implemented in hardware instead of the cache hierarchy. Since the resources needed to build this network grow with the number of supported GLocks, Abellán et al. recommend to only use them for the most contended locks and to use spinlocks otherwise. The main drawback of GLocks is that they require specific hardware support not provided by current machines.

Finally, given the large number of proposed lock algorithms, choosing one is not always a simple task. Smartlocks [Eastep et al. 2010] aims to solve this issue by dynamically switching between existing lock algorithms in order to choose the most appropriate one at runtime. They use heuristics and machine learning in order to optimize toward a user-defined goal, which may relate to performance or problem-specific criteria. In particular, on heterogeneous architectures, Smartlocks is able to optimize which waiter will get the lock next for the best long-term effect when multiple threads are busy-waiting for a lock. Since Smartlocks relies on a set of lock algorithms, however, it does not remove the need for designing efficient lock algorithms in the first place.

Hierarchical locks trade fairness for throughput by executing several critical sections consecutively on the same *cluster* of hardware threads (core, die, CPU, or NUMA bank). Doing so allows for better throughput, since synchronization local to a cluster is faster than global synchronization, for two reasons: (i) critical sections executed on the same cluster can reuse shared variables that are stored in their common caches, and (ii) the synchronization variables used for busy-waiting are allocated on a local NUMA node, which may reduce the overhead of busy-waiting, as will be shown in Section 5.1. The Hierarchical Backoff Lock (HBO) [Radovic and Hagersten 2003] is a backoff lock algorithm with an adaptive delay that favors hardware threads of the same cluster: these threads are granted shorter backoff times, whereas remote hardware threads are granted longer backoff times. The Hierarchical CLH lock (H-CLH) [Luchangco et al. 2006] creates a CLH-style queue for each cluster, and the thread at the head of each local queue occasionally splices the local queue into a global queue. Critical sections are executed following the global queue, as if it were a traditional CLH queue. However, given the way the global queue is built, nodes from the same cluster are neighbors in the global queue and their critical sections are executed consecutively.

Another example of a hierarchical lock has been proposed by Dice et al. [2011], using a combination of Flat Combining and MCS. Each cluster uses one instance of Flat Combining that efficiently creates local MCS queues of threads, and merges them into a global MCS queue. The lock is handed over in the MCS queue exactly like with a MCS lock, except the global queue created by the combiners is ordered by clusters, like with H-CLH. However, the approach by Dice et al. is more efficient than H-CLH because with H-CLH, all threads need to enqueue themselves by using an atomic instruction that is applied to the global tail of the queue, which can cause bottlenecks. Moreover, with H-CLH, threads must know which thread is the master of their local cluster, which complicates their busy-waiting semantics.

Finally, Dice et al. [2012] propose Lock Cohorting, a general technique that makes it possible to build hierarchical locks from any two nonhierarchical lock algorithms G and S, as long as these lock algorithms satisfy certain (widespread) properties. The general idea is to use one instance of S per cluster, and one global instance of G. The first thread to acquire a lock acquires both G and S, and then releases only S if other threads from the same cluster are waiting for the lock (these threads will only have to acquire S to own the lock), otherwise, it releases both G and S to let threads from other clusters acquire the lock. Dice et al. use Lock Cohorting to build new hierarchical locks by choosing either a backoff lock, the ticket lock, MCS, or CLH for G and for S. They show that some of the resulting combinations outperform both HBO and H-CLH.

While hierarchical locks offer good performance, they are generally based on traditional lock algorithms. Consequently, they are a powerful optimization for lock algorithms, but do not replace them.

3. REMOTE CORE LOCKING

In order to improve the performance of applications on multicore architectures, we present RCL, a lock mechanism whose main idea is to dedicate a *server* hardware thread to the execution of critical sections. RCL has two main advantages. First, it reduces synchronization overhead, because the *client* threads and the server communicate using a fast client/server cache-aligned messaging scheme that is similar to the one used in Barrelfish [Baumann et al. 2009]. And second, RCL improves data locality, because the shared variables that the critical sections protect are all accessed from the same dedicated server hardware thread, where no application thread can be scheduled. Therefore, the shared variables are more likely to always remain in that hardware thread's cache hierarchy. RCL has additional advantages. In particular, the fact that the server hardware thread is dedicated to the execution of critical sections ensures that the server can never be preempted by application threads: it always makes progress on a critical path. Furthermore, the fact that critical sections are not implemented by application threads makes it possible for RCL to implement condition variables.

This section first gives an overview of RCL, by describing how it works in simple cases. We then focus on the RCL runtime: we describe implementation details, including how the runtime handles complex situations such as blocking in critical sections, nested critical sections, and waiting on condition variables. We then describe statistics that are gathered by the RCL runtime. Finally, we compare RCL with other locks.

3.1. Overview

With RCL, each critical section is replaced by a remote procedure call to a server that executes the code of the critical section. Communication between the client and server threads uses an array of mailboxes that store requests for the execution of critical sections. The following paragraphs describe the request array and how it is used by client and server threads to implement fast remote procedure calls.

Request Array. The request array is illustrated in Figure 1. It contains C elements of size L. C is a large number, typically much higher than the number of hardware threads.[5] Each element of the array stores a mailbox req_i that is used for the communication of client c_i with the server. In order to make communication as efficient

[5] To be more precise, in order to avoid the need to reallocate the request array when new client threads are created, its size is fixed and chosen to be very large (256KB), and a *client identifier allocator* implements an adaptive long-lived renaming algorithm [Brodsky et al. 2006] that keeps track of the highest client identifier and tries to reallocate smaller ones.

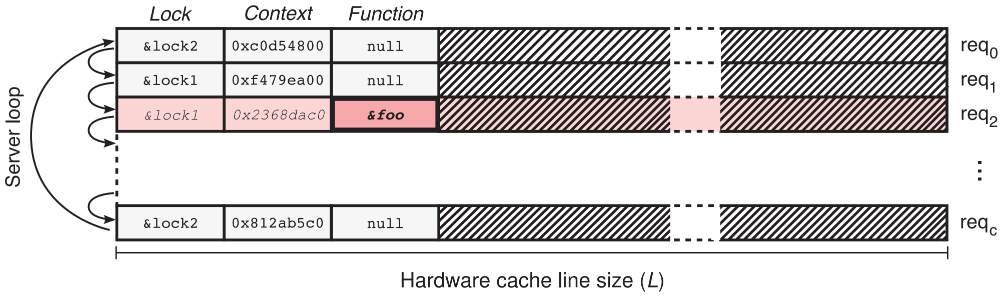


Fig. 1. The request array.

as possible, (i) C is cache aligned and L is chosen to be the hardware cache line size in order to avoid false sharing (if the underlying hardware uses multiple cache line sizes, the least common multiple is used), and (ii) on NUMA architectures, the array of mailboxes is allocated on the NUMA bank of the server to avoid unnecessary communication to an external node during client-server communication.

The first three machine words of each mailbox req_i contain, respectively, (i) the address of the lock associated with the critical section, (ii) the address of a structure encapsulating the *context*, that is, the variables referenced or updated by the critical section that are declared by the function containing the critical section code, and (iii) the address of a function that encapsulates the critical section for which the client c_i has requested the execution,[6] or **null** if no critical section execution is requested.

Client Side. In order to execute a critical section, a client c_i first writes the address of the lock into the first word of its structure req_i, then writes the address of the context structure into the second word, and finally writes the address of the function that encapsulates the code of the critical section into the third word. The client then busy-waits for the third word of req_i to be reset to **null**, which indicates that the server has executed the critical section. Busy-waiting can be considered too energy demanding in some contexts: to address this issue, on some architectures, it is possible to write an energy-aware version of RCL where cores wait in power-saving mode instead of busy-waiting, as will be explained in Section 5.5.3. Alternatively, the busy-wait loop of the client may yield the processor at each iteration, as will be discussed in Section 5.4.5.

Server Side. A servicing thread iterates over the mailboxes, waiting for one of the mailboxes to contain a non-**null** value in its third word. When such a value is found, the servicing thread acquires the lock[7] and executes the critical section using the function pointer and the context. When the servicing thread is done executing the critical section, it resets the third word to **null**, and resumes iterating over the request array.

Issues Raised by RCL. If more than one lock is used by the application, the RCL server can handle the critical sections of several locks, but doing may result in *false serialization*, that is, the unnecessary serialization of independent critical sections. To alleviate this issue, more servers can be used, each of them executing the critical sections of one or several locks. In practice, it is not necessary to use a lot of servers to improve performance, because RCL is not meant to be used for all locks: since RCL's strongest point is its high performance under high contention, and the lock algorithms

[6] Since RCL relies on function pointers to ship the code of critical sections from client to server threads, it cannot be used for interprocess mutual exclusion in its current form.

[7] Since a server executes all critical sections for a given lock, the lock is always free when it tries to acquire it, except in the case of nested critical sections.

used for uncontended locks have negligible impact on performance, RCL should only be used for heavily contended locks. As shown in Section 5.4.1, the number of contended locks in the applications used in the evaluation is low enough that less than three servers are needed to reach optimal performance. Since modern multicore architectures provide dozens of hardware threads whose processing power cannot always be harvested efficiently because applications lack scalability, many hardware threads are often left unused by applications: these hardware threads can be used for RCL servers in order to improve performance.

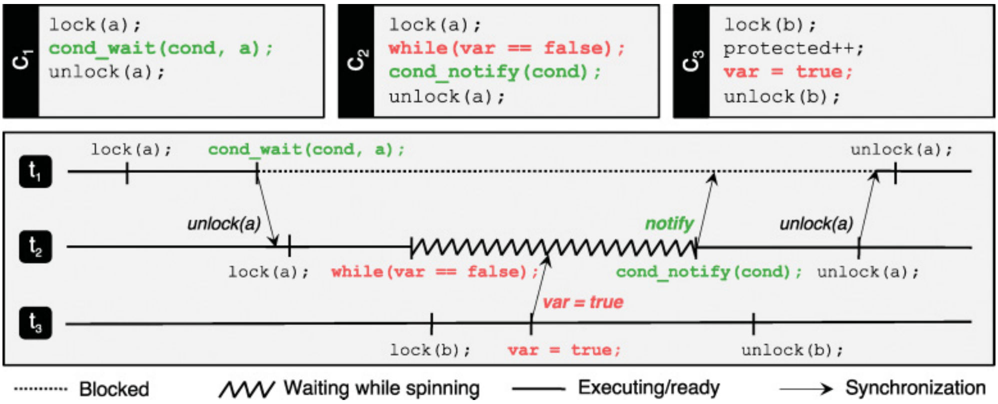
3.2. Implementation of the RCL Runtime

The core algorithm, described in Section 3.1, only refers to a single servicing thread, and thus requires that this thread is never blocked at the operating system level and never busy-waits. We want to ensure that RCL works seamlessly and efficiently (i) with legacy applications, which may use blocking or busy-waiting inside critical sections, and (ii) on existing operating systems, which were not designed with exotic synchronization schemes in mind. Because of these two constraints, the RCL implementation has to rely on some low-level tricks, such as using POSIX FIFO scheduling in a nonstandard way.

In this section, we describe how the RCL runtime extends the core algorithm to always ensure liveness and responsiveness in legacy applications on mainstream operating systems, and we present the pseudocode of key parts of the RCL runtime. We later describe statistics that are gathered by the RCL runtime and that can be used to help optimize the placement of locks on servers.

3.2.1. Ensuring Liveness and Responsiveness. Three kinds of situations may induce liveness or responsiveness issues if the server uses a single servicing thread. First, the servicing thread may be blocked at the operating system level. This can happen when a critical section tries to acquire a blocking lock (e.g., a POSIX lock on Linux or Solaris) that is already held, performs I/O, or waits on a condition variable, for instance. Second, the servicing thread may enter a busy-wait loop if a critical section tries to acquire a nested RCL or a lock that uses busy-waiting, or if it uses some other form of ad hoc synchronization [Xiong et al. 2010]. Finally, the servicing thread may be preempted at the operating system level, either because its timeslice expires [Ousterhout 1982] or because of a page fault. Blocking and waiting within a critical section may cause a deadlock, because the servicing thread is unable to execute critical sections associated with other locks, even when doing so may be necessary to allow the blocked critical section to unblock. Additionally, blocking, of any form, including waiting and preemption, degrades the responsiveness of the server because a blocked thread is unable to serve other locks managed by the same server. In order to solve these issues, RCL uses a pool of servicing threads on each server to ensure liveness and responsiveness, as is described next.

Ensuring Liveness. Blocking and waiting within a critical section raise a problem of liveness, because a blocked servicing thread cannot execute critical sections associated with other locks, even when doing so may be necessary to allow the blocked critical section to unblock. A pathological case happens when the servicing thread busy-waits in a critical section while waiting for a variable to be set by another critical section that is handled by the same server but is protected by a different lock: this situation is illustrated by clients c_2 and c_3 in Figure 2. To ensure liveness, a pool of servicing threads is maintained on each server to ensure that when a servicing thread blocks or waits, there is always at least one other *free* servicing thread that is not currently executing a critical section, and that this servicing thread will eventually be scheduled. To ensure the existence of a free servicing thread, the RCL runtime provides a *management thread*, which is activated regularly at each expiration of a *timeout* (set to the operating



```
lock(a);
cond_wait(cond, a);
unlock(a);
```

```
lock(a);
while(var == false);
cond_notify(cond);
unlock(a);
```

```
lock(b);
protected++;
var = true;
unlock(b);
```

Fig. 2. Ad hoc synchronization example.

system's timeslice value) and runs at highest priority. When activated, the management thread checks that at least one of the servicing threads has made progress since the last activation of the management thread, using a server-global flag is_alive. The management thread clears this flag just before sleeping, and any servicing thread that enters a critical section sets it. If the management thread observes that is_alive is cleared when it wakes up, it suspects that all servicing threads are either blocked or waiting. In this case, it checks that no free thread indeed exists in the pool of servicing threads and if so, it adds a new one.

Ensuring Responsiveness. Blocking, of any form, including waiting and preemption, degrades the responsiveness of the server because a blocked servicing thread is unable to serve other locks managed by the same server. The RCL runtime implements a number of strategies to improve responsiveness issues introduced by the underlying operating system and by RCL design decisions.

As was explained in Section 2, a well-known problem in the use of locks is the risk that the operating system will preempt a thread at the expiration of a timeslice while it is executing a critical section, thereby extending the duration of the critical section and increasing contention [Ousterhout 1982]. RCL dedicates a pool of threads on each dedicated server hardware thread to the execution of critical sections, which makes it possible to manage these threads according to a scheduling policy that does not use preemption. The POSIX FIFO scheduling policy is used, because it both respects priorities, as is needed to ensure liveness, and allows threads to remain scheduled until they are blocked or manually yield the processor. The use of the FIFO policy, however, raises a liveness issue: if a servicing thread is executing a busy-wait loop, it will never be preempted by the operating system, and a free thread will never be scheduled. To solve this, when the manager thread detects no progress, it makes sure a servicing thread is scheduled by first decrementing and then incrementing the priorities of the other threads, effectively moving them to the end of the FIFO queue. The use of the FIFO policy also implies that when a servicing thread unblocks after blocking in a critical section, it is placed at the end of the FIFO queue. If there are many servicing threads, there may be a long delay before the unblocked thread is rescheduled. To minimize this delay, the RCL runtime tries to always minimize the number of servicing threads: each servicing thread regularly checks that there are no other free servicing threads, and if there are, it leaves the pool, since another thread will be able to handle requests.

The RCL management thread ensures the liveness of the server but only reacts after a timeout. When all servicing threads are blocked in the operating system, the operating system's scheduler is exploited to schedule a new free thread immediately.

Concretely, the RCL runtime maintains a *backup thread* that runs at a lower priority than all servicing threads. The FIFO scheduling policy never schedules a lower priority thread when a higher priority thread that is not blocked exists, and thus the backup thread is only scheduled when all servicing threads are blocked. When the backup thread is scheduled, it adds a new servicing thread, which immediately preempts the backup thread and can service the next request.

Finally, when a critical section needs to execute a nested RCL managed by the same hardware thread and the lock is already owned by another servicing thread, the servicing thread yields the processor in order to allow the lock owner to release it.

Example. Figure 2 presents a complete example of ad hoc synchronization in a critical section that illustrates liveness and responsiveness issues. Initially, thread t_1 is the only servicing thread, and it handles a critical section of client c_1. That critical section tries to acquire a condition variable (`cond_wait()` function call), causing t_1 to block at the operating system level. At that point, there is no more runnable servicing thread. Therefore, the operating system's scheduler immediately schedules the backup thread, which adds a new free servicing thread t_2 that immediately preempts the backup thread. Thread t_2 executes c_2's request. Client c_2's critical section causes t_2 to busy-wait on the variable `var`. At some point, the management thread awakens, and clears the server-global `is_alive` flag. Then, at the management thread's next activation, since t_1 is blocked in the operating system and t_2 is busy-waiting, no servicing thread has been able to set `is_alive`. Since there is no free servicing thread at that point, the management thread creates a new servicing thread, t_3, and schedules it, as it is now the only servicing thread that has not recently been scheduled. Therefore, at that point, the server has three servicing threads, t_1, t_2, and t_3, one of which, t_3, is free. When t_3 executes the request from c_3, it modifies the value of `var`.

After executing client c_3's request, thread t_3 detects that it is not the only free servicing thread in the pool: while t_1 is blocked at the operating system level, t_2 is free, therefore, t_3 yields the processor in order to let t_2 be scheduled and make progress. Since the critical section executed by t_3 has set `var`, thread t_2 can exit its busy-wait loop and unblock the condition variable, causing t_1 to awaken. Client c_2's critical section ends, and t_2 detects that the server has two free servicing threads: itself and t_3. Consequently, thread t_2 leaves the servicing pool. Thread t_3 is then scheduled, but it notices that there is no pending critical section. Furthermore, since it notices that another servicing thread is in the servicing pool and that thread is not free, it yields the processor to let it make progress. Thread t_1 is therefore scheduled, and it completes client c_1's request, before noticing that two servicing threads are free, itself and t_3. Consequently, it leaves the pool of servicing threads: the server is back to its initial state, except its only servicing thread is now t_3.

3.2.2. Detailed Implementation. We now present the pseudocode of key parts of the RCL runtime. Algorithms 2 and 3 show the functions that can be used by clients to execute critical sections with RCL. Algorithm 4 show the pseudocode of the servicing threads, and Algorithm 5 shows the pseudocode of the manager and backup threads.

Executing a Critical Section. We provide a function named `execute_cs_lock()` that executes a critical section using RCL. When a client thread runs the `execute_cs_lock()` function, it submits its request by filling the `lock`, `context`, and `function` fields of its `request` structure (the mailbox) in the `requests` array, as seen in lines 14–20 of Algorithm 2. On a RCL server, it is possible that during the execution of a critical section on a servicing thread, the execution of a nested critical section is requested, either (i) on the local RCL server if the lock of the nested critical section is handled by the same RCL server as the lock of the outer critical section, or (ii) on a

ALGORITHM 2: Executing a critical section, “lock” mode (client)

```
thread-local variables:
    int $client\_index$;                    // An integer value that identifies this thread
    boolean $is\_server\_thread$;           // Is this thread a servicing thread, that is, not a client thread?
    server_t *$my\_server$;                 // If this thread is a servicing thread, a pointer to the corresponding server
function $execute\_cs\_lock$(lock_t $*lock$, function_t $*function$, void $*context$)
    var int $real\_me$;                     // The identifier of the client thread from which the critical section originates
    var request_t *$request$                // A pointer to the mailbox that will be used to execute the critical section
    $request := \&lock \rightarrow server \rightarrow requests[client\_index]$;
    if $\neg is\_server\_thread$ then
        $real\_me := client\_index$
    else
        $real\_me := my\_server \rightarrow requests[client\_index] \rightarrow real\_me$
    if $\neg is\_server\_thread$ or $my\_server \neq lock \rightarrow server$ then
    // RCL to a remote hardware thread
        $request \rightarrow lock := lock$;
        $request \rightarrow context := context$;
        $request \rightarrow real\_me := real\_me$;
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        $request \rightarrow function := function$;
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        while $request \rightarrow function \neq$ null do
            $pause()$;
.................................................................................
        return;
    else                                                          // Local nested lock
        while $local\_compare\_and\_swap(\&lock \rightarrow holder, NONE, real\_me) = real\_me$ do
            $yield()$;                                            // Give a chance to other thread to release lock
.................................................................................
        $context := function(context)$;                           // Execute the critical section
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        $lock \rightarrow holder := NONE$;
        return;
```

```
..........  Acquire fence        - - - - - - - -  Release fence
```

ALGORITHM 3: Executing a critical section, “trylock” mode (client)

```
thread-local variables:
    int $client\_index$;                    // An integer value that identifies this thread
    boolean $is\_server\_thread$;           // Is this thread a servicing thread, that is, not a client thread?
    server_t *$my\_server$;                 // If this thread is a servicing thread, a pointer to the corresponding server
function $execute\_cs\_trylock$(lock_t $*lock$, function_t $*function$, void $*context$)
    var int $real\_me$;                     // The identifier of the client thread from which the critical section originates
    if $\neg is\_server\_thread$ then
        $real\_me := client\_index$
    else
        $real\_me := my\_server \rightarrow requests[client\_index] \rightarrow real\_me$
    if $lock \rightarrow holder$ then
        if $lock \rightarrow is\_recursive$ and $lock \rightarrow holder = real\_me$ then
            $((char*)context)[0] =$ true;                // Recursive lock already acquired by me: trylock success
        else
            $((char*)context)[0] =$ false;               // Lock already acquired: trylock failure
        $context := function(context)$;                 // Execute function, lock may not have been acquired
    else
        $((char*)context)[0] =$ true;
        $execute\_cs(lock, function, context)$;         // Lock seems free, acquire (may busy-wait)
```

remote RCL server, if the two locks are handled by distinct RCL servers. Servicing threads use the same method as clients to request the execution of a critical section that is managed by a remote RCL server. However, in order to request the execution of a critical section that is managed by the local RCL server, a servicing thread must ensure that the lock is free, and if not, wait until it is free. In order to give a chance to the servicing thread that owns the lock to finish executing its critical section, the thread repetitively yields the processor (lines 22 and 23) while it busy-waits for the lock.

"Trylock" Mode. While the `execute_cs_lock()` function can be used instead of traditional *lock()* and *unlock()* operations to execute a critical section using RCL, the `execute_cs_trylock()` function can be used instead of a traditional *trylock()* operation, such as `pthread_mutex_trylock()` on POSIX systems. The pseudocode of `execute_cs_trylock()` is shown in Algorithm 3. Like `execute_cs_lock()`, `execute_cs_trylock()` takes three parameters: a lock, a function, and a context object. The first byte of the context object must be reserved since it will store a Boolean value that is equivalent to the return value of a traditional *trylock()* operation.

The `execute_cs_trylock()` function first checks whether the lock is taken; if so, it stores the value **`false`** into the first byte of the context object to signal the client thread that the lock was not acquired (line 15), and it runs the function locally (line 16). If the lock was free, however, `execute_cs_trylock()` stores **`true`** into the first byte of the context object (line 18) and executes the critical section using RCL through a call to `execute_cs_lock()` (line 19). The `execute_cs_trylock()` function also handles recursive locks, in a similar way to `pthread_mutex_trylock()` used with a `PTHREAD_MUTEX_RECURSIVE` POSIX lock: if the lock's `is_recursive` field is set to **`true`** and the lock has already be acquired previously in a parent critical section, `execute_cs_trylock()` will store **`true`** in the first byte of the context object and run the new critical section locally (line 16).

Note that while `execute_cs_trylock()` is useful to support legacy applications that use calls to `pthread_mutex_trylock()`, its behavior differs from a traditional trylock: a client *c* that calls `execute_cs_trylock()` may have to wait for a significant amount of time before its critical section is executed, since the server may execute other critical sections before *c*'s.

Servicing Threads. Algorithm 4 shows the pseudocode of servicing threads. Only the *fast path* (lines 12–25) is executed when the pool of servicing threads contains a single thread. A *slow path* (lines 26–33) is executed when the pool contains several servicing threads.

Lines 12–23 of the fast path implement the RCL server loop described in Section 3.1. Note that even though a large memory block is allocated for the request array (256KB), we only poll its first elements as was explained in Section 3.1 (the last element we poll corresponds t the client thread with the highest identifier, and identifiers are reused to remain low). The servicing thread indicates that it is not free while it is executing the server loop by decrementing (line 14) and incrementing (line 25) `number_of_free_threads`. Because the thread may be preempted due to a page fault, all operations on variables shared between the threads, including `number_of_free_threads`, must be atomic. However, since servicing threads of a RCL server are all bound to that server's dedicated hardware thread, the atomic instructions that only use server-local data only need to be executed atomically in the context of the threads running on the local hardware thread. Therefore, we use hardware-thread-local versions of the atomic operations that do not request machine-wide ownership of the cache lines they are working on (`local_fetch_and_add()` and `local_compare_and_swap()` in the pseudocode). In ×86 assembly, this is done by issuing atomic instructions without the `LOCK` prefix. These local atomic operations are never contended and they are much less costly than regular atomic instructions because they do not require additional synchronization between hardware threads.

ALGORITHM 4: Structures and servicing thread (server)

```
structures:
    // "Lock" type: contains a pointer to the server handling the lock, and the identifier of the lock holder.
    lock_t:       { server_t *server, int holder };

    // "Request" type: contains pointers to (i) the function that encapsulates the critical section, (ii) the context object, and
    // (iii) the lock that protects the critical section.
    request_t:    { function_t *function, void *context, lock_t *lock };

    // "Servicing thread" type: contains (i) a pointer to the server to which the servicing thread belongs, (ii) a timestamp that
    // helps determine whether a servicing thread has not recently been able to handle new requests, and (iii) a boolean that
    // specifies whether the servicing thread is currently servicing requests.
    thread_t:     { server_t *server, int timestamp, boolean is_servicing };

    // "Server" type: contains (i) a list of all threads running on the server, (ii) a list of "prepared" servicing threads, i.e.,
    // servicing threads that were deactivated because they were not needed anymore (they may be activated again if more
    // active servicing threads are needed in the future), (iii) the total number of servicing threads, (iv) the number of
    // servicing threads that are not currently executing critical sections, (v) the request array, (vi) a timestamp and (vii) a
    // boolean. The two last parameters are used to determine whether the server is currently making progress so that
    // servicing threads can be created/activated if needed.
    server_t:     { List<thread_t *> all_threads, LockFreeStack<thread_t *> prepared_threads,
                      int number_of_servicing_threads, int number_of_free_threads, request_t *requests,
                      int timestamp, boolean is_alive }

global variables:
    int last_client_id;                                                  // Identifier of the last client

function servicing_thread(thread_t * thread)
    var server_t *server := thread→server;                               // Server this servicing thread belongs to

    // The currently used mailbox, lock, and critical section (whose code is encapsulated in a function)
    var request_t *request, lock_t *lock, function_t *function;

    while true do
        server→is_alive := true;
        thread→timestamp := server→timestamp;

        local_fetch_and_add(server→number_of_free_threads, −1);           // This thread is not free anymore

        for i := 0 to last_client_id do
            request := server→requests[i];

            if request→function ≠ null then
··························································································
                lock := request→lock;

                if local_compare_and_swap(&lock→holder, NONE, request→real_me) = false then
··························································································
                    function := request→function;                       // Make sure function not modified:
                    if function ≠ null then                             // should not happen with FIFO scheduling
··························································································
                        Critical section:
                        request→context := function(request→context);
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                        request→function := null;                       // Signal completion to the client
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                    lock→holder := NONE;

        local_fetch_and_add(server→number_of_free_threads, 1);            // This thread is now free

        if server→number_of_servicing_threads > 1 then                    // Some servicing threads are blocked
            if server→number_of_free_threads ≤ 1 then
                yield();                                                // Allow other servicing threads to progress

            else
                thread→is_servicing := false;                           // Keep only one free servicing thread
                local_fetch_and_add(server→number_of_servicing_threads, −1);
                local_atomic_insert(server→prepared_threads, thread);

                // Atomic since the manager may wake up thread before call to sleep() (on Linux, use futexes)
                atomic(<if ¬thread→is_servicing then sleep();>)
```

·················· Acquire fence - - - - - - - Release fence

In the fast path, a servicing thread executes the following operations for each client. It first retrieves its mailbox (line 16), and checks if the execution of a critical section has been requested by this client (line 17). If this is the case, the server tries to acquire the corresponding lock (line 19). In lines 20 and 21, the server makes sure that no other servicing thread has executed the critical section before the lock acquisition, which should never occur with FIFO scheduling (no preemption during the fast path), but may occur in the degraded version of RCL that we will describe later in this section. Once the server holds the lock and knows which function to execute, it executes the critical section, then signals to the client that its critical section has been executed by resetting `function` to **`null`**.

The slow path is executed if the active servicing thread detects the existence of other servicing threads (line 26). If the other servicing threads are all executing critical sections (line 27), the servicing thread yields the processor (line 28) so that they can be scheduled and make progress. Otherwise, it makes itself inactive (line 30), updates `number_of_servicing_threads` accordingly (line 31), removes itself from the pool (line 32), and sleeps (line 33). We make sure that if a manager thread sets `is_servicing` to **`true`** at the last moment, the thread will not sleep, by using a Futex on Linux, for instance. This is needed to avoid a race condition: a lost update of the `is_servicing` variable could, in this case, render `number_of_servicing_threads` invalid.

Management and Backup Threads. As explained in Section 3.2.1, each RCL server runs one management thread and one backup thread, whose pseudocodes are shown in Algorithm 5. If, on wake up, the management thread notices, based on the value of `is_alive`, that none of the servicing threads has made progress since the previous timeout, it ensures that at least one free thread will eventually exist. This is done in lines 8–19, by first incrementing `number_of_servicing_thread` and `number_of_free_threads` (a new, free servicing thread will eventually be available), and then by either creating a new servicing thread (lines 13–16), or by waking up a thread that is not currently servicing (lines 18 and 19).

Following this, the management thread elects a servicing thread that has not been scheduled recently (lines 20–26). To this end, the management thread and servicing threads all possess a timestamp counter. The server uses a timestamp counter that is increased every time it went through all threads without being able to find a servicing thread that has not been scheduled recently. Threads that are not blocked or busy-waiting in a critical section will eventually store this new value from the server's timeout into their own timeout since that is what they do every time they enter their fast path (line 10 of Algorithm 4). Servicing threads that are blocked will all be woken up by the server thread, one after the other, until a new cycle begins (i.e., the server's `timestamp` variable is incremented on line 26). This technique ensures that all servicing threads that are blocked or busy-waiting are eventually given a chance to make progress.

The backup thread (lines 30–33) simply sets `is_alive` to **`false`** and wakes up the management thread. Due to its low priority, the backup thread is scheduled when all other threads are sleeping, ensuring that a new servicing thread will quickly be available.

Degraded Version. Using FIFO scheduling may require special privileges, such as processes being launched using `CAP_SYS_NICE` on Linux or the `proc_priocntl` privilege on Solaris 10. We propose a degraded version that does not use backup or management threads on RCL servers, and servicing threads can preempt each other when they are scheduled on the same server hardware thread. While this degraded version of RCL performs worse when several locks are handled by the same RCL server and when

ALGORITHM 5: Management and backup threads (server)

```
function management_thread(server_t *server)
    var thread_t *thread;
    server→is_alive := false;
    server→timestamp := 1;
    while true do
        if server→is_alive = false then
            server→is_alive := true;
            if server→number_of_free_threads = 0 then  // Ensure a thread can handle remote requests
                // Activate prepared thread or create new thread
                local_fetch_and_add(server→number_of_servicing_threads, 1);
                local_fetch_and_add(server→number_of_free_threads, 1);
                thread := local_atomic_remove(server→prepared_threads);
                if thread = null then
                    thread := allocate_thread(server);
                    insert(server→all_threads, thread);
                    thread→is_servicing := true;
                    thread→start(PRIO_SERVICING);
                else
                    thread→is_servicing := true;
                    wakeup(thread);
            while true do                                   // Elect thread that has not recently been elected
                for thread in server→all_threads do
                    if thread→is_servicing = true and thread→timestamp < server→timestamp then
                        thread→timestamp = server→timestamp;
                        elect(thread);
                        goto end;
                server→timestamp++;                         // All threads were elected once, begin a new cycle
        else
            server→is_alive := false;
        end:
        sleep(TIMEOUT);
function backup_thread(server_t *server)
    while true do
        server→is_alive := false;
        <wake up the management thread>
```

threads busy-wait or block in critical sections, it still makes it possible to benefit from the full performance of RCL otherwise.

3.2.3. Statistics. The RCL runtime gathers a number of statistics described in the following. These statistics can be used to help optimize the placement of locks on servers, as will be shown in Section 5.4.5.a.

—*Cache misses.* The RCL server measures the average number of cache misses per critical section, optionally including cache misses caused by the RCL algorithm itself.

—*Use rate.* The use rate measures the server workload. It is defined as the total number of executed critical sections in iterations where at least one critical section is executed, divided by the number of client threads. Therefore, a use rate of 1.0 means that all elements of the array contain pending critical section requests, whereas a low use rate means that the server spends most of its time waiting for critical section execution requests.

—*False serialization rate.* The false serialization rate is defined as the average number of different locks that critical sections use during one active iteration of the server loop, divided by the number of client threads. It measures the amount of *false serialization*, that is, the needless serialization of independent critical sections that happens when one server handles several locks.

Other statistics are provided by the RCL runtime, such as the *slow path rate* that measures how often the server uses the slow path, as defined in Section 3.2.2; the number of times the manager thread gets woken up; the number of times the `is_alive` flag was not set (see Algorithm 5); and the total number of critical sections. These statistics can be useful for debugging and optimization, but are not used in the evaluation.

3.3. Comparison with Other Locks

We now discuss and compare the behavior of RCL and all lock algorithms presented in the previous section, with the exception of: (i) GLocks, because they cannot be run on existing architectures, (ii) Smartlocks, because they are not an actual lock algorithm, but a technique that makes it possible to switch between lock algorithms instead, and (iii) hierarchical locks, for the sake of simplicity, since a large number of hierarchical locks have been designed, often by extending a nonhierarchical lock algorithm or by combining several of them through Lock Cohorting [Dice et al. 2012].[8]

Reactivity/Performance under high contention. Locks that use busy-waiting are very reactive under high contention because they do not require context switches between the execution of critical sections, as is the case with blocking locks. The only exception to this is backoff locks since the waiting time before a thread tries to acquire a lock can be long under high contention. The basic spinlock performs very poorly under high contention due to the fact that it busy-waits using atomic instructions on a single synchronization variable. Backoff locks and ticket locks use two techniques to improve performance under high contention, namely, waiting before trying to acquire the lock or making sure that busy-wait loops are read only. Queue locks (CLH, MCS, and MCS-TP) also perform better than the basic spinlock because they use one synchronization variable per thread. Oyama and Flat Combining perform even better because they execute a series of critical sections without needing synchronization between them other than signaling each thread when its critical section has been executed. However, Oyama and Flat Combining still use a global lock. CC-Synch, RCL, and DSM-Synch remove the global lock completely. RCL has the added advantage that it never hands over the role of the server, which shortens the critical path even more.

Reactivity/Performance Under Low Contention. Most lock algorithms are able to acquire the lock instantly under low contention. Flat Combining sometimes has a nonnegligible overhead when contention is low because it occasionally goes through the whole list of threads after the execution of a critical section to disable inactive nodes. RCL needs to execute more operations than other lock algorithms under low contention: context variables have to be copied to a specific structure and back to their original addresses, a transfer of control is always needed, and a servicing threads has to poll as many mailboxes as there are threads before the critical section is executed. However, we will show in Section 5.3 that this overhead remains reasonable, especially considering that it is not located on a contended critical path.

Contended Atomic Instructions Issued on the Critical Path. The basic spinlock issues a lot of contended atomic instructions on the critical path when many threads busy-wait on the same synchronization variable. Backoff locks issue fewer atomic instructions

[8] Hierarchical locks will, however, be evaluated empirically, in relation to RCL, in Section 5.6.

because they wait in their busy-wait loop, and ticket locks even fewer because they only use an atomic instruction to exit the critical section. In all queue locks (CLH, MCS, MCS-TP), Oyama, and combining locks, threads use atomic instructions to insert their nodes into the global queue, but these atomic instructions are not located on the critical path. Oyama and Flat Combining use an internal global spinlock, which makes them use a significant amount of potentially contended atomic instructions on the critical path. RCL uses neither a global synchronization variable nor global queue: while it uses critical sections on the critical path, they are local to a single hardware thread and therefore never suffer from contention.

Ordering and Starvation. Most lock algorithms execute critical sections in FIFO order with the exception of (i) the basic spinlock and backoff locks, in which the fastest thread to request the lock acquires it, and (ii) Oyama, since it uses a LIFO queue. For this reason, basic spinlocks can lead to starvation (one thread that is faster than the others may always obtain the lock first). Oyama can also cause starvation if critical sections are added to the queue at a very high rate, that is, too fast for the thread that executes them to ever reach the end of the queue. Starvation is impossible for Flat Combining, because threads enqueue themselves at the head of the queue, that is, behind the combiner. CC-Synch and DSM-Synch use the parameter `MAX_COMBINER_OPERATIONS` to prevent starvation. With RCL, critical sections are not served in FIFO order: at each iteration of a servicing thread, critical sections are served following the ordering of threads in the request array. However, starvation is impossible, since the fact that a servicing thread loops over the request array ensures that when a thread t_1 asks for the execution of its critical section, at most one critical section from the same lock by another thread t_2 may be executed before the execution of t_1's critical section (at most one iteration of the server loop can be executed before reaching t_1's mailbox). According to the definitions of fairness and unfairness given in Chabbi et al. [2015], FIFO algorithms are fully fair, while RCL has an unfairness that is equal to the number of application threads minus one: this is the maximum number of critical sections that can be executed by threads t_i that posted their request after t_1, before t_1's critical section is finally executed.

Resistance to Preemption. All locks that use busy-waiting and a global queue are prone to convoys, as will be seen in Section 5.4.5.b. Therefore, their resistance to preemption is very low. The resistance to preemption of other locks is moderate, except for MCS-TP, which was specifically designed to be resistant to preemption and convoys. RCL is immune to the issue of preemption inside of a critical section since its server threads are always scheduled on a dedicated hardware thread and therefore cannot be preempted by an application thread: the server always makes progress.

Parameters. Locks that do not use parameters always run at optimal performance, while locks that use parameters may require fine-tuning to perform well. In particular, MCS-TP is hard to configure because it uses five parameters, and some of them, such as the upper bound on the length of critical sections, depend both on the architecture and the application used: such values can only be determined precisely through complex profiling. The parameters used by Flat Combining make it possible to choose between a lenient or aggressive cleanup policy for the queue, but both policies can be detrimental, therefore, choosing efficient parameters requires empirical evaluation. CC-Synch and DSM-Synch use a single parameter for which a large value can safely be chosen for good performance, even if too large a value may be detrimental to fairness. While RCL does not use any parameters, deciding how many servers to use and where to place each lock is sometimes needed to reach optimal performance. This process can be time-consuming, as will be shown in Section 5.4.5.

Data Locality of Internal Structures. The basic spinlock and the backoff locks have very poor data locality on their shared synchronization variable because all threads concurrently apply compare-and-swap operations on it: the shared variable “ping-pongs” between the caches of all cores. This effect is reduced in the case of the ticket lock since most threads only read the synchronization variable in their busy-wait loop. CLH has better locality because different threads busy-wait on different synchronization variables. MCS and MCS-TP have the advantage of only busy-waiting on local condition variables, even though this should not improve performance on modern cache-coherent architectures, where synchronization variables are brought to local caches during busy-wait loops. Oyama and Flat Combining use both a global lock and a local synchronization variable for each node in the global queue, therefore, their data locality is moderate for internal structures. CC-Synch uses one synchronization variable per thread, like CLH. DSM-Synch, like MCS, also ensures that each thread always busy-waits on its own queue node, therefore, it should waste less bandwidth and be more reactive on non-cache-coherent architectures.

Like MCS, MCS-TP, CLH, CC-Synch, and DSM-Synch, RCL uses one synchronization variable per client: the locality of mailboxes is similar to that of the thread nodes in other algorithms, with the advantage that mailboxes are cache-aligned and stored in an array, which avoids unnecessary cache misses, removes the possibility of false sharing, and facilitates prefetching. The fact that RCL servers are bound to dedicated hardware threads has additional advantages: all data and synchronization structures are allocated on the local NUMA bank, and no application thread can be allocated on the dedicated hardware thread, which reduces the possibility of application threads polluting the caches of the thread that executes critical sections.

Data Locality in Critical Sections. Oyama, Flat Combining, CC-Synch, and DSM-Synch all improve data locality by making threads execute sequences of critical sections: since critical sections of a given lock often protect a set of shared variables, these variables are likely to remain in a local cache during the execution of several critical sections. Similarly to combining locks, RCL improves data locality by making some threads execute sequences of critical sections. Since critical sections of a given lock often protect a set of shared variables, these variables may remain in a local cache during the execution of at least part of a sequence. However, RCL takes one step further by ensuring that these threads are bound to a specific server hardware thread. Therefore, the data they handle never has to be migrated between hardware threads: the shared data that is accessed by critical sections is likely to remain in the server’s caches during the whole execution. Moreover, the fact that no client thread may be scheduled on server hardware threads removes the risk of application threads polluting the caches with their data.

Usability in Legacy Applications. Legacy applications typically use blocking locks, such as the POSIX locks on Linux or Solaris, because in contrast with other lock algorithms, blocking locks work properly on architectures with a single hardware thread. Legacy applications can easily switch to the basic spinlock to backoff locks or to ticket locks, because they use the same interface as standard operating system locks (POSIX locks, for instance): their functions that acquire and release the lock take only one argument that represents the lock. In the case of queue locks, the functions that acquire and release the lock typically take two arguments: the list of requests for that particular lock (which can be seen as representing the lock itself), and the thread’s current node for that particular lock. While global thread-local variables can solve this issue, they are not available in all environments. Alternatively, the K42 lock [Auslander et al. 2003], a variant of MCS that takes a single argument that represents the lock, can be used. Implementing condition variables for the basic spinlock or queue locks using system primitives (POSIX primitives, for instance) is trivial. Finally, using Oyama or combining locks in legacy applications is difficult for two reasons. First, like RCL, these locks need critical sections to be encapsulated into functions, which requires a lot of code refactoring. We solved this problem, however, with our reenginering tool (presented in Section 4.2), which can be directly used for these lock algorithms. A more important problem, however, is that since server/combiner threads are normal application threads, any application thread could unpredictably block when it executes a critical section from another thread that blocks on a condition variable, which could cause undesirable unexpected effects such as deadlocks. On the other hand, RCL is able to handle condition variables thanks to a pool of threads on the servers, without risking putting the combiner to sleep as is the case with combining locks.

4. TOOLS

In this section, we describe two tools that were written to facilitate the use of RCL for application developers: a profiler that makes it possible to predict with reasonable accuracy which locks from which applications may benefit from RCL, and a reengineering tool that automatically transforms the code of legacy applications to allow them to use RCL.

4.1. Profiler

In order to help the user decide which locks to transform into RCLs, a profiler was implemented as a dynamically loaded library that intercepts calls involving POSIX locks, condition variables, and threads. An application that may benefit from RCL either suffers from high lock contention or its critical sections suffer from poor data locality. The profiler measures two metrics: (i) the overall percentage of time spent in critical sections including lock acquisitions and releases, which helps detect applications that suffer from high lock contention, and (ii) the average number of cache misses in critical sections, which helps detect applications whose critical sections suffer from poor data locality. As shown in the evaluation (Section 5), these metrics make it possible to reliably predict if an application can benefit from RCL. The profiler can also measure the two metrics for a specific lock, identified by its allocation point,[9] and thus provide per-lock information. Measuring the global time spent in critical sections and the global number of cache misses in critical sections make it possible to identify which applications may benefit from RCL, and per-lock information helps decide which locks to transform into RCLs in such an application.

4.2. Reengineering Legacy Applications

If the profiling results show that some locks used by the application can benefit from RCL, the developer must reengineer all critical sections that may be protected by the selected locks as a separate function that can be passed to the RCL server. This reengineering amounts to an "Extract Method" refactoring [Fowler 1999]. It was implemented with the program transformation tool Coccinelle [Padioleau et al. 2008], in 2,115 lines of code.

The main problem in extracting a critical section into a separate function is to bind the variables used by the critical section code. The extracted function must receive the values of variables that are initialized prior to the critical section and read within

[9] The profiler identifies locks with the file name and line number where they were allocated, and returns a list of the backtraces taken at the allocation points of each lock. Each lock is identified by a hash of the backtrace taken at its allocation point, and the profiler can be run again with the identifier of a lock in order to measure more precisely the two metrics for a particular lock.

LISTING 1: Critical section from Raytrace

```
1  int GetJob(RAYJOB *job, int pid)
2  {
3    ...
4    ALOCK(gm->wplock, pid)                                        /* Lock acquisition */
5    wpentry = gm->workpool[pid][0];
6
7    if (!wpentry) {
8      gm->wpstat[pid][0] = WPS_EMPTY;
9      AULOCK(gm->wplock, pid)                                        /* Lock release */
10     return (WPS_EMPTY);
11   }
12
13   gm->workpool[pid][0] = wpentry->next;
14   AULOCK(gm->wplock, pid)                                          /* Lock release */
15   ...
16 }
```

LISTING 2: Critical section from Listing 1, after transformation

```
union context {
  struct input { int pid; } input;
  struct output { WPJOB *wpentry; } output;
};

void cs(void *ctx) {
  struct output *outcontext = &(((union instance *)ctx)->output);
  struct input *incontext = &(((union instance *)ctx)->input);
  WPJOB *wpentry;
  int pid = incontext->pid, int ret = 0;

  /* Start of original critical section code */
  wpentry = gm->workpool[pid][0];
  if (!wpentry) {
    gm->wpstat[pid][0] = WPS_EMPTY;
    /* End of original critical section code */

    ret = 1;
    goto done;
  }
  gm->workpool[pid][0] = wpentry->next;
  /* End of original critical section code */

done:
  outcontext->wpentry = wpentry;
  return (void *)(uintptr_t)ret;
}

int GetJob(RAYJOB *job, int pid)
{
  int ret;
  union instance instance = { pid, };

  ...
  ret = execute_cs_lock(&gm->wplock[pid], &cs, &instance);
  wpentry = instance.output.wpentry;
  if (ret) { if (ret == 1) return (WPS_EMPTY); }
  ...
}
```

the critical section, and return the values of variables that are updated in the critical section and read afterward. Only variables local to the function are concerned; alias analysis is not required because aliases involve addresses that can be referenced from the server. Listing 1 shows a critical section from the Raytrace benchmark from the SPLASH-2 suite that will be presented in Section 5.1, and Listing 2 shows how it is transformed by the reengineering. The critical section of lines 4–14 of Listing 1 is protected by the `ALOCK()` and `AULOCK()` macros that are transformed by the C preprocessor into calls to the POSIX library functions `pthread_mutex_lock()` (lock acquisition) and `pthread_mutex_unlock()` (lock release), respectively. After transformation, the code of this critical section is encapsulated into the `cs()` function declared at line 6 of Listing 2, and a union named `context` whose instances will be able to hold the variables used by the critical section is defined at line 1 of Listing 2. To run the critical section, an instance of the `context` union is declared and filled with the values of the variables that are read by the critical section. The critical section is then run through a call to the `execute_cs_lock()` function from the RCL runtime (line 35 of Listing 2): this function takes three parameters: the lock, the address of the function that encapsulates the critical section, and the address of the instance of the `context` union. Finally, results are read from the `context` union (line 36 of Listing 2), since this union contains either the input (variables that are read) or the output (variables that are written) of the critical section, before and after the call to `execute_cs_lock()`, respectively. As an optimization, the context can be stored in the empty space at the end of the client's cache line (hatched space in Figure 1). The reengineering also addresses a common pattern in critical sections, illustrated in lines 7–11 of Listing 1, where a conditional statement in the critical section releases the lock and returns from the function. In this case, the code is transformed such that the critical section returns a flag value indicating which lock release operation ends the critical section, and the code following the call to `execute_cs_lock()` executes the code following the lock release operation that is indicated by the corresponding flag value (line 37 of Listing 2).

The reengineering tool also modifies various other POSIX functions to use the RCL runtime. In particular, the function for initializing a lock receives additional arguments indicating whether the lock should be implemented as an RCL. Finally, the reengineering tool also generates a header file, incorporating the profiling information, that the developer can edit to indicate which lock initializations should create POSIX locks and which ones should use RCLs. The header also makes it possible to choose which RCL server is used for each lock.

5. EVALUATION

This section describes how developers can use RCL to improve the performance of their applications, and evaluates the performance of RCL relative to other lock algorithms. All graphs and tables present data points and values that are averaged over five runs. Section 5.1 describes the machines used in the evaluation, and performs a comparative analysis of their sequential and parallel performance. Section 5.2 presents Liblock, a library that makes it possible to easily switch between lock algorithms in legacy applications. Liblock is used for all performance experiments in this evaluation. Section 5.3 describes a microbenchmark that is used to compare the performance of RCL with that of other lock algorithms. Section 5.4 presents a methodology to help developers detect which applications may benefit from RCL. This methodology is then applied to a set of legacy applications, and RCL as well as other lock algorithms are evaluated on the subset of applications that were identified as being good candidates for RCL. Section 5.5 presents additional, specialized experiments that evaluate some overheads of RCL and briefly discusses an energy-aware version of RCL. Finally, Section 5.6 compares RCL with additional lock algorithms.

5.1. Analysis of the Machines Used in the Evaluation

We perform a comparative analysis of the two machines used in the evaluation, in order to better understand the impact of their different architectures on the performance of RCL and other lock algorithms.

The first machine used in the evaluation is Magnycours-48, an ×86 machine with four 2.1GHz AMD Opteron 6172 CPUs (the Opteron 6100 series is code named

"Magny-cours," hence the name of the machine). It runs Ubuntu 11.10 (Oneiric Ocelot) with a 3.9.7 Linux kernel, Glibc 2.13, Libnuma 2.0.5, and GCC 4.6.1. Each of the four CPUs comes with 12 cores split across two dies: Magnycours-48 has 48 cores in total, that is, 48 hardware threads, since Opterons do not use hardware multithreading. Each die is a NUMA node, therefore, Magnycours-48 has eight NUMA banks. The interconnect links between the eight dies do not form a complete graph: each die is only connected to the other die on the same CPU and to three remote dies. Therefore, the diameter of the interconnect graph is two: intercore communications (fetching cache lines from remote caches, or NUMA accesses, for instance) require at most two hops.

The second machine used in the evaluation is Niagara2-128, a SPARC machine with two 1.165GHz Sun UltraSPARC T2+ CPUs (code named Niagara 2). It runs Solaris 10 (SunOS 5.10) with GCC 4.7.1. Each CPU comes with eight cores on a single die, and each core runs eight hardware threads thanks to fine-grained multithreading [Shah et al. 2007]: Niagara2-128 has 128 hardware threads in total. The main memory is interleaved (NUMA is disabled) and the two CPUs are connected with a single interconnect link.

Cache Access Latencies. We used the Memal benchmark [Boyd-Wickizer et al. 2008] to measure cache access latencies on Magnycours-48 and Niagara2-128. Figure 3(a) summarizes the results, with latencies converted from cycles (c) to nanoseconds (ns) to ease comparison between the two machines. On Niagara2-128, hardware threads access data from the local core up to 25.8 times slower than on Magnycours-48, and they access data from a different core on the local die up to 2.9 times slower than on Magnycours-48. However, Niagara2-128 can almost be twice as fast as Magnycours-48 when it comes to accessing data that is located on a remote die because it has a faster interconnect and its two CPUs are directly connected (at most one hop is needed). These results show that Magnycours-48 is slower when it comes to uncontended interdie communication, but Niagara2-128 has slower uncontended communication inside its dies. Since Magnycours-48 has eight dies with six hardware threads on each, instead of only two dies with 64 hardware threads on each for Niagara2-128, Magnycours-48 uses more interdie communication, which is its weak point, and Niagara2-128 uses more communication that is local to its dies, which is also its weak point. To conclude, it is difficult to determine which of the two machines has the best performance when it comes to cache access latencies.

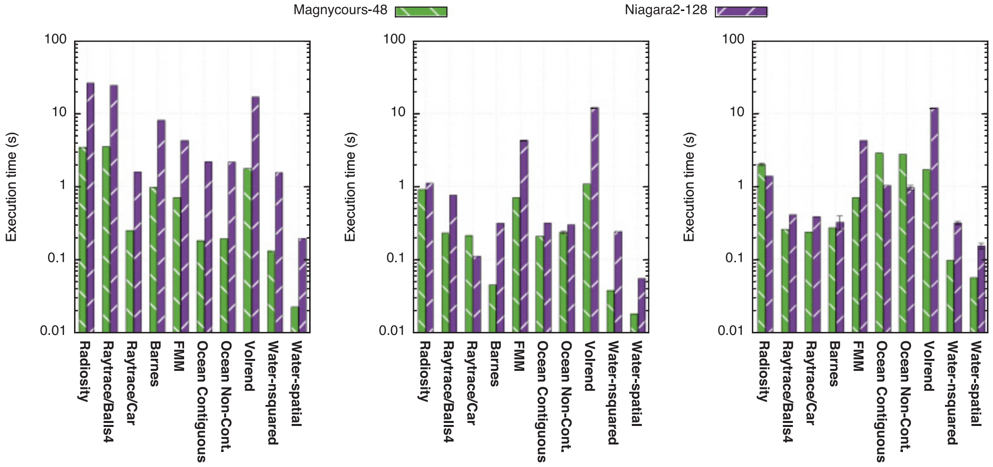
Contention Overhead. We wrote a custom microbenchmark to assess the overhead of contention on regular and atomic instructions on Magnycours-48 and Niagara2-128. This benchmark runs a *monitored thread* that executes an instruction 1,000,000 times on a shared variable, and measures the execution time of every 1,000th instruction: not all of the instructions are monitored in order to prevent the performance measurements from causing too much overhead. The measurements are then averaged in order to produce an estimate of the execution time of the monitored instruction, which can either be an assignment (store) or an atomic Compare-And-Swap (CAS) instruction. Concurrently, the benchmark runs *nonmonitored threads* that repeatedly write random values to the shared variable used by the monitored instruction, in order to simulate contention. The more hardware threads are added, the more contended the shared variable becomes. All threads (monitored or nonmonitored) are bound to separate hardware threads. Threads are bound in such a way that they are first spread on the first hardware thread of each core of the first die, then CPU, and so on until they are bound to the first hardware thread of every core in the machine. After this, the same process continues with the second hardware thread of each core, until all hardware threads are used. For Magnycours-48, since each core runs a single hardware thread, this amounts to a compact binding policy. For Niagara2-128, however, the

	Local core access	Local die access	Remote die access
Magnycours-128	L1: 3 c / 1.4 ns L2: 15 c / 7.1 ns L3: 30 c / 14.3 ns	38c / 18.1 ns	One hop: 220 c / 104.1 ns Two hops: 300 c / 142.9 ns
Niagara2-128	L1 / L2: 42 c / 36.1 ns	CPU 1: 46 c / 39.5 ns CPU 2: 60 c / 52.5 ns	90 c / 77.3 ns

(a) Cache access latencies

		1 thread	2 threads	24 threads	48 threads	64 threads	128 threads
Magnycours-128	**Store**	73 c / 63 ns	183 c / 158 ns	3,032 c / 2,603 ns	6,610 c / 5,674 ns		
Local NUMA bank	**CAS**	98 c / 84 ns	182 c / 157 ns	947 c / 813 ns	5,561 c / 4,774 ns		
Magnycours-128	**Store**	73 c / 63 ns	220 c / 190 ns	4,112 c / 3,530 ns	9,206 c / 7,903 ns		
Remote NUMA bank	**CAS**	98 c / 84 ns	341 c / 293 ns	5,656 c / 4,856 ns	11,749 c / 10,085 ns		
Niagara2-128	**Store**	56 c / 48 ns	56 c / 48 ns	1,623 c / 1,393 ns	4,768 c / 4,093 ns	6,444 c / 5,532 ns	14,752 c / 12,663 ns
	CAS	75 c / 64 ns	75 c / 64 ns	1,633 c / 1,402 ns	4,749 c / 4,076 ns	6,353 c / 5,454 ns	14,860 c / 12,756 ns

(b) Overhead of contention on store and CAS instructions



Single-core 48 threads 128 threads

(c) SPLASH-2 Results

Five runs per data point, error bars show standard deviation when relative standard deviation is > 5%.

Fig. 3. Cost of cache accesses and instructions.

binding policy populates all cores with a single thread at first, in order to see the effects of write-access contention on the cache and memory hierarchy independently from the contention between write operations of different hardware threads on the same core. On Magnycours-48, the shared variable can either be allocated on the NUMA bank of the monitored hardware thread, or on a remote NUMA bank.

Figure 3(b) summarizes the results of the experiment. On Magnycours-48, when only one thread (the monitored thread) is used, the cost of a store (CAS, respectively) instruction is 63.4ns (84.1ns, respectively). In this case, the shared variable is always stored in the L1 cache. When two threads are used, they are located on two cores of the same die, and the nonmonitored thread often brings the shared variable to its local L1 cache, invalidating it from the monitored thread's L1 and L2 caches. However, even though the difference in latency between accessing a variable that resides in a local L1 cache and a remote L1 cache is only 16.7ns (see Figure 3(a)), using two threads instead of one increases the costs of store and CAS instructions by at least 94.3ns. Therefore, the overhead of adding another thread is not only caused by the additional cache misses: the synchronization mechanisms from the cache-coherence protocol induce an overhead when two threads try to write to the same cache line concurrently. Adding more threads keeps increasing the cost of store and CAS instructions to several thousand nanoseconds: using a store (CAS, respectively) instruction under high contention is up to 124.6 (107.2, respectively) times costlier than executing it locally under low contention. Accessing a shared variable that belongs to a local NUMA bank has a lower latency than accessing data that belongs to a remote NUMA bank (−28.1% for store instructions, −52.7% for CAS instructions), even though the shared variable is usually directly transferred between caches during the benchmark and no RAM access is needed. This is due to the HT (HyperTransport) Assist technology: when the monitored thread repeatedly accesses a variable that was allocated on a remote NUMA bank, all messages have to transit by the die whose memory controller is connected to that NUMA node, because the L3 cache of that die contains the NUMA node's cache directory. On the contrary, when the monitored thread accesses data that belongs to its own NUMA node, this indirection is not needed, because the monitored thread can directly access the corresponding cache directory on its local die. Finally, store and CAS instructions have similar costs on Magnycours-48. CAS instructions are more expensive than store instructions under low contention (+55.0% with 24 threads) and under high contention when accessing data from a remote NUMA bank (+27.6% with 48 threads), however, they scale better than store instructions when they are performed on a local NUMA bank (−15.9% with 48 threads).

On Niagara2-128, executing store and CAS instructions is faster than on Magnycours-48 under low contention (24.1% faster for store instructions and 23.5% faster for CAS instructions with one thread). Moreover, Niagara2-128 also scales better than Magnycours-48 on this benchmark: Niagara2-128 is up to 48.2% faster for store instructions and 59.6% faster for CAS instructions than Magnycours-48 when using 48 hardware threads. Increasing the number of threads beyond 48 threads keeps increasing the overhead of store and CAS instructions. In fact, the overhead increases linearly with the number of threads from 24 threads onwards, and with 128 threads, executing a single store (CAS, respectively) instruction takes 263.8 (198.7, respectively) times longer than executing it locally under low contention. Finally, even though CAS instructions are 34.0% slower than store instructions under low contention, they perform similarly under moderate to high contention.

To conclude, Niagara2-128 is able to perform more write accesses to a cache line than Magnycours-48: its architecture resists better when contention is high for concurrent accesses to a cache line.

Sequential vs. Parallel Performance. The second version of the Stanford ParalleL Applications for SHared memory (SPLASH-2) is a benchmark suite that consists of parallel scientific applications for cache-coherent architectures [University of Delaware 2007; Singh et al. 1992; Woo et al. 1995]. We run the applications from the SPLASH-2 suite on both Magnycours-48 and Niagara2-128, for 1, 48, and 128 threads; the results are shown in Figure 3(c). In the single-threaded version, Magnycours-48 clearly outperforms Niagara2-128, with performance improvements ranging between 6.1 times and 12.0 times. On average, Magnycours-48 is 8.9 times faster than Niagara2-128. With 48 threads, Magnycours-48 still outperforms Niagara2-128 most of the time, however, the performance gap is reduced, and on one benchmark (Raytrace/Car), Niagara2-128 manages to outperform Magnycours-48. Niagara2-128 is able to run 64 threads on the same die and can therefore benefit from faster communications than Magnycours-48 (no need to go through the interconnect), which helps compensate for its very low sequential performance. Niagara2-128 outperforms Magnycours-48 on Raytrace/Car because this benchmark spends most of its time performing synchronization (lock acquisitions) instead of sequential computations, as will be seen later in this section. When we use 128 threads, Magnycours-48 is still much faster than Niagara2-128 for most benchmarks, even though it only has 48 hardware threads. Niagara2-128 manages, however, to outperform Magnycours-48 on three benchmarks (Radiosity, Ocean Contiguous, and Ocean Noncontiguous) thanks to its larger amount of hardware threads.

In conclusion, Niagara2-128 has much worse sequential performance than Magnycours-48. Its faster communication between hardware threads and larger number of hardware threads make it possible to reduce the performance gap with Magnycours-48 when a lot of threads are used. Still, it is clearly outperformed by Magnycours-48 on most applications of the SPLASH-2 suite.[10]

Summary. Magnycours-48 is a machine that has much faster sequential performance and faster intradie communication, while Niagara2-128 is a slower machine that sometimes exhibits better performance when it comes to communication between hardware threads, especially under high contention. However, the faster communication and larger amount of hardware threads of Niagara2-128 is not sufficient to make it perform better than Magnycours-48 on a parallel benchmark suite. Since Magnycours-48 has better sequential performance relative to its communication performance than Niagara2-128, it can be expected that synchronization will be more of a bottleneck on Magnycours-48 than on Niagara2-128. The rest of this section will confirm this intuition.

5.2. Liblock

In this evaluation, we compare the performance of RCL with that of other lock algorithms on a microbenchmark and on other applications. For clarity, in all sections other than 5.6, we only compare the performance of RCL with that of a subset of the lock algorithms that were mentioned in this article. This subset of lock algorithms is the following: the basic spinlock, a blocking lock (we use the POSIX implementation provided by Linux or Solaris and simply refer to it as the “POSIX lock,” for short, in this evaluation), MCS, MCS-TP, Flat Combining, CC-Synch, and DSM-Synch, all of which were presented in Section 2. These lock algorithms are (i) a very basic lock algorithm (the basic spinlock), (ii) a very common lock algorithm (a blocking lock), (iii) queue locks (MCS and MCS-TP), and (iv) combining locks (Flat Combining, CC-Synch, and DSM-Synch). Two of these algorithms (CC-Synch and DSM-Synch) are state-of-the-art lock algorithms that were designed at the same time as RCL. CLH is not part of the chosen lock algorithms since it is similar to MCS. Oyama was also not chosen, because other combining locks use the same basic idea as Oyama, with clear improvements, as also explained in Section 2.2. Since MCS-TP is a lock algorithm that only aims to improve the performance of MCS when threads often get preempted, it is only evaluated in experiments that sometimes use more application threads simultaneously than there are hardware threads on the machine, that is, more threads use the contended locks than there are hardware threads on the machine, since in our experiments, all contended locks are used by all application threads. Finally, hierarchical locks (including Cohort Locks) are not part of the subset of chosen lock algorithms because they focus on improving performance on architectures that have a strong cache and memory hierarchy, which is not the main focus of RCL: a hierarchical version of RCL could be designed, possibly using the Lock Cohorting [Dice et al. 2012] technique described in

[10] Interestingly, it can be seen in Figure 3(c) that adding more threads when executing the SPLASH-2 applications often worsens performance instead of improving it: the SPLASH-2 suite was released in the 1990s, when multi-CPU systems only featured a few CPUs with low sequential performance. Such legacy applications are often unable to scale on newer systems, and increasing the number of threads increases contention on shared resources such as locks and more generally, cache lines, which decreases performance.

Section 2.4. Most of the algorithms that were left out will still be evaluated against RCL on a microbenchmark in Section 5.6.

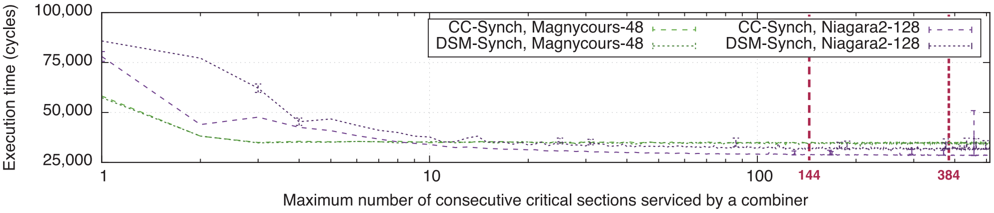
In order to ease the comparison of lock algorithms, we wrote a library named Liblock. This library makes it possible to easily switch between lock implementations in an application. To use Liblock, the application must use critical sections that are encapsulated into functions, since Liblock supports combining locks and RCL. Critical sections are already encapsulated into functions in our microbenchmark (see Section 5.3), and the legacy applications used in Section 5.4 are all transformed using our reengineering tool described in Section 4.2. Using transformed applications instead of replacing the functions that acquire and release the locks has negligible performance impact for noncombining locks.

Liblock's extensible design makes it possible for developers to make it support any lock algorithm. It comes with an implementation of each of the lock algorithms used in this evaluation. In the next paragraphs, we give more information on these implementations.

Lock Implementations in the Liblock. For other lock algorithms than RCL, we use official implementations when they are available. Special attention is given to preserve memory barriers, prefetcher hints, and on Solaris, hints to prevent threads from being preempted (`schedctl_start()`/`schedctl_stop()` function calls); we occasionally add some of these elements to the official implementations when we find they improve performance. RCL aligns its mailbox structure on cache lines and allocates server data structures on the server's NUMA bank. In order to be as fair as possible, the nodes in the lists of the queue locks (MCS and MCS-TP) and combining locks (Flat Combining, CC-Synch, and DSM-Synch) are cache aligned and allocated on their thread's local NUMA bank (threads are bound in most experiments, as explained in Section 5.4.2). Recommended parameter values are used when available. Moreover, we partially explore the parameter space when needed in order to ensure that lock algorithms use satisfactory values for their parameters.

The implementation of MCS is straightforward and does not use any parameters. For MCS-TP, we use the implementation provided by the authors [He et al. 2005b]. MCS-TP uses three parameters: (i) an upper bound on the length of critical sections, for which we choose a value of 10ms: this value is chosen by measuring the length of the critical sections of all applications used in the evaluation and by picking the lowest possible upper bound with a safety margin in order to account for variance; (ii) the approximate length of time it takes a thread to see a timestamp published on another thread: we choose a value of 10μs since this value is sufficient to prevent all deadlocks; and (iii) the maximum amount of time spent waiting in the queue: we choose a value of 50μs, which is the value used in the original paper proposing MCS-TP [He et al. 2005a]. Exploring the parameter space locally shows that these values constitute a local optimum. For Flat Combining, we use the original authors' code [Hendler et al. 2010b], as well as the parameter values they use. The cleanup frequency is set to 100, and the cleanup threshold is set to 10. Again, local exploration of the parameter space shows that Flat Combining performs well with these parameter values.

CC-Synch and DSM-Synch use a single parameter, `MAX_COMBINER_CS`, which specifies the maximum number of critical sections a combiner services before handing over the role of combiner to another thread (see Section 2.3.3). The original paper [Fatourou and Kallimanis 2012] recommends using a value of $n \times h$ with n being a small integer. The implementation proposed by the authors [Fatourou and Kallimanis 2011] uses a value of $n = 3$. It corresponds to a value of `MAX_COMBINER_CS` $= 3 \times h_{m48} = 144$ for Magnycours-48, and `MAX_COMBINER_CS` $= 3 \times h_{n128} = 384$ for Niagara2-128. Figure 4 shows the results of the microbenchmark at highest contention (delay of 100) when



Five runs per data point, error bars show standard deviation when relative standard deviation is > 5%.

Fig. 4. Influence of MAX_COMBINER_CS on CC-Synch and DSM-Synch.

MAX_COMBINER_CS varies between 1 and 500. Both 144 and 384 are located in an area of the graph where the latency of CC-Synch and DSM-Synch is minimal; therefore, these values are used for MAX_COMBINER_CS.

On Magnycours-48, the Liblock uses the optimized RCL runtime described in Section 3.2. On Niagara2-128, however, we were not granted proc_priocntl privileges and therefore, on that machine, the Liblock uses the degraded version of the RCL runtime that was described in the same section.

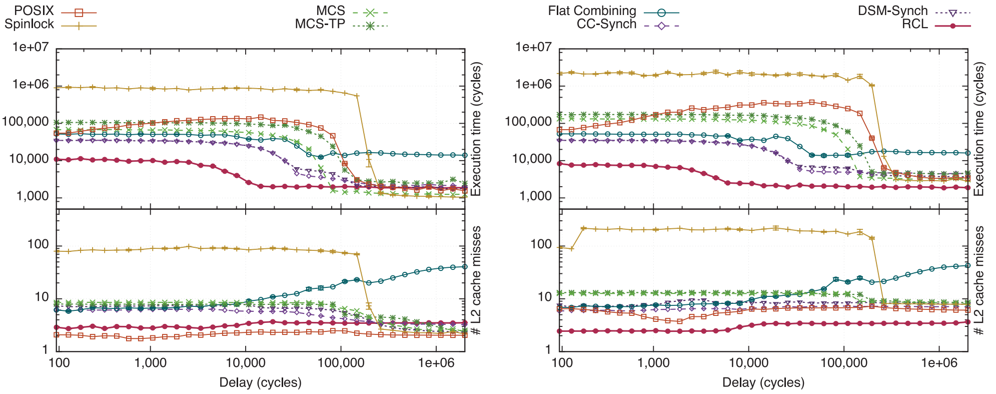
5.3. Microbenchmark

We wrote a microbenchmark in order to measure the performance of RCL relative to other lock algorithms. For other locks than RCL, the microbenchmark executes critical sections repeatedly on all h hardware threads[11] (one software thread per hardware thread, bound), except one that manages the lifecycle of the threads. In the case of RCL, critical sections are executed on only $h - 2$ hardware threads, since one hardware thread manages the lifecycle of threads and another one is dedicated to a RCL server. Only one RCL server is needed, because all critical sections are protected by the same lock. The microbenchmark varies the *degree of contention* on the lock by varying the delay between the execution of the critical sections: the shorter the delay, the higher the contention. The microbenchmark also varies the *locality* of critical sections by making them read and write either one or five cache lines. To ensure that cache lines are not prefetched, they are not contiguous (Magnycours-48's prefetcher always fetches two cache lines at a time), and the address of the next memory access is built from the previously read value [Yotov et al. 2005] when accessing cache lines. This technique is similar to the one used by our benchmark that was used to fine-tune the second version of the profiler, as described in Section 4.1.

Magnycours-48. Figure 5(a) presents the average execution time of a critical section (top) and the number of L2 cache misses (bottom) when each thread executes 10,000 critical sections that each access one shared cache line on Magnycours-48. This experiment mainly measures the effect of lock access contention. Figure 5(b) presents the increase in execution time when each critical section accesses five cache lines instead. This experiment focuses more on the effect of data locality of shared cache lines. In practice, as will be seen in Figure 8, in the evaluated applications, most critical sections trigger between one and five cache misses; therefore, performing either one or five cache line accesses is realistic. Highlights of the experiment are summarized in Figure 5(c).

On Magnycours-48, under high contention (left part of the graph), with five cache line accesses, RCL is several times faster than all other lock algorithms. CC-Synch and DSM-Synch are ~323% slower than RCL, even though they are state-of-the-art

[11] In this section, the total number of hardware threads of a machine is always noted h. The total number of hardware threads of Magnycours-48 (Niagara2-128, respectively) is noted h_{m48} (=48) (h_{n128} (=128), respectively).



(a) One shared cache line per CS (b) Five shared cache lines per CS

Five runs per data point, error bars show standard deviation when relative standard deviation is > 5%.

	High contention (10^2 cycles)		Low contention (2.10^6 cycles)	
	Exec. time (cycles)	# L2 cache misses	Exec. time (cycles)	# L2 cache misses
Spinlock	2,186,419	94.0	2,822	7.7
POSIX	67,740	6.5	3,321	6.1
MCS	132,448	12.7	3,086	8.1
MCS-TP	171,854	13.1	4,332	8.4
Flat Combining	52,358	6.9	16,017	44.2
CC-Synch	35,194	5.9	4,359	7.1
DSM-Synch	35,067	7.6	3,588	7.8
RCL	*8,298*	*2.4 (client) + 0.0 (server)*	*1,899*	*2.4 (client) + 1.2 (server)*

(c) Comparison of the lock algorithms for five shared cache lines

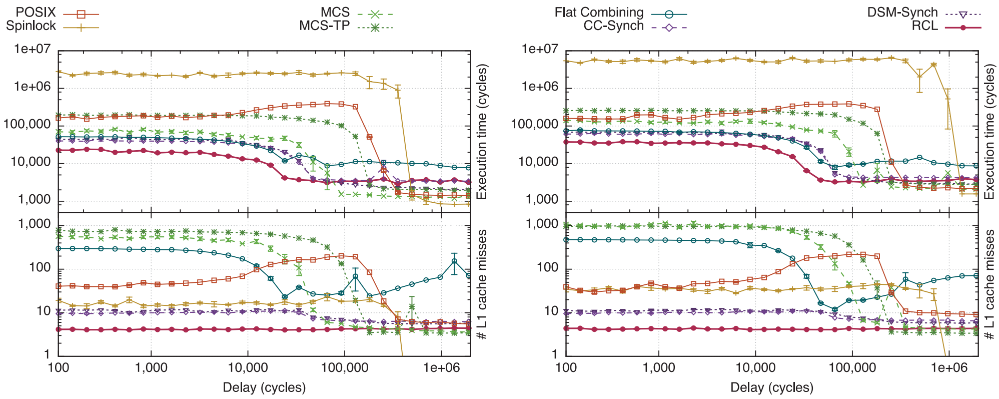
Fig. 5. Microbenchmark results on Magnycours-48.

algorithms that were published shortly before RCL. Both algorithms have comparable performance, as expected on a cache-coherent architecture (see Section 2.3.3). Flat Combining, on which these algorithms are based, performs 49% slower than they do: the removal of the global lock of Flat Combining for handing over the role of combiner is a very effective optimization. MCS is much slower than combining locks: it is ∼277% slower than CC-Synch and DSM-Synch, and 153% slower than Flat Combining. This overhead comes from the fact that with MCS and noncombining locks, synchronization between two threads is necessary between the execution of two critical sections, whereas combining locks execute dozens of critical sections consecutively without any synchronization, as long as the list of requests is full. Let $h_{m48} = 48$ be the number of hardware threads provided by Magnycours-48. Since, under high contention, all $h_{m48} - 1 = 47$ threads are waiting for the execution of a critical section, on average, threads have to wait for the execution of $h_{m48} - 2 = 46$ other critical sections before they can execute theirs. Therefore, the synchronization overhead between two threads in the list is paid 46 times when waiting for the execution of each critical section. MCS-TP makes MCS more resilient to preemption, as shown later in this section; however, this optimization comes at a cost: MCS-TP has an overhead of 30% relative to MCS. The performance of the POSIX lock is reasonably good at very high contention (between that of Flat Combining and MCS), but its performance decreases when contention is average, becoming worse than that of MCS and MCS-TP. This comes from the fact that the POSIX lock directly tries to acquire the lock using the global lock variable before blocking: when contention is very high, that is, the delay between two critical sections is very low, a thread that just released the lock is able to reacquire it immediately if doing so is faster than waking up the next blocked thread in the list that is waiting for the critical section (which is bad for fairness). When contention is less high, a context switch is needed every time the lock is handed over from one thread to the next, which slows down every lock acquisition (but improves fairness). Again, this overhead is paid 46 times since a thread typically has to wait for the execution of 46 other critical sections before it executes its own. Finally, the basic spinlock's performance under high contention is very poor because its repeated use of atomic compare-and-swap instructions saturates the processor interconnect with messages from the cache-coherence protocol.

The performance of lock algorithms at low contention does not matter as much as the performance under high contention, because when contention is low, locks are not a bottleneck. On Magnycours-48, most lock algorithms have comparable performance at low contention (right part of the graph). Flat Combining performs worse than other lock algorithms at low contention because after the execution of each critical section, the combiner thread cleans up the global list. To do so, the combiner thread accesses all remote nodes in the list, which increases the number of cache misses (44.2) and decreases performance. RCL is as efficient as other lock algorithms under low contention when the microbenchmark only accesses one shared cache line, but when it accesses five cache lines, RCL becomes more efficient than the others, because the additional cache line accesses do not incur an overhead in the case of RCL: all accessed variables remain in the cache of the server hardware thread. This is not the case with combining locks because combiners only execute multiple consecutive requests under high contention: when contention is low, a thread that needs to execute a critical section becomes the combiner, executes its own critical section, sees that no other thread has added a request for the execution of a critical section, and goes back to executing its client code. This effect is visible on the bottom part of Figures 5(a) and 5(b): at low contention, while the number of cache misses is higher for most lock algorithms when five cache lines are accessed instead of one, it remains almost constant with RCL. Finally, Figure 5(c) shows that most cache misses incurred by RCL are on the client side, and not on the server side: they are therefore outside the critical path of the server and do not slow down the execution of critical sections for all clients.

Niagara2-128. Figure 6 shows the microbenchmark results on Niagara2-128. Instead of L2 cache misses, L1 cache misses are measured on Niagara2-128, since this machine has one L2 cache per CPU and one L1 cache per core: measuring L2 caches would only measure communication between the first sixty-four hardware threads and the last sixty-four hardware threads, while L1 cache misses are a much better measure of inter-core communication. Also note that, since Niagara2-128 provides $h_{n128} = 128$ hardware threads, $h_{n128} - 1 = 127$ threads execute critical sections are used instead of $h_{48} - 1 = 47$ on Magnycours-48. Therefore, the results of the two experiments are not directly comparable.

The microbenchmark results are qualitatively similar on Niagara2-128 and Magnycours-48. However, the performance gap is lower due to the fact that Niagara2-128 has better communication performance relative to its sequential performance, as explained in Section 5.1: on Niagara2-128, synchronization is less of a bottleneck, and thus, using efficient locks does not improve performance as much as on Magnycours-48. Under high contention, CC-Synch and DSM-Synch are still 62% and 82% slower than RCL, respectively. Flat Combining is 22% slower than CC-Synch and 8% slower than DSM-Synch. Removing the global lock is not as effective in improving performance as it is on Magnycours-48, because, as seen in Section 5.1, the compare-and-swap instruction scales better on Niagara2-128 than on Magnycours-48. MCS is between 55% and 132% slower than combining locks, and the basic spinlock performs even worse than on Magnycours-48, because more hardware threads are used, which increases contention.



(a) One shared cache line per CS

(b) Five shared cache lines per CS

Five runs per data point, error bars show standard deviation when relative standard deviation is > 5%.

	High contention (10^2 cycles)		Low contention (2.10^6 cycles)	
	Exec. time (cycles)	**#L2 cache misses**	**Exec. time (cycles)**	**#L2 cache misses**
Spinlock	5,423,513	34.9	1,550	0.2
POSIX	159,573	39.3	2,110	8.3
MCS	140,557	1063.6	1,857	4.0
MCS-TP	253,772	967.6	2,864	3.8
Flat Combining	74,063	473.9	8,251	72.9
CC-Synch	60,920	9.6	4,201	7.3
DSM-Synch	68,376	11.3	3,788	8.6
RCL	*37,516*	*4.3 (client) + 0.0 (server)*	*4,339*	*5.4 (client) + 0.4 (server)*

(c) Comparison of the lock algorithms for five shared cache lines

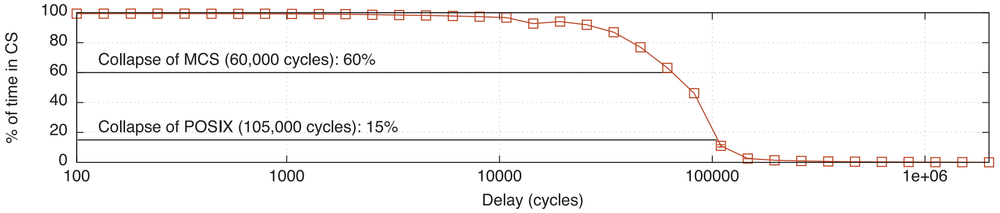
Fig. 6. Microbenchmark results on Niagara2-128.

Highlights. On our microbenchmark, RCL is the most efficient algorithm under high contention, followed by combining locks, then traditional lock algorithms. RCL is not always the fastest algorithm under low contention. Accessing five shared cache lines instead of one in critical sections has more overhead for other lock algorithms than RCL, since with RCL, shared cache lines remain in the cache memory of the server hardware thread.

5.4. Applications

We now turn to the performance of RCL in multithreaded applications. We first present a methodology designed to help developers decide whether RCL may be beneficial for their application using the custom profiler that we presented in Section 4.1. We then evaluate the profiler on applications from the SPLASH-2 and Phoenix 2 benchmark suites, as well as Memcached and Berkeley DB with a TPC-C client. Next, we compare the performance of RCL and other lock algorithms on the subset of applications and datasets that were selected thanks to the profiler data. Finally, we give more detailed results for the experiments from SPLASH-2/Phoenix 2, Memcached, and Berkeley DB, respectively.

5.4.1. Profiling. Our microbenchmark makes it possible to determine at which level of contention lock algorithms collapse, but the metric it uses for measuring contention is the delay between the execution of critical sections: this metric is not a very good measure of contention in real-world applications because contention is also highly dependent on other factors such as the length of critical sections. Instead, our profiler presented in Section 4.1 measures the percentage of execution time spent in critical



Five runs per data point, error bars show standard deviation when relative standard deviation is > 5%.

(a) Estimating thresholds on Magnycours-48

		Upper threshold	Lower threshold
All hardware threads (48/128)	**Magnycours-48**	60,000c / 60%	105,000c / 15%
	Niagara2-128	60,000c / 85%	240,000c / 15%
Half (22 / 62), for Memcached	**Magnycours-48**	25,000c / 55%	30,000c / 45%
	Niagara2-128	8,000c / 75%	40,000c / 45%

(b) All thresholds

Fig. 7. Time spent in critical sections in the microbenchmark and thresholds.

sections, including lock acquisition and release time. This metric was found to be simple to measure, and yet it is a good estimate of lock contention, because when lock contention is high enough that locks are a bottleneck, lock acquisition and release operations become very long and end up taking up most of the execution time. In order to estimate at which point the performance of locks collapse in terms of this metric instead of the delay, we run our microbenchmark through our profiler.

Figure 7(a) shows the result of applying our profiler to our microbenchmark with POSIX locks[12] and one cache line access, on Magnycours-48: the percentage of time spent in critical sections is plotted as a function of the delay. As seen in Figure 5(a), on Magnycours-48, the POSIX lock collapses when the delay is under 105,000 cycles, and RCL is better than all other lock algorithms when the delay is under 60,000 cycles. As shown in Figure 7(a), running the microbenchmark through the profiler makes it possible to deduce that these two values correspond to 15% and 60% of the execution time spent in critical sections, respectively. Therefore, it can be deduced that the POSIX lock collapses when the microbenchmark spends 15% or more of its time in critical sections (lower threshold), and RCL performs better than all other lock algorithms when the microbenchmark spends 60% or more of its time in critical sections (upper threshold). These results are preserved, or improved, as the number of accessed cache lines increases, because the execution time increases at least as much, and usually more, for other algorithms than for RCL, due to RCL's improved data locality.

From our microbenchmark-based analysis, we deduce the following criteria to decide whether RCL can be beneficial to an application that uses POSIX locks on Magnycours-48: if the application spends more than 15% (lower threshold) of its time in critical sections, then using RCL may be beneficial, but not necessarily more than using lock algorithms other than POSIX, and if the application spends more than 60% (upper threshold) of its time in critical sections, using RCL may be more beneficial than using any of the other lock algorithms. A similar analysis on Niagara2-128 shows that an application may benefit from RCL if it spends more than 15% of its time in critical sections (lower threshold), and that it may benefit from RCL more than from any other lock algorithm if it spends more than 85% of its time in critical sections (upper threshold).

[12] We assume that the targeted applications use the default operating system implementation of POSIX locks since it is the most commonly used lock on POSIX systems. A similar analysis could be made for any other lock algorithm.

		% in CS = f(# hardware threads)						Lock info for max. # hardware threads			
		1	4	8	16	32	48	Description	#	#L2 cache misses / CS	% in CS
SPLASH-2	Radiosity	3.7%	6.5%	10.4%	38.4%	76.6%	89.2%	Linked list access	1	1.7	87.7%
	Raytrace/Balls4	0.5%	1.0%	1.6%	3.0%	22.6%	66.8%	Counter increment	1	1.4	65.7%
	Raytrace/Car	9.2%	19.6%	45.8%	70.1%	86.5%	91.7%	Counter increment	1	0.6	90.2%
	Barnes						10.7%				
	FMM						1.0%				
	Ocean Contiguous					0.1%[†]					
	Ocean Non-Cont.					0.1%[†]					
	Volrend						14.0%				
	Water-nsquared						6.8%				
	Water-spatial						3.3%				
Phoenix 2	Linear Regression	0.9%	26.5%	39.4%	68.7%	60.1%	81.6%	Task queue access	1	3.8	81.6%
	String Match	0.2%	5.7%	10.2%	23.0%	37.6%	63.9%	Task queue access	1	4.5	63.9%
	Matrix Multiply	0.9%	27.5%	41.8%	67.3%	79.7%	92.2%	Task queue access	1	3.1	92.2%
	Histogram						13.8%				
	PCA						12.2%				
	KMeans						1.1%				
	Word Count						4.1%				
Memcached	Get	3.2%	20.3%	40.7%	69.7%	22 hw. t.: 79.0%[‡]		Hashtable access	1	2.1	78.3%
	Set	3.7%	19.3%	28.7%	39.1%	22 hw. t.: 44.7%[‡]		Hashtable access	1	16.5	44.7%
Berkeley DB + TPC-C	Payment						10.1%				
	New Order						5.2%				
	Order Status	2.1%	2.5%	2.1%	2.3%	1.1%	41.2%	DB struct. access	11	2.4	40.1%
	Delivery						4.3%				
	Stock Level	2.1%	2.2%	2.4%	0.5%	0.4%	46.3%	DB struct. access	11	2.4	46.3%

[†] Number of hardware threads must be a power of 2. [‡] Other hardware threads are executing clients.

Fig. 8. Profiling results for the evaluated applications on Magnycours-48.

The preceding thresholds have been found by running the microbenchmark on all hardware threads. However, as seen later in this section, Memcached is run with half of the hardware threads dedicated to a client that sends requests to the Memcached instance. Therefore, the same analysis was performed with half the hardware threads in order to find suitable thresholds for this application. All obtained thresholds are listed in Figure 7(b).

Applications. Figure 8 (9, respectively) shows the results of the profiler for 18 applications on Magnycours-48 (Niagara2-128, respectively). The evaluated applications are the following:

—The nine applications of the SPLASH-2 benchmark suite (parallel scientific applications, see Section 5.1). Since Raytrace is provided with two datasets, Balls4 and Car, the results for both are shown (a third data set, Teapot, is also provided but only for debugging purposes).

—The seven applications of the Phoenix 2.0.0 benchmark suite [Stanford University 2011; Talbot et al. 2011; Yoo et al. 2009; Ranger et al. 2007] developed at Stanford. These applications implement common uses of Google's MapReduce [Dean and Ghemawat 2008] programming model in the C programming language. Phoenix 2 provides a small, a medium, and a large dataset for each application. The medium datasets were used for all applications.

—Memcached 1.4.6 [Danga Interactive 2003; Fitzpatrick 2004], a general-purpose distributed memory caching system that is used by websites such as YouTube, Wikipedia, and Reddit. Memcached is run on half of the hardware threads of each machine, because the other half of the hardware threads are dedicated to the client that generates the load. This can be done without impacting performance because Memcached's scalability is too low for it to benefit from more than half the hardware

		% in CS = f(# hardware threads)							Lock info for max. # hardware threads			
		1	4	8	16	32	64	128	Description	#	# L1 cache misses / CS	% in CS
SPLASH-2	Radiosity	1.6%	2.8%	2.8%	3.0%	3.1%	3.1%	38.7%	Linked list access	1	5.4	35.5%
	Raytrace/Balls4	0.3%	0.5%	0.4%	0.5%	0.5%	0.6%	14.3%	Counter increment	1	4.6	13.4%
	Raytrace/Car	5.4%	5.9%	5.8%	6.0%	9.2%	44.7%	79.1%	Counter increment	1	3.8	77.0%
	Barnes							3.7%				
	FMM							0.5%				
	Ocean Cont.							0.0%				
	Ocean Non-Cont.							0.0%				
	Volrend							1.0%				
	Water-nsquared							7.3%				
	Water-spatial							0.1%				
Phoenix 2	Linear Regression							3.4%				
	String Match							2.9%				
	Matrix Multiply							10.2%				
	Histogram							3.2%				
	PCA							0.1%				
	KMeans							0.0%				
	Word Count							2.0%				
Memcached	Get	5.4%	10.4%	16.5%	42.3%	62.2%	62 hw. t.: 69.9%‡		Hashtable access	1	13.5	69.2%
	Set	5.0%	8.7%	14.0%	17.5%	19.3%	62 hw. t.: 20.4%‡		Hashtable access	1	73.4	20.2%
Berkeley DB + TPC-C	Payment							1.8%				
	New Order							0.3%				
	Order Status	0.2%	0.1%	0.1%	0.1%	43.8%	63.6%	76.4%	DB struct. access	11	4.0	76.4%
	Delivery							0.8%				
	Stock Level	0.2%	0.1%	0.1%	0.1%	55.5%	78.2%	87.1%	DB struct. access	11	3.4	87.1%

‡ Other hardware threads are executing clients.

Fig. 9. Profiling results for the evaluated applications on Niagara2-128.

threads of Magnycours-48 or Niagara2-128. As shown in Section 5.4.4, this is true even when Memcached is modified to use other lock algorithms, including RCL.

Memcached uses two threads that perform tasks other than executing requests: one dispatches incoming packets to the worker threads, and another is responsible for performing maintenance operations on the global hashtable that stores cached data, such as resizing it when needed. Therefore, for Memcached to run n worker threads, $n+2$ hardware threads are needed: this explains why the maximum number of hardware threads used in the experiment is $h_{m48}/2 - 2 = 22$ ($h_{n128}/2 - 2 = 62$, respectively) for Magnycours-48 (Niagara2-128, respectively) instead of $h_{m48}/2 = 24$ ($h_{n128}/2 = 64$, respectively). The client used to generate the load is Memslap, from Libmemcached 1.0.2 [Data Differential 2011]. Two experiments are performed: in the first one, the client only executes Get requests (reads from the cache), and in the second one, it only executes Set requests (writes to the cache).

—Berkeley DB 5.2.28 [Oracle Corporation 2004; Olson et al. 1999], a database engine maintained by Oracle, with TpccOverBkDB, a TPC-C [Leutenegger and Dias 1993] benchmark written by Alexandra Fedorova and Justin Fuston at Simon Fraser University. Five experiments are performed, one for each of the request types offered by TPC-C. Berkeley DB takes the form of a library, and the client is an application that creates one thread that uses Berkeley DB routines for each simulated client.

The left part of the tables in Figures 8 and 9 shows the time spent in critical sections for the selected applications, for different numbers of threads. The time spent in critical sections increases when the number of threads increases, because increasing the number of threads increases contention. A gray box in the tables indicates that the application is not profiled for the corresponding number of threads, because even when using one software thread per hardware thread, the time spent in critical sections is very low (under the lower threshold), and therefore, locks are not a bottleneck. The right part of the tables shows the time spent in critical sections and the number of cache misses for the most used locks.

In SPLASH-2/Radiosity, the most used lock is used to concurrently access a linked list. In SPLASH-2/Raytrace, the most used lock protects a shared counter. In the Phoenix benchmarks, the most used lock protect the task queue from the MapReduce implementation. In Memcached, the most used lock protects the shared hashtable that stores all of the cached data.[13] For the contended SPLASH-2 and Phoenix 2 applications, as well as Memcached, the time spent in critical sections for the most used lock is extremely close to the global time spent in critical sections: this shows that their lock bottleneck comes from a single lock. Most benchmarks have a low number of cache misses, except Memcached/Set (and, to some extent, Memcached/Get on Niagara2-128, but 13.5 is not a very high number for L1 cache misses). For Berkeley DB, the profiler identifies a group of eleven locks that are highly contended, because these locks are all allocated at the same code location (same file and line number) in the Berkeley DB library, and the profiler identifies locks by their allocation site. These locks protect the accesses to a structure that is unique for each database, and 11 databases are used in TPC-C. The time spent in critical sections by these 11 locks is equal to the global time spent in critical sections, which shows that no other lock in the application is a bottleneck. The number of cache misses is always low in the experiments with Berkeley DB.

5.4.2. Performance Overview. The two metrics provided by the profiler, that is, the time spent in critical sections and the number of cache misses, do not, of course, completely determine whether an application will benefit from RCL. Many other factors (length of critical sections, interactions between locks, etc.) affect the execution of critical sections. However, as shown in this section, using the time spent in critical sections as the main metric and the number of cache misses in critical sections as a secondary metric works well: the former is a good indicator of contention, and the latter of data locality.

In order to evaluate the performance of RCL, the performance of applications listed in Figures 8 and 9 is measured with the lock algorithms (including RCL) listed in Figures 5(c) and 6(c). The following paragraphs describe the applications that were selected for the evaluation based on the results of the profiler.

SPLASH-2 and Phoenix. For the SPLASH-2 and Phoenix benchmark suites, we present results for the experiments whose time spent in critical sections is higher than one of the thresholds identified in Section 5.3. As explained in Section 5.4.1, since the time spent in critical sections grows with lock contention, when one of the thresholds is reached, using RCL may be beneficial thanks to RCL's good performance under high contention. Since all SPLASH-2 and Phoenix applications always exhibit a low number of cache misses in critical sections (fewer than five) in Figures 8 and 9, data locality is disregarded as a criterion to select experiments. Based on the results shown in Figures 8 and 9, for Magnycours-48, the results of String Match (Phoenix 2), Raytrace/Balls4 (SPLASH-2), Linear Regression (Phoenix 2), Radiosity (SPLASH-2), Raytrace/Car (SPLASH-2), and Matrix Multiply (Phoenix 2) are presented, since these experiments all spend more time in critical sections than the upper threshold. On Magnycours-48, there is no experiment whose time spent in critical sections is between the lower and the upper thresholds. On Niagara2-128, none of the experiments from SPLASH-2 or Phoenix 2 spends more time in critical sections than the upper threshold. On that machine, the results of Radiosity (Phoenix 2) and Raytrace/Car (SPLASH-2) are presented, since the time they spend in critical sections is between the lower

[13] Memcached 1.4.6 was the most recent version available when we were working on the initial RCL paper [Lozi et al. 2012]. Since then, the multithreaded implementation of Memcached has improved, with finer-grained locking being used to protect access to the hashtable.

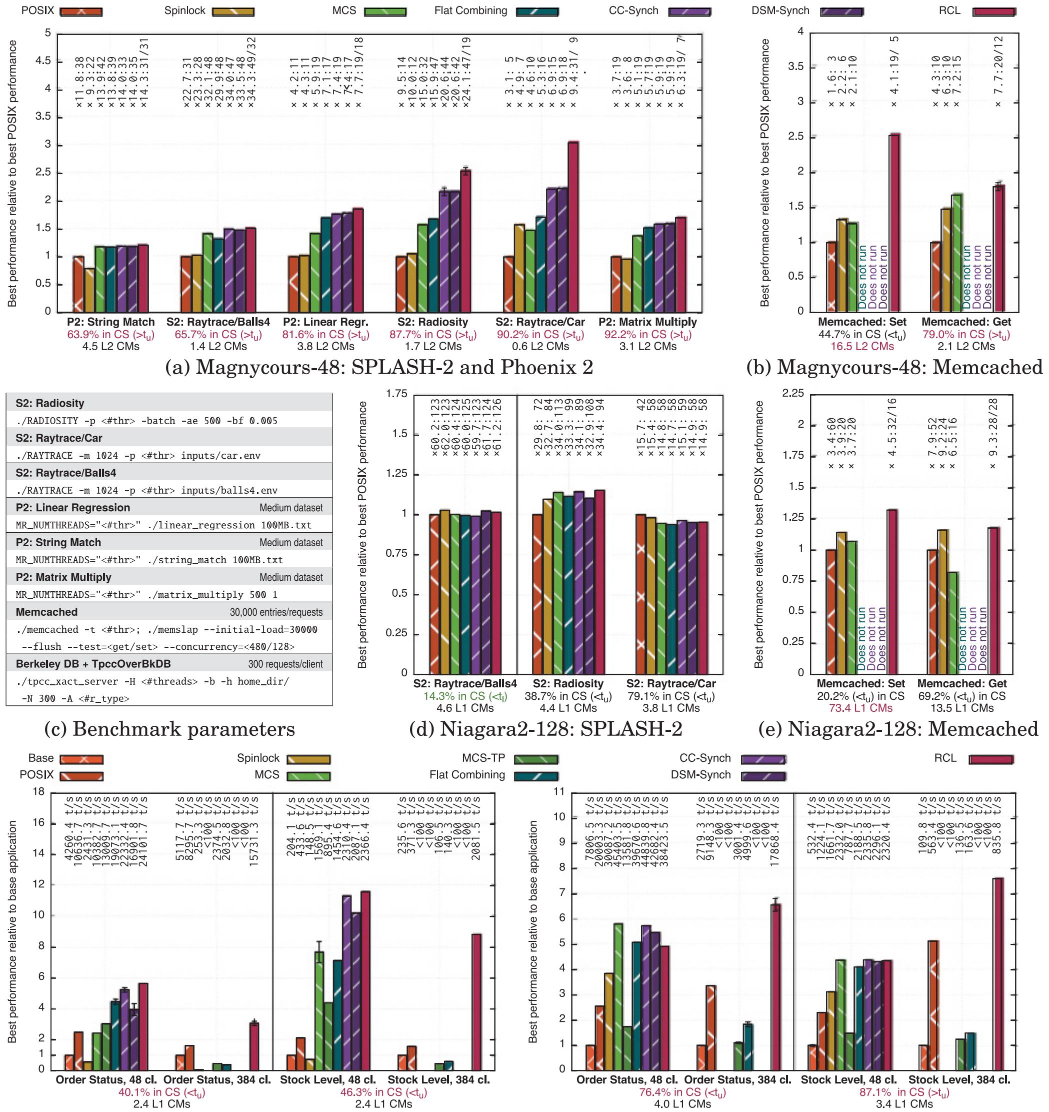
threshold and the upper threshold. Finally, the results of Raytrace/Balls4 are shown even though the time spent in critical sections for this experiment is below the lower threshold: here, we illustrate a case where lock contention is low enough that using more efficient lock algorithms, including RCL, should not improve performance.

Memcached. For Memcached, on Magnycours-48, the experiment with Get requests spends more time in critical sections than the upper threshold. Even though the experiment with Set requests spends an amount of time in critical sections that is between the two thresholds, it also exhibits a large number of cache misses in critical sections (16.5 L2 cache misses). Therefore, a significant performance improvement can be expected in that experiment. Similarly, on Niagara2-128, both experiments (with Get and Set requests) spend an amount of time in critical sections that is between the lower and the upper threshold. On that machine, Memcached/Set exhibits a large number of cache misses in critical sections (72.4 L1 cache misses), which indicates that RCL could be beneficial for locality in this experiment. The performance of combining locks was not evaluated with Memcached, because Memcached periodically blocks on condition variables, and as explained in Section 2, implementing condition variables for combining locks is not trivial. Condition variables were, however, implemented for other lock algorithms, using POSIX primitives.

Berkeley DB with TpccOverBkDb. For Berkeley DB with TpccOverBkDb, we present results for Order Status and Stock Level requests on both machines. On Magnycours-48, the time spent in critical sections for these two experiments, albeit high, is lower than the upper threshold. However, the profiler underestimates the actual time spent in critical sections for Berkeley DB, because Berkeley DB uses hybrid locks that busy-wait for a moment before going to sleep with a POSIX lock: the busy-waiting time is not included in the percentage of time spent in critical sections because the profiler is written for POSIX locks. The performance of Berkeley DB with TpccOverBkDb is not evaluated for the three other types of requests, namely, Payment, New Order, and Delivery, because the time they spend in critical sections is extremely low (lower than both thresholds). On Niagara2-128, the results of the profiler are similar, except for the fact that on that machine, the experiment with Stock Level requests spends more time in critical sections than the upper threshold.

Figure 10 presents an overview of the performance results of all the selected experiments for all lock algorithms. As a reminder, some of the profiling data is shown again in this figure, and the amount of time spent in critical sections is compared to the lower and upper thresholds (noted t_l and t_u) that are suitable for each experiment. The experiments are run with the parameters shown in Figure 10(c). Custom parameters were used for Radiosity (SPLASH-2) in order to make the benchmark perform more work: since SPLASH-2 applications were designed in the 1990s, some of them execute too fast on modern architectures with default parameters to give usable results. As explained in Section 5.4.1, Phoenix 2 uses the medium datasets for its benchmarks, which corresponds to 100MB input files for Linear Regression and String Match, and a 500×500 matrix for Matrix Multiply. For Memcached, the hashtable is preloaded with 30,000 entries when Get requests (reads) are used, whereas it is preloaded with only 10,000 requests for Set requests, since in that experiment, the client fills the hashtable. On Magnycours-48, each hardware thread runs 10 clients (`--concurrency=480`), while on Niagara2-128, each hardware thread runs a single client (`--concurrency=128`). For Berkeley DB with TpccOverBkDb, each client uses a separate thread, and each of these threads executes 300 requests.

In the applications from Figures 10(a), 10(b), 10(d), and 10(e), that is, SPLASH-2 and Phoenix 2 applications as well as Memcached, only one lock is replaced, as indicated in Figures 5(c) and 6(c). Therefore, for RCL, these applications only use one server hardware thread. In the figures of this section, when an application is run with *n*



(a) Magnycours-48: SPLASH-2 and Phoenix 2

(b) Magnycours-48: Memcached

S2: Radiosity	
`./RADIOSITY -p <#thr> -batch -ae 500 -bf 0.005`	
S2: Raytrace/Car	
`./RAYTRACE -m 1024 -p <#thr> inputs/car.env`	
S2: Raytrace/Balls4	
`./RAYTRACE -m 1024 -p <#thr> inputs/balls4.env`	
P2: Linear Regression	Medium dataset
`MR_NUMTHREADS="<#thr>" ./linear_regression 100MB.txt`	
P2: String Match	Medium dataset
`MR_NUMTHREADS="<#thr>" ./string_match 100MB.txt`	
P2: Matrix Multiply	Medium dataset
`MR_NUMTHREADS="<#thr>" ./matrix_multiply 500 1`	
Memcached	30,000 entries/requests
`./memcached -t <#thr>; ./memslap --initial-load=30000 --flush --test=<get/set> --concurrency=<480/128>`	
Berkeley DB + TpccOverBkDB	300 requests/client
`./tpcc_xact_server -H <#threads> -b -h home_dir/ -N 300 -A <#r_type>`	

(c) Benchmark parameters

(d) Niagara2-128: SPLASH-2

(e) Niagara2-128: Memcached

(f) Magnycours-48: Berkeley DB

(g) Niagara2-128: Berkeley DB

Five runs per data point, error bars show standard deviation when relative standard deviation is > 5%. In this figure, t_l and t_u stand for the lower and the upper threshold corresponding to each experiment.

Fig. 10. Application performance overview.

hardware threads, it either means that (i) $n - s$ application threads and s servers are run when RCL is used, or (ii) n application threads are used for other applications. All software threads are bound to hardware threads in all applications, except for Berkeley DB, since in that case, when more software threads can be used than the number of hardware threads, dynamic scheduling can make better use of the hardware threads than static scheduling.

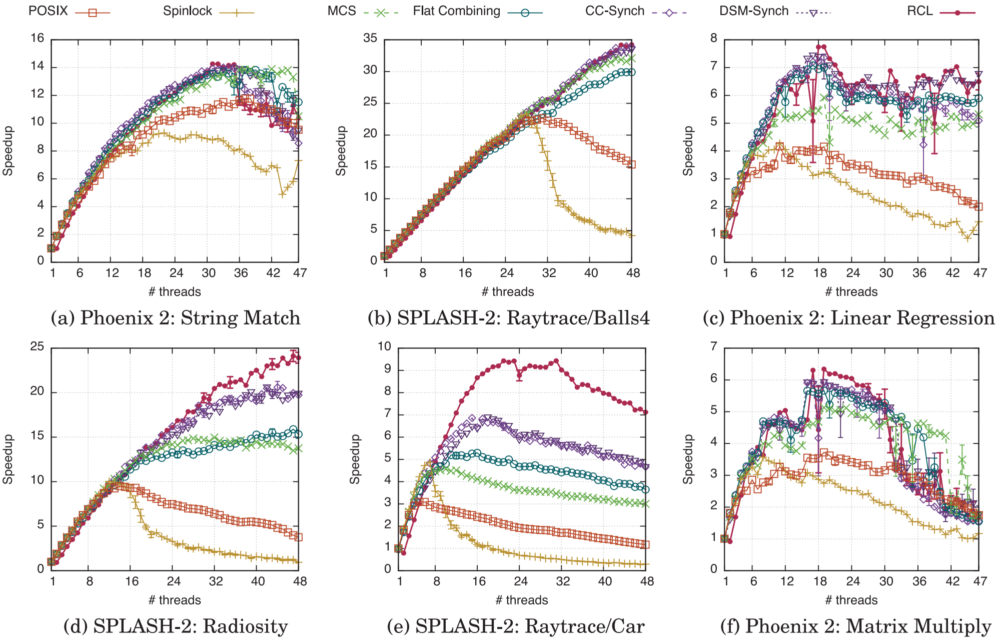
The numbers above the histograms ($\times\alpha : \eta/\mu$) report the improvement α over the execution time of the original application on one hardware thread, the number η of hardware threads that gives the shortest execution time (i.e., the scalability peak), and the minimal number μ of hardware threads for which RCL is faster than all other lock algorithms. The histograms show the ratio of the execution time with each of the lock algorithms relative to the execution time with POSIX locks.

The following general trends can be observed in Figure 10: RCL is generally faster than other lock algorithms, and when the percentage of time spent in critical sections increases or when the number of cache misses increases, the performance improvement offered by RCL also increases. On Magnycours-48, the percentage of time spent in critical sections is generally higher than on Niagara2-128 for applications that use POSIX locks (i.e., all applications other than Berkeley DB). Again, this is due to the fact that on Niagara2-128, as was explained in Section 5.1, the cost of communication is lower relative to its sequential performance. Therefore, synchronization is less of a bottleneck. On Berkeley DB with TpccOverBkDb, the percentage of time spent in critical sections is higher for Niagara2-128, but since Berkeley DB uses non-POSIX locks, measurements of this metric by the profiler are not reliable: the performance gains offered by RCL on Niagara2 for Berkeley DB are lower than on Magnycours-48, which seems to indicate that Berkeley DB with TpccOverBkDb actually suffers from more lock contention on Magnycours-48 than on Niagara2-128. We were able to confirm this intuition with profiling tools (Oprofile and Dtrace): Berkeley DB spends a significantly larger proportion of its execution time in busy-wait loops on Magnycours-48 than on Niagara2-128 (at least 15% more).

5.4.3. Performance Analysis of SPLASH-2 and Phoenix Applications. As shown in Figure 10(a), on Magnycours-48, where all of the selected experiments from SPLASH-2 and Phoenix 2 spend more time in critical sections than the upper threshold, the performance gain for efficient lock algorithms, RCL in particular, increases with the time spent in critical sections: the time spent in critical sections is a good indicator of contention, and consequently, it is a good indicator of how efficient using lock algorithms that resist better to contention will be. However, even though Matrix Multiply (Phoenix 2) spends 92.2% of its time in critical sections when using POSIX locks, its performance improvement with RCL is similar to that of Linear Regression (Phoenix 2) which only spends 81.6% of its time in critical sections. This comes from the fact that Matrix Multiply suffers from another bottleneck that we were able to identify using Oprofile: with the standard "medium" dataset, Matrix Multiply spends a majority of its time creating threads for the Map operations. When many threads are created simultaneously, we observe major contention on the page table lock in the `try_preserve_large_page()` function. This seems to be the same bug that was found with Metis in Boyd-Wickizer et al. [2010].

On average, the basic spinlock performs similarly to the POSIX lock. MCS and Flat Combining improve performance significantly, with Flat Combining being slightly more efficient than MCS most of the time. CC-Synch and DSM-Synch consistently perform better than POSIX, the basic spinlock, MCS, and Flat Combining, and they both provide similar results as expected on a cache-coherent machine. RCL performs significantly better than all other lock algorithms, and the performance gain increases as the time in critical sections increases, except for Matrix Multiply. The performance of lock algorithms in these applications is consistent with the results from the microbenchmark, except for the basic spinlock, which sometimes performs better than the POSIX lock even though it always performs much worse in the microbenchmark.

Detailed Results on Magnycours-48. Figure 11 shows detailed results for the selected SPLASH-2 and Phoenix 2 experiments on Magnycours-48: for each lock algorithm on each benchmark, the speedup relative to the single-threaded version is plotted as a function of the number of threads used. Since RCL uses one hardware thread for the server, its performance is worse when only a few application threads are used, but it



(a) Phoenix 2: String Match (b) SPLASH-2: Raytrace/Balls4 (c) Phoenix 2: Linear Regression

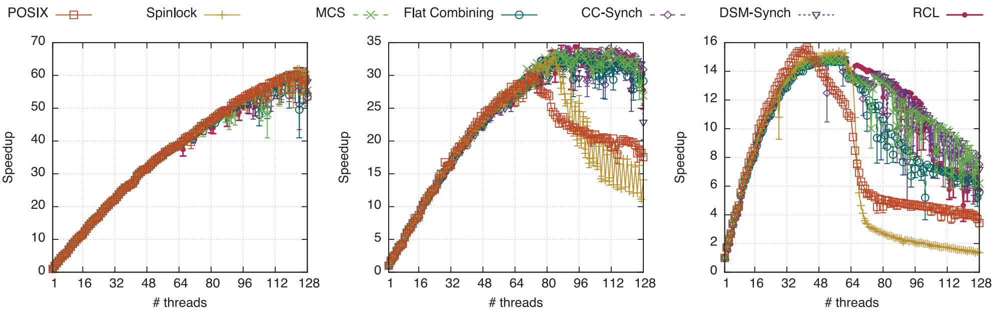
(d) SPLASH-2: Radiosity (e) SPLASH-2: Raytrace/Car (f) Phoenix 2: Matrix Multiply

Five runs per data point, error bars show standard deviation when relative standard deviation is > 5%.

Fig. 11. SPLASH-2 and Phoenix 2 speedup on Magnycours-48.

catches up with other lock algorithms quickly (after 3 to 10 threads). The more time experiments spend in critical sections, the earlier the unmodified version of the application with POSIX locks collapses: String Match (Phoenix 2) spends 63.9% of its execution time in critical sections and starts collapsing at 39 hardware threads, Raytrace/Balls4 (SPLASH-2) spends 65.7% of its execution time in critical sections and starts collapsing at 32 hardware threads, Linear Regression (Phoenix 2) spends 81.6% of its time in critical sections and starts collapsing at 20 hardware threads, Radiosity (SPLASH-2) spends 87.7% of its time in critical sections and starts collapsing at 15 hardware threads, and Raytrace/Car spends 90.2% of its time in critical sections and starts collapsing at eight hardware threads. An early collapse indicates high contention, which shows that the profiler efficiently identifies highly contended locks. The higher the contention, the more using more efficient lock algorithms, RCL in particular, improves performance (better speedup) and scalability (later collapse). Again, the only outlier is Matrix Multiply (Phoenix 2), which spends 92.2% of its time in critical sections and yet starts collapsing for all lock algorithms at around 20 hardware threads, because of the bottleneck described previously, which prevents all lock algorithms from improving performance beyond that point. Even though the basic spinlock usually collapses before the POSIX lock, its performance peak is sometimes higher than the POSIX lock's. This shows that the basic spinlock can exhibit good performance when the number of hardware threads is low because it has not saturated the bus yet, which explains the performance gap of the basic spinlock with the microbenchmark: in Figure 5, the basic spinlock always performs poorly because the maximum number of hardware threads is always used.

Detailed Results on Niagara2-128. As seen in Figure 10(d), on Niagara2-128, the performance of Raytrace/Balls4 (SPLASH-2) does not improve when using more efficient lock algorithms, and Figure 12(a) shows that the performance of Raytrace/Balls4



(a) Raytrace/Balls4 (b) Radiosity (c) Raytrace/Car

Five runs per data point, error bars show standard deviation when relative standard deviation is > 5%.

Fig. 12. SPLASH-2 speedup on Niagara2-128.

never collapses. This is the expected behavior, because Raytrace/Balls4 spends less time in critical sections than both thresholds: the profiler correctly estimates that the experiment does not suffer from a level of lock contention that is high enough for performance to be improved by using better lock algorithms. The fact that replacing POSIX locks by other lock algorithms does not worsen performance indicates that even if a developer mistakenly replaces a POSIX lock by a more efficient one, such as RCL, due to a false positive from the profiler, no negative consequences are to be expected when it comes to performance. Moreover, the profiler does not return any false negatives on the 18 experiments that were profiled: although we do not show these results in this article, we did evaluate the performance of other experiments whose time spent in critical sections was below both thresholds with all lock algorithms, and the result was always the same: changing lock algorithms does not alter performance in these cases.

Note that even though the performance of Raytrace/Balls4 in Figure 12(a) never collapses, the slope of the curve decreases as the number of threads increases. This may indicate that the application simply does not reach the point where adding more threads increases lock contention enough that its performance collapses and using other lock algorithms improves performance: on similar yet newer architectures with more hardware threads, one might expect that the application would reach that point and that using more efficient lock algorithms would improve performance.

Radiosity (SPLASH-2) and Raytrace/Car (SPLASH-2) spend an amount of time in critical sections that is between the two thresholds; therefore, using other lock algorithms than POSIX should improve performance, but RCL may not improve performance more than the other lock algorithms. This effect can be seen for Radiosity in Figure 12(b): any lock algorithm other than POSIX improves performance, but MCS, Flat Combining, CC-Synch, DSM-Synch, and RCL all give similar results. With Raytrace/Car, however, the POSIX lock is more efficient than all other lock algorithms even though according to the profiler, the other lock algorithms should improve performance. As seen in Figure12(c), in that experiment, even though the POSIX lock collapses first, the other lock algorithms do not perform better because the application always collapses when it reaches a speedup of 16 times, which seems to indicate that a bottleneck other than locks prevents the application from scaling beyond that speedup.

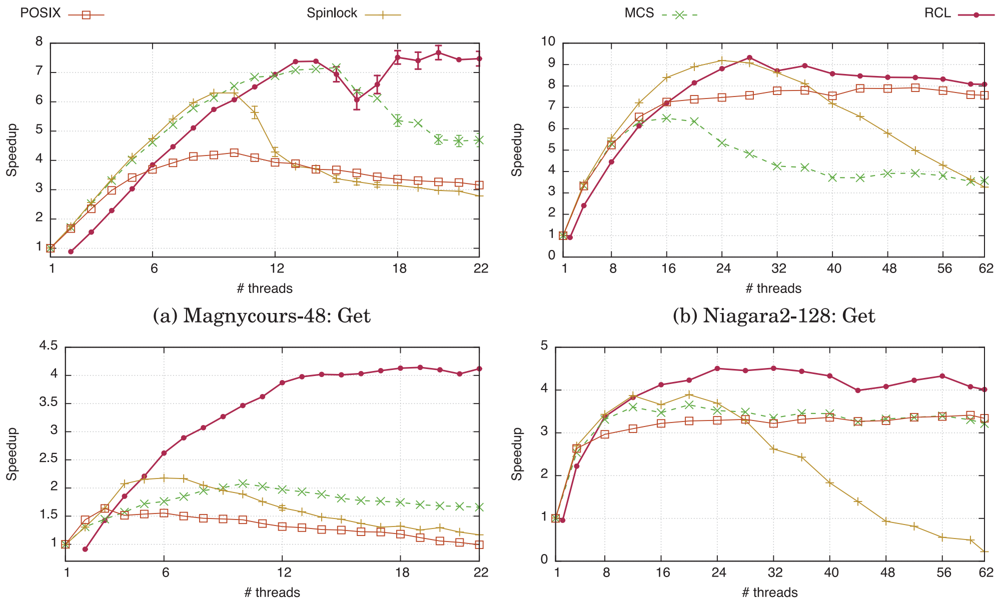
Highlights. On Magnycours-48, in most cases, the more time SPLASH-2 and Phoenix 2 benchmarks spend in critical sections, the more using lock algorithms that perform well under high contention improves performance. Again, RCL is generally the most efficient algorithm, followed by combining locks, then traditional locks. The more efficient a lock algorithm is, the more it improves performance and the later it collapses (improved scalability). On Niagara-2, RCL does not improve performance more than some other lock algorithms, as expected, since our profiler did not detect very high contention on SPLASH-2 and Phoenix 2 benchmarks on this machine.

5.4.4. Performance Analysis of Memcached. As seen in Figure 8, on Magnycours-48, Memcached/Get spends 79% of its time in critical sections and improves performance by 1.80 times, which is consistent with the performance improvements observed on SPLASH-2 and Phoenix in Figure 10(b): Memcached/Get spends more time in critical sections than Raytrace/Balls4 but less than Linear Regression, and in these two experiments, RCL improves performance by 1.51 times and 1.86 times, respectively. For Memcached/Set, RCL drastically improves performance, by 2.53 times, even though it only spends 44.7% of its time in critical sections: this discrepancy is caused by the fact that in that case, RCL not only performs better because lock contention is high, it also increases locality.

To quantify the locality increase, we measure the number of cache misses inside critical sections (i.e., not counting lock acquisition) and on the RCL server after transformation. Our objective here is to quantify the locality increase when the code of critical sections is being executed, since we already evaluated the locality overhead of lock acquisitions in Figures 5(c) and 6(c). The results are shown in Figure 14(a), with $h/2-2$ threads for Memcached. The base Memcached/Set application triggers a lot of cache misses (16.5) which slow down the execution of the critical path, and RCL is able to improve locality, roughly dividing the number of L2 cache misses by 3: with RCL, critical sections only trigger 5.7 cache misses on the server. The contended critical sections in Memcached/Set protect writes to the hashtable. With RCL, large parts of the hashtable remain stored in the cache hierarchy of the server hardware thread, therefore, the number of cache misses drops and performance is vastly improved. However, as seen in Figure 14(a), RCL does not always decrease the number of L2 cache misses significantly in applications whose critical sections only trigger a few (<5) cache misses, because even though it improves locality, RCL also adds cache misses for accessing context variables: a significant amount of cache misses in critical sections seems to be needed for RCL to ensure better locality.

Detailed Results on Magnycours-48. As seen in Figures 13(a) and 13(c), with both Get and Set requests, RCL not only improves performance, it also improves scalability. In Memcached/Get, both the POSIX lock and the basic spinlock start collapsing at around 11 hardware threads, MCS starts collapsing at around 16 hardware threads, while RCL reaches a plateau from 18 hardware threads onwards. RCL is initially slower than other lock algorithms due to the fact that it loses one hardware thread for the server, but it then reaches the performance of the POSIX lock, the basic spinlock, and MCS at 6, 11, and 12 hardware threads, respectively. In Memcached/Set, the POSIX lock, the basic spinlock, and MCS start collapsing at 4, 8, and 11 hardware threads, respectively, while RCL reaches a plateau at around 14 hardware threads. RCL surpasses the performance of all other lock algorithms at only five hardware threads.

Detailed Results on Niagara2-128. As seen in Figure 9, on Niagara2-128, Memcached/Get spends an amount of time in critical sections that is between the lower and the upper threshold, which means that using RCL should improve performance, but not necessarily more than other lock algorithms. This is indeed what can be observed in Figure 10(e): both the basic spinlock and RCL improve performance, with a small advantage for RCL that might not be significant. However, as seen in Figure 13(b), even though RCL and the basic spinlock have similar peak performance, RCL performs better when a large number of hardware threads is used: at 62 hardware threads,



(a) Magnycours-48: Get

(b) Niagara2-128: Get

(c) Magnycours-48: Set

(d) Niagara2-128: Set

Five runs per data point, error bars show standard deviation when relative standard deviation is > 5%.

Fig. 13. Memcached speedup.

the speedup of the application collapses to 3.3 times when the basic spinlock is used, whereas using RCL makes it possible to maintain a speedup of 8.1 times.

Similarly to what was observed on Magnycours-48, for Memcached/Set, RCL improves performance more than it should (1.3 times, better than other lock algorithms) given the relatively low amount of time it spends in critical sections (20.2%). Again this is due to the large number of cache misses per critical section for that experiment: 73.4 L1 cache misses.[14] This number is more than halved when RCL is used: as seen in Figure 14(b), with RCL, critical sections only trigger 32.3 L1 cache misses because large parts of the hashtable remain in the server's cache hierarchy. Again, detailed results (see Figure 13(d)) show that using RCL improves scalability as well as performance. As a side note, Figure 14(b) also shows that on Niagara2-128, locality is improved for all applications, even when the number of cache misses per critical section is low.

Highlights. On Magnycours-48, RCL improves perfomance and scalability for Memcached/Get, as expected from the profiler results. On both Magnycours-48 and Niagara2-128, RCL improves performance more than expected from the profiler results for Memcached/Set, because critical sections access a lot of shared data, and RCL benefits from improved locality over other lock algorithms.

5.4.5. Berkeley DB with TpccOverBkDb. As can be seen in Figures 8 and 9, using RCL in Berkeley DB with TpccOverBkDb is more complex than using it in other applications because in this case, multiple locks are contended. Therefore, in order to reach optimal performance with RCL, a preliminary analysis is needed to decide how many servers

[14] This number is higher than the number of cache misses for Memcached/Set on Magnycours-48 because on Niagara2-128, L2 cache misses are measured instead of L1 cache misses. L1 cache misses are triggered more often than L2 cache misses because L1 caches are smaller than L2 caches, and on Niagara2-128, eight hardware threads share each L1 cache, which leads to frequent cache line evictions.

		L2 CMs in base app.	# RCL server L2 CMs / CS	
			S1	S2
SPLASH-2	Radiosity	1.7		1.5
	Raytrace / Car	0.6		1.0
	Raytrace / Balls4	1.4		1.0
Phoenix 2	Linear Regression	3.8		2.1
	String Match	4.5		2.2
	Matrix Multiply	3.1		2.2
Memcached	Get	2.1		3.0
	Set	*16.5*		*5.7*
Berkeley DB + TPC-C	Order Status	2.4	2.7	2.8
	Stock Level	2.4	2.6	2.7

(a) L2 cache misses on Magnycours-48

		L1 CMs in base app.	# RCL server L1 CMs / CS		
			S1	S2	S3
SPLASH-2	Radiositya	5.4			2.0
	Raytrace / Car	4.6			4.1
	Raytrace / Balls4	3.8			3.0
Memcached	Get	13.5			10.7
	Set	*73.4*			*32.3*
Berkeley DB + TPC-C	Order Status	4.0	2.1	2.8	3.1
	Stock Level	3.4		4.9	0.5

(b) L1 cache misses on Niagara2-128

All applications are run with their maximum number of threads (h threads for SPLASH-2 and Phoenix 2, $h/2-2$ threads for Memcached), except Berkeley DB with TpccOverBkDB, for which one application thread per hardware thread is used. In contrast with Figures 5c and 6c, these numbers do not include the cache misses caused by the RCL algorithm itself.

Fig. 14. Number of cache misses per critical section on the RCL server.

to use and which server should handle which locks. In Section 5.4.5.a, we first perform this analysis in order to find an optimal configuration for RCL, and in Section 5.4.5.b we discuss the results of running Berkeley DB with TpccOverBkDb using all lock algorithms, including RCL with the optimal configuration we found. Since Berkeley DB with TpccOverBkDb can use more client threads than there are hardware threads on the machine, repeatedly yielding the processor in busy-wait loops can be beneficial when it does: in Section 5.4.5.c, we implement this optimization for most of the lock algorithms, including RCL. We then discuss the influence of this optimization on the performance of Berkeley DB with TpccOverBkDb.

5.4.5.a. Experimental Setup. A difficulty in transforming Berkeley DB for use with RCL is that the function call in the source code that allocates the most used locks allocates 11 locks in total in a loop, and not all of them suffer from high contention. A limitation of our transformation tool is that if several locks are created at the same position in the code in a loop, all allocated locks must be implemented in the same way, since the transformation tool uses line numbers (and file names) to identify locks. Consequently, all 11 locks are implemented as RCLs.

Using a single server to handle these locks would cause critical sections to be needlessly serialized, and using 11 servers would waste computing power that could be used for client threads. To prevent this, as explained in Section 3, the RCL runtime makes it possible to choose the server where each lock will be dispatched. However, in order to choose the number of servers used and to decide which locks will be handled by which server, the experiments must be run once in order to gather statistics about the execution, which reveals which locks are the most used. This analysis can either be performed with the profiler or with the statistics gathered by the RCL runtime that were presented in Section 3.2.3. We choose the second option in order to provide an illustration of how these statistics can be used.

In order to find which locks are the most used in Berkeley DB with TpccOverBkDb, after transformation, the experiment is run once with 11 RCL servers, with each server handling one of the locks that were found by the profiler. We run the experiment for both Order Status and Stock Level requests on Magnycours-48 and Niagara2-128. The results are shown in Figure 15(a). For Order Status, on Magnycours-48, the four most used locks are L10, L3, L4, and L6, in that order. On Niagara2-128, the four most used

		Use rate										
		L11	L10	L9	L8	L7	L6	L5	L4	L3	L2	L1
Order Status	Magnycours-48	1.4%	3.5%	1.4%	1.4%	1.5%	1.8%	1.6%	2.4%	2.6%	1.6%	1.6%
	Niagara2-128	0.7%	1.7%	0.7%	0.7%	0.7%	1.5%	0.9%	3.2%	2.7%	0.7%	0.7%
Stock Level	Magnycours-48	1.4%	5.5%	1.5%	6.7%	1.5%	1.5%	1.5%	1.5%	1.6%	1.6%	1.6%
	Niagara2-128	0.7%	4.1%	0.7%	5.6%	0.7%	0.7%	0.7%	0.7%	0.7%	0.7%	0.7%

(a) Use rate with one lock per hardware thread

# srv.	Server configuration	Lock locations										
		L11	L10	L9	L8	L7	L6	L5	L4	L3	L2	L1
1	All locks on same server	S1	S1	S1	S1	S1	S1	S1	S1	S1	S1	S1
2	2 most cont. locks on ≠ serv. (OS & SL)	S2	S2	S2	S1	S2	S2	S1	S1	S1	S1	S1
3	3 most cont. locks on ≠ serv. (OS)	S3	S3	S3	S2	S2	S2	S1	S2	S1	S1	S1
4	4 most cont. locks on ≠ serv. (OS)	S4	S4	S3	S3	S2	S3	S2	S2	S1	S1	S1
11	One lock per server	S11	S10	S9	S8	S7	S6	S5	S4	S3	S2	S1

(b) RCL server configurations

Fig. 15. Server configurations for Berkeley DB with TpccOverBkDB.

locks are L4, L3, L10, and L6. For Stock Level, the two most used locks are L10 and L8 on Magnycours-48 and L8 and L10 on Niagara2-128, with all other locks being used much less.[15] Using this information, we construct the five server configurations listed in Figure 15(b). The first and the last configurations put all locks on one server or use one lock for each server, respectively. The three other configurations place the two, three, and four most used locks of the Order Status experiment on different servers in such a way that the same configurations exhibit this characteristic on both machines. Other locks are distributed so that all servers handle a similar number of locks. Additionally, Configuration 2 places the two most used locks of the Stock Level experiment on two different servers for both machines, with other locks being evenly distributed on both machines. Since, with Stock Level, two locks are used much more than all others, and these other locks are all equally used, there is no need to use a configuration for three or four servers for that type of request.

The experiments are run again for each configuration on Magnycours-48 and Niagara2-128, with one client thread per hardware thread. Figure 16 shows the results. The configuration that gives the maximum number of transactions per second is considered optimal and is used for the performance evaluation in Sections 5.4.5.b and 5.4.5.c.

Figure 16(a) shows the results for Magnycours-48. For Order Status, when using only one server, the false serialization rate is high (66.3%), which indicates that many independent critical sections are needlessly executed in mutual exclusion. Adding more servers decreases the false serialization rate: with two, three, four, and 11 servers, the false serialization rate drops to 34.1%, 10.6%, 2.5%, and 0.0%, respectively. Similarly, the use rate drops from 22.7% to 7.9%, 4.0%, 3.2%, and 1.9%. This decrease is not only due to the load being shared between servers. Going from one server to two servers divides the use rate by more than 2 (2.87) because the execution is faster with two servers (+21%) than with one server due to the decreased amount of false serialization. With one server, false serialization makes the server busier, which leads to more clients waiting for their requests to be executed. Adding more servers reduces false serialization but also wastes hardware threads that could be used for clients,

[15] While the use rates differ slightly between Magnycours-48 and Niagara2-128, the lists of most used locks are globally the same on both machines: only their order differs. The fact that the profiling results are quantitatively similar on two machines with very different architectures could indicate that this profiling phase may not be needed for every new machine the application is run on.

	# srv.	Tr. / s	Use rate					False serialization rate				
			S4	S3	S2	S1	Avg.	S4	S3	S2	S1	Avg.
Order Status	1	19,357				22.7%	22.7%				66.3%	66.3%
	2	*23,556*			*7.3%*	*8.4%*	*7.9%*			*23.9%*	*44.2%*	*34.1%*
	3	23,057		4.8%	3.8%	3.4%	4.0%		0.0%	21.2%	10.7%	10.6%
	4	23,432	4.7%	2.1%	3.1%	3.0%	3.2%	0.0%	0.0%	10.1%	0.0%	2.5%
	11	21,706	<4.0% for all 11 servers				1.9%	0.0% for all 11 servers				0.0%
Stock Level	1	1,905				58.5%	58.5%				46.4%	46.4%
	2	*2,351*			*9.4%*	*11.5%*	*10.5%*			*0.1%*	*0.0%*	*0.0%*
	11	2,286	<7.0% for all 11 servers				2.4%	0.0% for all 11 servers				0.0%

(a) RCL server statistics on Magnycours-48

	# srv.	Tr. / s	Use rate					False serialization rate				
			S4	S3	S2	S1	Avg.	S4	S3	S2	S1	Avg.
Order Status	1	11,100				20.3%	20.3%				45.1%	45.1%
	2	18,551			1.8%	2.0%	1.9%			18.4%	35.3%	26.9%
	3	*23,925*		*1.8%*	*2.2%*	*1.7%*	*1.9%*		*0.0%*	*26.8%*	*12.7%*	*13.1%*
	4	23,319	1.7%	1.1%	2.4%	1.8%	1.8%	0.0%	0.0%	15.9%	0.0%	4.0%
	11	22,479	<4.0 for all 11 servers				1.3%	0.0% for all 11 servers				0.0%
Stock Level	1	844				10.9%	10.9%				44.2%	44.2%
	2	*946*			*3.8%*	*4.1%*	*3.9%*			*0.9%*	*0.0%*	*0.4%*
	11	939	<6.0 for all 11 servers				2.4%	0.0% for all 11 servers				0.0%

(b) RCL server statistics on Niagara2-128

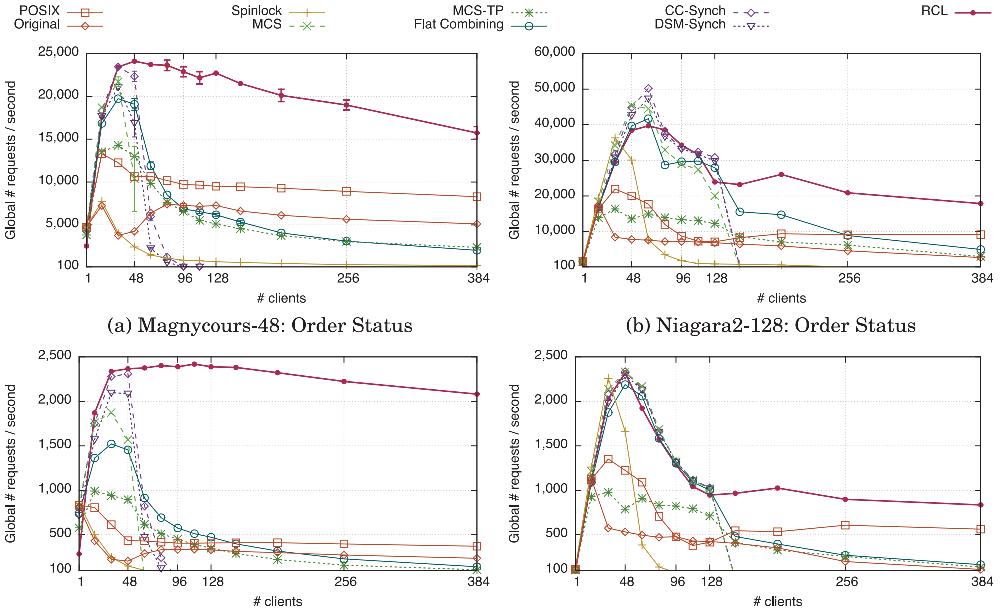
Fig. 16. Impact of false serialization with RCL (Berkeley DB with TpccOverBkDb).

which is why the peak performance is reached for two servers only. Using three or four servers provides similar performance, which indicates that even if a developer does not perform a precise contention analysis and mistakenly uses a few more servers than needed, the resulting overhead should be low.

For Stock Level, with only one server, both the false serialization rate and the use rate are high (46.4% and 58.5%, respectively). Using two servers is sufficient to make the false serialization rate drop to almost zero (0.4%). The use rate is divided by 6 which shows that with two servers much fewer clients are waiting for their critical sections to be executed. Unsurprisingly, the peak performance is obtained for two servers, but interestingly, using 11 servers is only slightly slower than using two servers (<3% performance drop), which shows again that using more servers than needed does not seem to decrease performance significantly: using trial and error to try to decrease the false serialization rate should be sufficient for developers to obtain good results.

Figure 16(b) shows the results for Niagara2-128. Again, increasing the number of servers decreases the false serialization rate and the use rate. However, since Niagara2-128 has more hardware threads than Magnycours-48, using more servers wastes relatively less CPU resources for the client threads, and the fact that a degraded version of the RCL runtime is used for Niagara2-128 makes false serialization costlier. Because of these two factors, more servers may be needed to reach peak performance: for Order Status, peak performance is reached for three servers instead of two, with a much lower use rate (13.1%) than for the peak configuration of Magnycours-48 (34.1%). Similarly to what was observed on Magnycours-48, for Stock Level, using two servers is enough to remove almost all false serialization. Moreover, adding more servers than needed only slightly decreases performance: using 11 servers instead of the best configuration leads to an overhead of only 6.0% for Order Status, and 0.7% for Stock Level.

5.4.5.b. Performance Analysis. Figure 17 shows the performance of Berkeley DB with TpccOverBkDb, using the server configurations chosen earlier. In these experiments,



(a) Magnycours-48: Order Status

(b) Niagara2-128: Order Status

(c) Magnycours-48: Stock Level

(d) Niagara2-128: Stock Level

Five runs per data point, error bars show standard deviation when relative standard deviation is > 5%.

Fig. 17. Berkeley DB with TpccOverBkDb speedup.

each client uses its own thread, and up to 384 clients are run concurrently, which is more than the number of hardware threads for both machines.

Detailed results for Magnycours-48. Figures 17(a) and 17(c) show the results for Magnycours-48. For both Order Status and Stock Level requests, MCS collapses when more client threads are used than hardware threads because of the convoy effect. MCS-TP, which was designed to prevent convoys, does not suffer from this issue but its peak performance is much lower than that of MCS. CC-Synch and DSM-Synch exhibit good peak performance, but these lock algorithms also collapse rapidly after $h_{m48} = 48$ client threads, that is, when there are more client threads than hardware threads because of the design flaw presented in Section 2.3.4 that makes a combiner hand over its role to a preempted thread. The basic spinlock performs very poorly except when the number of client threads is low. At 384 client threads, only RCL, the POSIX lock, Berkeley DB's custom lock algorithm (noted “Original” in the figures), Flat Combining, and MCS-TP have not collapsed. Interestingly, the POSIX lock is faster than Berkeley DB's custom lock, which may be due to the fact that the lock implementation of Berkeley DB is old: it has been designed on machines with much less than 48 hardware threads. RCL always performs better than all other lock algorithms: it has the best peak performance (24.1K transactions per seconds for Order Status and 2.4K for Stock Level), and at 384 client threads, for Order Status (Stock Level, respectively), using RCL makes the experiment's throughput 89.6% (460.7%, respectively) higher than with the next best lock algorithm (POSIX), and 207.3% (783.4%, respectively) higher than the base application.

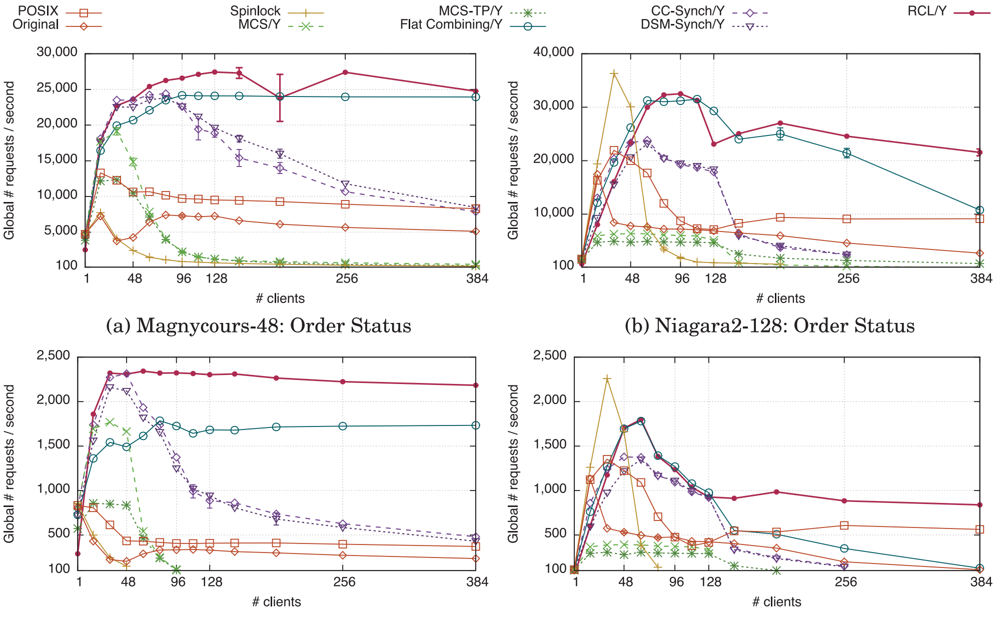
Detailed Results for Niagara2-128. Figures 17(c) and 17(d) show the results for Niagara2-128. For Order Status requests, RCL has a lower peak performance (39.7K transactions per second) than CC-Synch (50.2K t/s), DSM-Synch (47.4K t/s), MCS (45.4K t/s), and Flat Combining (41.7K t/s). For Stock Level requests, RCL's peak throughput is similar to CC-Synch's, DSM-Synch's, and MCS-TP's. However, MCS, CC-Synch, and DSM-Synch collapse rapidly after $h_{n128} = 128$ client threads, that is, when there are more client threads than hardware threads, which may be due to the convoy effect. The basic spinlock performs poorly except when the number of client threads is low. Moreover, when there are more client threads than hardware threads, RCL performs better than all other lock algorithms: at 384 client threads, for Order Status (Stock Level, respectively), using RCL makes the throughput of the experiment 95.3% (48.3%, respectively) higher than with the next best lock algorithm (POSIX), and 557.3% higher (761.4%, respectively) than the base application.

Highlights. MCS, CC-Synch, and DSM-Synch collapse when there are more client threads than there are hardware threads on the machine. On Magnycours-48, RCL always outperforms all other lock algorithms. On Niagara2-128, RCL does not always outperform all other lock algorithms when there are less client threads than hardware threads on the machine, but it always does otherwise. To conclude, RCL performs especially well when there are more client threads than hardware threads on the machine.

5.4.5.c. Yielding the Processor in Busy-Wait Loops. As was explained in Section 2, when there are more application threads than there are hardware threads on the machine, lock algorithms that use busy-waiting other than RCL can get preempted while they execute critical sections, with application threads waking up and wasting their time quantum busy-waiting: this can lead to slowdowns and convoys. RCL, on the other hand, always makes progress because its server threads run undisturbed on dedicated hardware threads. However, one could suspect that the experiments from Section 5.4.5 are unfair for lock algorithms other than RCL, because a lot of these lock algorithms could simply repeatedly yield the processor in their busy-wait loops. This technique (i) prevents waiting threads from needlessly wasting CPU resources that could be used by a thread that can actually make progress on the critical path, and (ii) allows for faster progress outside the critical path: by reducing the load of the machine, yielding the processor also makes it possible for clients to perform other work than busy-waiting. Consequently, the transformation can also be applied to RCL, which will only benefit from that second advantage. In order to investigate the consequences of this optimization, we run the experiment again with modified versions of the lock algorithms (including RCL) that repeatedly yield the processor in busy-wait loops.

The lock algorithms are modified as follows. The POSIX lock and Berkeley DB's custom lock are not altered because they use blocking, which already yields the processor. The basic spinlock is also not altered since making it yield the processor in busy-wait loops would make it behave like an inefficient blocking lock. For noncombining locks, application threads repeatedly yield the processor instead of busy-waiting on their synchronization variable. Finally, for combining locks and RCL, clients threads are made to repeatedly yield the processor when they are waiting for the combiner or server to execute their critical section.

Detailed Results for Magnycours-48. Figures 18(a) and 18(c) show the results for Magnycours-48. Yielding the processor in busy-wait loops reduces the collapse of MCS, CC-Synch, and DSM-Synch. It greatly improves the performance of Flat Combining. With yielding, Flat Combining is much more efficient than CC-Synch and DSM-Synch when there are more client threads than hardware threads because Flat Combining always hands over the role of combiner to a thread that is scheduled, whereas CC-Synch and DSM-Synch can hand over the role of combiner to a preempted thread, which significantly slows down the critical path. With Stock Level, however, CC-Synch and DSM-Synch have a better peak performance than Flat Combining. RCL is still more efficient than all other lock algorithms for any number of clients. Yielding the processor improves the throughput with RCL when a lot of clients are used (+57.3% at 384 threads with Order Status, +5.0% at 384 threads for Stock Level), because



(a) Magnycours-48: Order Status (b) Niagara2-128: Order Status

(c) Magnycours-48: Stock Level (d) Niagara2-128: Stock Level

Five runs per data point, error bars show standard deviation when relative standard deviation is > 5%.

Fig. 18. Berkeley DB with TpccOverBkDb speedup, using yielding.

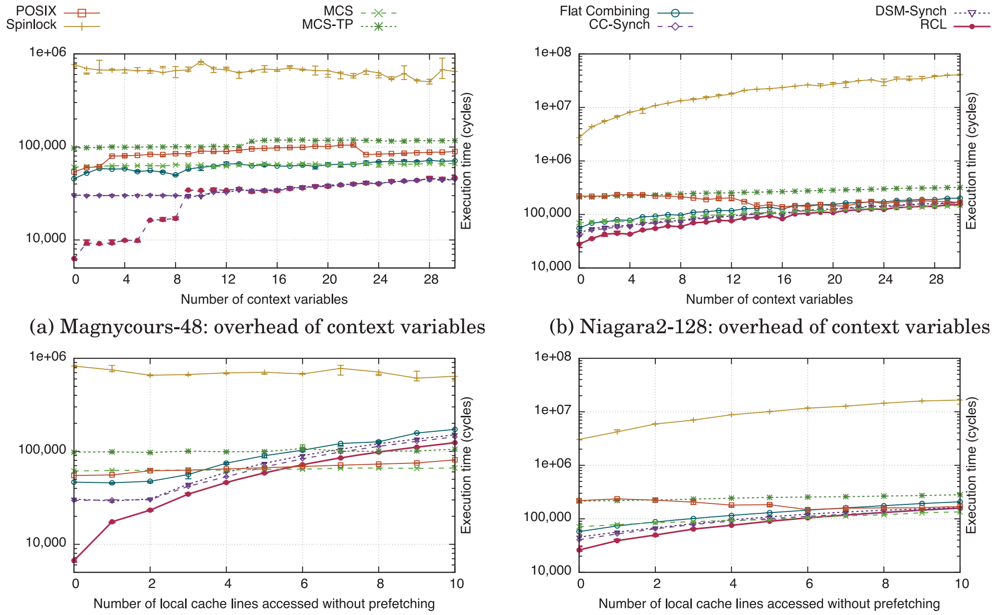
clients are able to supply the server with more concurrent requests. This effect is more visible with Order Status than for Stock Level because Order Status has a much higher throughput. At 384 client threads, for Order Status (Stock Level, respectively), using RCL makes the experiment's throughput 3.4% (26.0%, respectively) higher than with the next best lock algorithm (Flat Combining), and 383.6% (826.9%, respectively) higher than the base application.

Detailed Results for Niagara2-128. On Niagara2-128, yielding the processor in busy-wait loops alleviates the collapse of MCS, CC-Synch, and DSM-Synch, but it also significantly reduces the peak performance of these lock algorithms. Yielding improves throughput in Order Status when a lot of clients are used (+20.6% at 384 threads). RCL has the best peak performance (equal to that of Flat Combining) and the best performance of all of the lock algorithms at 384 threads, even when all the lock algorithms yield the processor in busy-wait loops. At 384 client threads, for Order Status (Stock Level, respectively), using RCL makes the experiment's throughput 100.0% (562.6%, respectively) higher than with the next altered lock algorithm (Flat Combining), and 692.7% higher (564.2%, respectively) than the base application.

Highlights. Even though yielding the processor in busy-wait loops can improve the performance of some lock algorithms when applications use more software threads than there are hardware threads, it also improves the performance of RCL at high throughput, and RCL still performs better than other lock algorithms with this optimization.

5.5. Specialized Benchmarks

This section presents specialized experiments that give a more complete picture of the performance of RCL. We first focus on the overheads of (i) context variables in critical



(a) Magnycours-48: overhead of context variables

(b) Niagara2-128: overhead of context variables

(c) Magnycours-48: overhead of local pointers

(d) Niagara2-128: overhead of local pointers

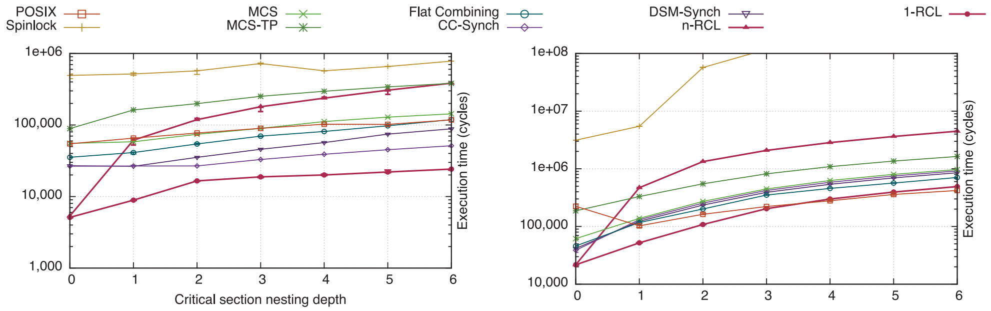
Five runs per data point, error bars show standard deviation when relative standard deviation is > 5%.

Fig. 19. Overhead of context variables and local pointers.

sections, and (ii) nested critical sections. Then, we briefly discuss the performance of an energy-aware version of RCL.

5.5.1. Overhead of Context Variables and Local Pointers. We alter our microbenchmark (presented in Section 5.3) to make it access (increment) a varying number of 32-bit integer context variables, stored on adjacent cache lines to emulate the behavior of our transformation tool (which first copies the context variables to a structure before executing the critical section). On Magnycours-48, we optimize the process by storing the first eight context variables in the padding area at the end of the client's mailbox (hatched area in Figure 1); we do not use this optimization on Niagara2-128 because we find it slightly worsens performance, due to a higher sensitivity to contention on the mailbox's cache line. The results are shown in Figures 19(a) and 19(b). On Magnycours-48, RCL is faster than all other lock algorithms with eight context variables or less, while its performance is similar to that of CC-Synch and DSM-Synch when more context variables are used. Increasing the number of context variables has less overhead on traditional lock algorithms than on combining locks and on RCL: this is to be expected, since context variables do not worsen locality in the case of traditional lock algorithms. However, barring the first eight context variables with RCL, this effect is very slight thanks to efficient prefetching. On Niagara2-128, even though this effect is much more obvious, it still takes more than 24 context variables before RCL falls to the performance of other lock algorithms.

Another source of overhead for combining locks and RCL is accesses, in critical sections, to data that is local to the thread acquiring the lock. These accesses are generally made through context variables that are pointers. We evaluated the corresponding overhead by accessing (incrementing) multiple cache lines in critical sections, using dependent randomized accesses to ensure they could not be prefetched (using a similar technique to the one used by Yotov et al. [2005]). This is a worst-case scenario



(a) Magnycours-48 (b) Niagara2-128

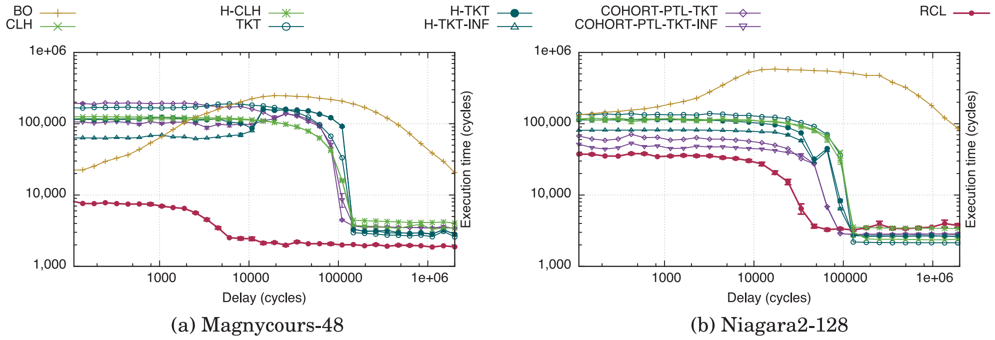
Five runs per data point, error bars show standard deviation when relative standard deviation is > 5%.

Fig. 20. Overhead of nested critical sections.

for combining locks and RCL, since each access writes to a different cache line, without prefetching: local accesses to variables that are close enough in memory to be on the same cache line or to allow for prefetching would have much lower overhead. The results are shown in Figures 19(c) and 19(d). The overhead of accessing nonprefetched local variables is significant for combining locks and RCL: it takes about six accesses for RCL to fall to the level of MCS. Consequently, combining locks and RCL should not be used for critical sections that access a lot of local data.

Highlights. On Magnycours-48 (Niagara2-128, respectively), RCL outperforms other lock algorithms when the number of context variables is low; if this number is above eight (24, respectively), RCL performs the same as CC-Synch/DSM-Synch (MCS, respectively). On both Magnycours-48 and Niagara2-128, RCL and combining locks have an overhead when critical sections access pointers that are local to the client threads, and when critical sections access more than six cache lines through local pointers, RCL performs similarly or worse than traditional lock algorithms.

5.5.2. Overhead of Nested Critical Sections. We evaluate the overhead of nested critical sections by altering the critical sections of our microbenchmark: this time, we acquire a varying number of nested locks, always in the same order, before incrementing one global variable. The results are shown in Figure 20. 1-RCL corresponds to RCL using a single server, while n-RCL corresponds to RCL using one server per lock. As shown on the graph, on Magnycours-48, the implementation of 1-RCL is very efficient: adding nested critical sections does not have a higher overhead for RCL than for other lock algorithms. On Niagara2-128, the performance of 1-RCL is not as good, because, as explained before, we are running RCL in a degraded mode on this machine (no FIFO scheduling because we do not have `proc_priocntl` Solaris 10 privilege). n-RCL, however, exhibits bad performance: a single level of nesting is sufficient to make it perform worse than most other lock algorithms. This is due to the fact that the inner locks have no contention at all, and for most lock algorithms, this means obtaining them almost instantly: all MCS needs to do is to enqueue a node in an empty list, for instance. For n-RCL, a transfer of control is needed, which is much costlier, especially since when contention is very high, each thread has to wait for all other threads to perform one transfer of control before they can execute their critical section, on average. Developers must therefore be careful to not use multiple servers for nested locks with RCL. Moreover, even if all nested locks are assigned to the same server, significant overhead is to be expected if some of the inner locks may not always be nested.



(a) Magnycours-48 (b) Niagara2-128

Five runs per data point, error bars show standard deviation when relative standard deviation is > 5%.

Fig. 21. Comparison with more locks.

Highlights. RCL performs similarly or better than other lock algorithms when nested critical sections are used, as long as all nested locks are handled by the same server (and not contended by external sources). Using several RCL servers for nested critical sections has major overhead: RCL is outperformed by all other lock algorithms in this case.

5.5.3. Energy-Aware RCL. Energy efficiency is beyond the scope of this article, since all lock algorithms presented in this article, with the exception of blocking locks, rely heavily on busy-waiting. We still briefly experimented with an energy-aware version of RCL on Magnycours-48 to check whether such a design is viable. The energy-aware version of RCL uses the SSE3 `MONITOR`/`MWAIT` instructions on the client side when waiting for a reply from the server: instead of busy-waiting, a client enters power-saving mode until its critical section has been executed. Our results show that the energy-aware version of RCL has an overhead of about 30% on our microbenchmark.

5.6. Comparison with More Locks

For clarity, until this point in the evaluation, we only compared RCL to a subset of the lock algorithms that were mentioned in the article. We now compare the performance of RCL with that of the lock algorithms that were presented in Section 2.4, using our microbenchmark. The results are shown in Figure 21. The locks we use are a backoff lock with random backoff delay (BO), the CLH queue lock [Craig 2003; Magnusson et al. 1994], a hierarchical version of the CLH queue lock [Luchangco et al. 2006], a ticket lock, a hierarchical version of the ticket lock [David et al. 2013], and a Cohort Lock [Dice et al. 2012]. We used implementations from David et al. [2013] for all lock algorithms, except for the Cohort Lock. Since the authors of the Lock Cohorting paper were not able to provide us with their code due to licensing issues, we picked a Cohort Lock (COHORT-PTL-TKT) for which they presented a very detailed pseudocode in Dice et al. [2015], in order to reduce the chances of writing an suboptimal implementation. COHORT-PTL-TKT also happens to be one of the fastest cohort locks according to the evaluation in Dice et al. [2015]. We access five cache lines in critical sections in order to be in a favorable scenario for hierarchical locks (including Cohort Locks): with only one cache line access, the performance improvements of these lock algorithms over their nonhierarchical counterparts are low.

High contention. The backoff lock performs very well at the highest contention level, because we adjusted the maximum backoff delay at that level of contention. However, it performs much worse at other contention levels. CLH performs similarly to MCS, which is to be expected since both are queue locks. On our hardware, and with our microbenchmark, H-CLH only gives a very small performance improvement over MCS. On Magnycours-48, this is consistent with the Opteron results presented in David et al. [2013], and on Niagara2-128, we expected to obtain a lower performance improvement than in Dice et al. [2015] since our Niagara2 only has two sockets instead of four. The ticket lock performs poorly, and its hierarchical version proposed by [David et al. 2013], which is similar to the COHORT-TKT-TKT lock, performs similar to queue locks. While the COHORT-PTL-TKT lock performs poorly on Magnycours-48, it performs well on Niagara2-128: its performance is similar performance to that of CC-Synch and DSM-Synch.

The hierarchical ticket lock and Cohort Locks have a parameter that specifies how many critical sections can be executed consecutively on the same node. We used 1,024 in the curves that are marked H-TKT and COHORT-PTL-TKT, since it was found to be an optimal parameter in the evaluation by Dice et al. [2015]. On our hardware and with our microbenchmark, we noticed that we could improve the performance of H-TKT and COHORT-PTL-TKT, by setting this limit to a very high value: by sacrificing fairness, we can reach optimal performance for these two lock algorithms. Doing so greatly improves the performance of the hierarchical ticket lock (H-TKT-INF) and the Cohort Lock (COHORT-PTL-TKT-INF) on Magnycours-48, but even with this performance boost, both lock algorithms are still more than three times slower than CC-Synch and DSM-Synch. On Niagara2-128, H-TKT-INF performs worse than COHORT-PTL-TKT, but COHORT-PTL-TKT-INF performs very well, being even faster than CC-Synch and DSM-Synch. In any case, none of the evaluated lock algorithms perform as well as RCL.

Low Contention. On Magnycours-48, RCL performs better than other evaluated lock algorithms under low contention. This may be due to the fact that critical sections access five cache lines, which gives a locality advantage to RCL. On Niagara2-128, RCL performs worse than most locks under low contention.

Highlights. Under high contention, RCL outperforms a backoff lock, a ticket lock, and CLH. RCL also outperforms the hierarchical locks we evaluated (including a cohort lock), even when they were fine-tuned to disregard all fairness in order to maximize performance. Under low contention, however, RCL is often outperformed by other lock algorithms.

6. CONCLUSION

This article has presented RCL, a novel locking technique that focuses on both reducing lock acquisition time and improving the execution speed of critical sections through increased data locality. The key idea is to go one step further than combining locks, and to dedicate hardware threads for the execution of critical sections: since current multicore architectures have dozens of hardware threads at their disposal that typically cannot be fully exploited because applications lack scalability, dedicating some of these hardware threads for a specific task such as serving critical sections can only improve the application's performance. RCL takes the form of a runtime library for Linux and Solaris that supports ×86 and SPARC architectures. In order to ease the reengineering of legacy applications, RCL proposes a profiler as well as a methodology for detecting highly contended locks and locks whose critical sections suffer from poor data locality, since these two kinds of locks can generally benefit from RCL. Once these locks have been identified, using RCL is facilitated by a provided reengineering tool that encapsulates critical sections into functions. We show that RCL outperforms other existing lock algorithms on a microbenchmark and on some applications where locks are a bottleneck.

Later Works Related to RCL. Following the publication of RCL [Lozi et al. 2012], RCL has been used and improved in several other works. In particular:

—Hassan et al. [2014] propose a new algorithm for software transactional memory that executes commit and invalidation routines on dedicated remote server threads, with an implementation that uses RCL internally.
—Petrović et al. [2014] propose a universal construction inspired by RCL and combining locks that dedicates servers to the execution of critical sections on partially non-cache-coherent architectures.
—Pusukuri et al. [2014] propose to migrate threads across multicore architectures so that threads seeking locks are more likely to find them on the same core, which is similar to RCL in that it improves data locality of critical sections.
—Brandenburg [2013] compares the performance of real-time locking protocols for partitioned fixed-priority (P-FP) scheduling and cites RCL as an example of a distributed locking protocol: the DFLP algorithm they use is reminiscent of RCL.

Availability. The implementation of the RCL runtime, Liblock, the profiler, the reengineering tool, as well as test scripts and results are available at URL: `http://rclrepository.gforge.inria.fr`.

REFERENCES

Jose L. Abellán, Juan Fernández, and Manuel E. Acacio. 2011. GLocks: Efficient support for highly-contended locks in many-core CMPs. In *Proceedings of the 2011 IEEE International Parallel and Distributed Processing Symposium (IPDPS'11)*. IEEE Computer Society, Washington, DC, 893–905.

Gene M. Amdahl. 1967. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18–20, 1967, Spring Joint Computer Conference (AFIPS'67 (Spring))*. ACM, New York, NY, 483–485.

Thomas E. Anderson. 1990. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems* 1, 1 (Jan. 1990), 6–16.

Marc Auslander, David Edelsohn, Orran Krieger, Bryan Rosenburg, and Robert Wisniewski. 2003. Enhancement to the MCS lock for increased functionality and improved programmability. (Oct. 2003). U.S. Patent Application No. 10/128,745.

Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principle (SOSP'09)*. ACM, New York, NY, 29–44.

Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. 2008. Corey: An operating system for many cores. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, 43–57.

Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2010. An analysis of linux scalability to many cores. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association.

Bjorn B. Brandenburg. 2013. Improved analysis and evaluation of real-time semaphore protocols for P-FP scheduling. In *Proceedings of the 2013 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'13)*. IEEE Computer Society, Washington, DC, 141–152.

Alex Brodsky, Faith Ellen, and Philipp Woelfel. 2006. Fully-adaptive algorithms for long-lived renaming. In *Proceedings of the 20th International Conference on Distributed Computing (DISC'06)*. Springer-Verlag, Berlin, 413–427.

Milind Chabbi, Michael Fagan, and John Mellor-Crummey. 2015. High performance locks for multi-level NUMA systems. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles* *and Practice of Parallel Programming (PPoPP 2015)*. ACM, New York, NY, 215–226. DOI:http://dx.doi.org/10.1145/2688500.2688503

Travis S. Craig. 2003. *Building FIFO and Priority-Queueing Spin Locks from Atomic Swap*. Technical Report TR 93-02-02. Department of Computer Science, University of Washington.

Danga Interactive. 2003. Memcached: Distributed Memory Object Caching System. Retrieved from http://memcached.org.

Data Differential. 2011. Libmemcached. Retrieved from https://launchpad.net/libmemcached.

Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2013. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*. ACM, New York, NY, 33–48.

Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified data processing on large clusters. *Communication of the ACM* 51, 1 (Jan. 2008), 107–113.

Dave Dice, Virendra J. Marathe, and Nir Shavit. 2011. Flat-combining NUMA locks. In *Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'11)*. ACM, New York, NY, 65–74.

David Dice, Virendra J. Marathe, and Nir Shavit. 2012. Lock cohorting: A general technique for designing NUMA locks. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12)*. ACM, New York, NY, 247–256.

David Dice, Virendra J. Marathe, and Nir Shavit. 2015. Lock cohorting: A general technique for designing NUMA locks. *ACM Transactions on Parallel Computing* 1, 2, Article 13 (Feb. 2015), 42 pages. DOI:http://dx.doi.org/10.1145/2686884

Edsger W. Dijkstra. 1965. Cooperating sequential processes. (Sept. 1965). Published as EWD:EWD123pub.

Jonathan Eastep, David Wingate, Marco D. Santambrogio, and Anant Agarwal. 2010. Smartlocks: Lock acquisition scheduling for self-aware synchronization. In *Proceedings of the 7th International Conference on Autonomic Computing (ICAC'10)*. ACM, New York, NY, 215–224.

Panagiota Fatourou and Nikolaos D. Kallimanis. 2011. Sim: A Highly-Efficient Wait-Free Universal Construction. Retrieved from https://code.google.com/p/sim-universal-construction/.

Panagiota Fatourou and Nikolaos D. Kallimanis. 2012. Revisiting the combining synchronization technique. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12)*. ACM, New York, NY, 257–266.

Brad Fitzpatrick. 2004. Distributed caching with memcached. *Linux Journal* 2004, 124 (Aug. 2004), 5–5.

Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA.

Ahmed Hassan, Roberto Palmieri, and Binoy Ravindran. 2014. Remote invalidation: Optimizing the critical path of memory transactions. In *Proceedings of the 2014 IEEE International Parallel and Distributed Processing Symposium (IPDPS'14)*. IEEE Computer Society.

Bijun He, William N. Scherer III, and Michael L. Scott. 2005a. Preemption adaptivity in time-published queue-based spin locks. In *Proceedings of the 11th International Conference on High Performance Computing (HiPC'05)*. 7–18.

Bijun He, William N. Scherer III, and Michael L. Scott. 2005b. Time-Published Queue-Based Spin Locks. Retrieved from http://www.cs.rochester.edu/research/synchronization/pseudocode/tp_lock s.html.

Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010a. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'10)*. ACM, New York, NY, 355–364.

Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010b. Flat Combining and the Synchronization-Parallelism Tradeoff. (2010). http://mcg.cs.tau.ac.il/projects/flat-combining.

Maurice Herlihy and Nir Shavit. 2008. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA.

Charles Antony Richard Hoare. 1974. Monitors: An operating system structuring concept. *Communications of the ACM* 17, 10 (Oct. 1974), 549–557.

David Koufaty, Dheeraj Reddy, and Scott Hahn. 2010. Bias scheduling in heterogeneous multi-core architectures. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys'10)*. ACM, New York, NY, 125–138.

Scott T. Leutenegger and Daniel Dias. 1993. A modeling study of the TPC-C benchmark. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD'93)*. ACM, New York, NY, 22–31.

Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia Lawall, and Gilles Muller. 2012. Remote core locking: Migrating critical-section execution to improve the performance of multithreaded applications. In

Proceedings of the 2012. USENIX Annual Technical Conference (USENIX ATC'12). USENIX Association, 65–76.

Victor Luchangco, Dan Nussbaum, and Nir Shavit. 2006. A hierarchical CLH queue lock. In *Proceedings of the 12th International Conference on Parallel Processing (Euro-Par'06)*. Springer-Verlag, Berlin, 801–810.

Peter Magnusson, Anders Landin, and Erik Hagersteny. 1994. Queue locks on cache coherent multiprocessors. In *Proceedings of the 8th International Parallel Processing Symposium (IPPS'94)*. IEEE Computer Society Press, 165–171.

John M. Mellor-Crummey and Michael L. Scott. 1991a. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems* 9, 1 (Feb. 1991), 21–65.

John M. Mellor-Crummey and Michael L. Scott. 1991b. Synchronization without contention. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*. ACM, New York, NY, 269–278.

Michael A. Olson, Keith Bostic, and Margo Seltzer. 1999. Berkeley DB. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (USENIX ATC'99)*. USENIX Association, 43–43.

Oracle Corporation. 2004. Berkeley DB. Retrieved from http://www.oracle.com/technetwork/database/berkeleydb.

John K. Ousterhout. 1982. Scheduling techniques for concurrent systems. In *Proceedings of the 3rd International Conference on Distributed Computing Systems (ICDCS'82)*. 22–30.

Yoshihiro Oyama, Kenjiro Taura, and Akinori Yonezawa. 1999. Executing parallel programs with synchronization bottlenecks efficiently. In *Proceedings of the International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications (PDSIA'99)*.

Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. 2008. Documenting and automating collateral evolutions in Linux device drivers. In *Proceedings of the 3rd European Conference on Computer Systems 2008 (Eurosys'08)*. ACM, New York, NY, 247–260.

Darko Petrović, Thomas Ropars, and André Schiper. 2014. Leveraging hardware message passing for efficient thread synchronization. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'14)*. ACM, New York, NY, 143–154.

Kishore Kumar Pusukuri, Rajiv Gupta, and Laxmi Narayan Bhuyan. 2014. Lock contention aware thread migrations. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'14)*. ACM, New York, NY, 369–370.

Zoran Radovic and Erik Hagersten. 2003. Hierarchical backoff locks for nonuniform communication architectures. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA'03)*. IEEE Computer Society, Washington, DC, 241–253.

Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. 2007. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA'07)*. IEEE Computer Society, Washington, DC, 13–24.

David P. Reed and Rajendra K. Kanodia. 1979. Synchronization with eventcounts and sequencers. *Communications of the ACM* 22, 2 (Feb. 1979), 115–123. DOI:http://dx.doi.org/10.1145/359060.359076

Manish Shah, Jama Barreh, Jeff Brooks, Robert Golla, Gregory Grohoski, Nils Gura, Rick Hetherington, Paul Jordan, Mark Luttrell, Christopher Olson, Bikram Saha, Denis Sheahan, Lawrence Spracklen, and Aaron Wynn. 2007. UltraSPARC T2: A Highly-Threaded, Power-Efficient, SPARC SOC. Retrieved from http://www.oracle.com/technetwork/systems/opensparc/02-t2-a-sscc2007-15 30395.pdf.

Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. 1992. SPLASH: Stanford parallel applications for shared-memory. *SIGARCH Computer Architecture News* 20, 1 (March 1992), 5–44.

Stanford University. 2011. The Phoenix System for MapReduce Programming. Retrieved from http://mapreduce.stanford.edu.

Justin Talbot, Richard M. Yoo, and Christos Kozyrakis. 2011. Phoenix++: Modular MapReduce for shared-memory systems. In *Proceedings of the 2nd International Workshop on MapReduce and Its Applications (MapReduce'11)*. ACM, New York, NY, 9–16.

University of Delaware. 2007. The Modified SPLASH-2 Home Page. Retrieved from http://www.capsl.udel.edu/splash.

Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. 1995. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA'95)*. ACM, New York, NY, 24–36.

Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. 2010. Ad hoc synchronization considered harmful. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, 1–8.

Richard M. Yoo, Anthony Romano, and Christos Kozyrakis. 2009. Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC'09)*. IEEE Computer Society, Washington, DC, 198–207.

Kamen Yotov, Keshav Pingali, and Paul Stodghill. 2005. Automatic measurement of memory hierarchy parameters. In *Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'05)*. ACM, New York, NY, 181–192.

Received February 2015; revised September 2015; accepted November 2015