Efficient locking for multicore architectures

Jean-Pierre Lozi, Gaël Thomas, Julia L. Lawall, Gilles Muller

Efficient locking for multicore architectures

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

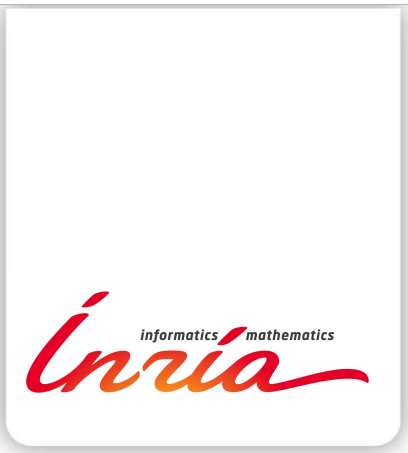
RESEARCH REPORT

N° 7779

November 2011

Project-Teams REGAL

ISSN 0249-6399 ISRN INRIA/RR--7779--FR+ENG



Efficient locking for multicore architectures

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

Project-Teams REGAL

Research Report n° 7779 — November 2011 — 27 pages

Abstract: The scalability of multithreaded applications on current multicore systems is hampered by the performance of critical sections, due in particular to the costs of access contention and cache misses. In this paper, we propose a new locking technique, Remote Core Locking (RCL) that aims to improve the performance 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 core. RCL limits the performance collapse observed with regular locks when many threads try to acquire a lock concurrently and removes the need to transfer lock-protected shared data to the core acquiring the lock: such data can typically remain in the server core's cache. Our microbenchmark shows that under high contention, RCL is always more efficient than the other state-of-the-art lock mechanisms, and a preliminary macrobenchmark evaluation shows performance gains on SPLASH-2 benchmarks (speedup up to 4.85) and on the Web cache application memcached (speedup up to 2.62).

Key-words: Locks, Multicore

* LIP6/INRIA
† DIKU,INRIA/LIP6
‡ INRIA/LIP6

RESEARCH CENTRE
PARIS – ROCQUENCOURT

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Verrouillage efficace pour les architectures multicores

Résumé : L'extensibilité des applications parallèles sur les architectures multicoeurs modernes est limitée par la performance des sections critiques, pour des raisons de contention sur le bus et de défauts de cache. Dans cet article, nous proposons une nouvelle approche pour l'implémentation des verrous, appelée Verrou À Distance (VAD), qui permet d'améliorer la performance des applications patrimoniales sur les architectures multicoeurs. L'idée du VAD est de remplacer les acquisitions de verrous par des appels de procédures à distance vers un ou plusieurs coeurs dédiés. Le VAD permet de limiter l'effet d'effondrement des performances observé avec les verrous classiques lorsque de nombreux fils d'exécution tentent d'acquérir simultanément un verrou. Le VAD évite également le transfert des données protégées par le verrou vers le coeur qui en fait l'acquisition. De fait, ces données restent dans le cache du coeur serveur. Sous haute contention, nos micro-évaluations montre que le VAD est toujours plus performant que l'état de l'art en matière de verrou. Sur des applications patrimoniales, nos expérimentations montrent un gain en performance pouvant aller jusqu'à 4.85 sur le banc d'essai SPLASH-2 et jusqu'à 2.62 sur le cache Web memcached.

Mots-clés : Verrous, Multicoeurs

1 Introduction

Over the last twenty years, a number of studies [2, 3, 6, 12, 13, 15, 20, 28, 31, 33] have attempted to optimize the execution of critical sections on multicore architectures, either by reducing access contention or by improving cache locality. Access contention occurs when many threads try to enter critical sections that are protected by the same lock simultaneously, causing the cache line associated with the lock to bounce between cores. Cache locality becomes a problem when a critical section accesses shared data that has recently been accessed on another core, resulting in cache misses, which greatly increase the critical section's execution time. Addressing access contention and cache locality together still remains a challenge.

Recently, several approaches have been proposed to execute a succession of critical sections on a single *server* core to improve cache locality [13, 33]. Such approaches also incorporate a fast transfer of control from other *client* cores to the server, to reduce access contention. Suleman *et al.* [33] propose a hardware-based solution that relies on the introduction of new instructions to perform the transfer of control, and the use of a special fast core to reduce the execution time of critical sections. Hendler *et al.* [13] propose a software-only solution, flat combining, in which the server is an ordinary client thread, and the role of server is handed off between clients periodically. This approach, however, slows down the thread playing the role of server, incurs an overhead for the management of the server role, and overall gives a substantial benefit only at very high rates of contention. Furthermore, it cannot accommodate threads that block within a critical section.

In this paper, we propose a new locking technique, Remote Core Locking (RCL), that aims to improve the performance of legacy multithreaded applications on multicore hardware by executing remotely, on a dedicated server, critical sections that access highly contended locks. Our approach is *entirely implemented in software* and targets commodity x86 multicore architectures. At the basis of our approach is the observation that most applications do not scale to the number of cores found in modern multicore architectures, and thus it is possible to *dedicate* the cores that do not contribute to improving the performance of the application to serving critical sections. Thus, it is not necessary to burden the application threads with the role of server, as done in flat combining. RCL also accommodates blocking within critical sections. The design of RCL addresses both access contention and locality. Contention is solved by a fast transfer of control to a server, using a dedicated cache line for each client to achieve busy-wait synchronization with the server core. Locality is improved because shared data is likely to remain in the server core's cache, allowing the server to access such data without incurring cache misses. In this, RCL is similar to flat combining, but has a lower overall overhead.

We propose a methodology along with a set of tools to facilitate the use of RCL in a legacy application. First, it is necessary to decide which locks to turn into RCLs. Because RCL serializes critical sections associated with locks managed by the same core, transforming all locks into RCLs on a smaller number of servers would induce a high overhead. Moreover, as also observed by Suleman *et al.* [32], some locks mainly access shared data while others also access substantial context data, i.e., data that are both used inside and outside the critical section. Transforming the latter lasts into RCL could induce a

high number of cache misses that could counterbalance the improvement in locality of accesses to shared data. The programmer must therefore decide which locks should be transformed into RCLs. On our benchmarks, we found that contexts are small, and thus the problem of cache misses for context data has only a marginal effect on performance. Instead, the main criterion is the lock contention rate, i.e., the number of failed lock acquisitions per second. We use mutrace [22] to extract this information. Second, the code within each critical section must be transformed so that it can be executed as an RPC; variables referenced or updated by the critical section that are declared by the function containing the critical section code must be sent as arguments to the server core. We have developed a tool that automates this transformation for C programs. Finally, we provide a runtime for Linux, that is compatible with POSIX threads and that supports a mixture of RCL and POSIX locks in a single application.

We have evaluated the performance of RCL on a custom latency microbenchmark measuring the execution time of critical sections that access a varying number of shared memory locations. Furthermore, based on the results of the profiler, we have identified memcached [21] and two benchmarks in the SPLASH-2 [30] suite as applications that could benefit from RCL. In each of these experiments, we compare RCL against the standard POSIX locks and the most efficient approaches for implementing locks of which we are aware: CAS spinlocks, MCS [20] and flat combining [13]. Comparisons are made for a same number of cores, which means that there are fewer application cores in the RCL case, since one or more cores are dedicated to RCL servers.

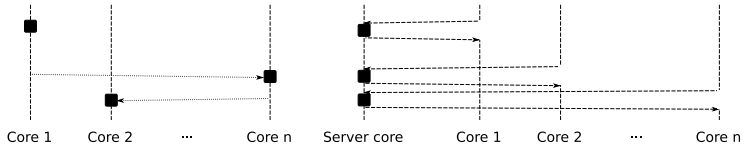
On an Opteron 6172 48-core machine running a 2.6.35 Linux kernel with glibc 2.12.1, our results are:

- Using our latency microbenchmark, under high contention, RCL is faster than all the other tested approaches, and more than three times faster than the second best approach, flat combining. In the worst case for RCL, at low contention and when the critical section accesses only a single shared cache line, RCL is the best after spinlocks with an overhead of only 15%.

- On the Raytrace application of the SPLASH-2 benchmark suite, which is the application of this benchmark suite having the highest lock contention, RCL provides a speedup of up to 4.85 times over POSIX locks, 3.11 times over MCS and 2.54 times over flat combining.

- On memcached, for `GET` requests, RCL provides a speedup of up to 2.62 times over POSIX locks, 1.34 times over spinlocks and 1.10 times over MCS. For `SET` requests, RCL provides a speedup of up to 2.42 times over POSIX locks, 1.59 times over spinlocks and 1.76 times over MCS.

The rest of the paper is structured as follows. Section 2 describes the design of RCL. Section 3 describes how to use profiling to automate the reengineering of a legacy application for use with RCL. Section 4 describes the results of our evaluation. Section 5 presents other work that has focused on improving locking on multicore architectures. Finally, Section 6 concludes.



(a) Traditional locks (b) Remote Core Locking (RCL)

Figure 1: Traditional locks (a) vs. RCL (b). With traditional locks, shared data migrate from one core to another, while with RCL shared data remain on the server core.

2 RCL Design Overview

The key idea of RCL is to transfer the execution of a critical section to a server core that is dedicated to one or more locks (Figure 1). This server core runs a pool of *servicing threads* that actively wait for requests to execute critical sections. We give an overview of the RCL core algorithm, describe how the RCL runtime system ensures liveness and responsiveness when a critical section blocks or waits for some event, and finally present the details of the algorithm.

2.1 Core algorithm

Using RCL, a critical section is replaced by a remote procedure call, to a procedure that executes the code of the critical section. To implement the remote procedure call, the clients and the server communicate through an array of request structures, specific to each server core, illustrated in Figure 2. This array has size $C \cdot L$ bytes, where C is a constant representing the maximum number of allowed clients and L is the size of the hardware cache line. Each request structure req_i is L bytes in length and allows communication between a specific client i and the server. The array is aligned so that each data block req_i is mapped to a single cache line.

The first three machine words of each request req_i contain respectively: (i) the address of a function that encapsulates the critical section for which the client c_i has requested the execution, or `NULL` if no execution of a critical section has been requested by this client, (ii) the address of a structure encapsulating

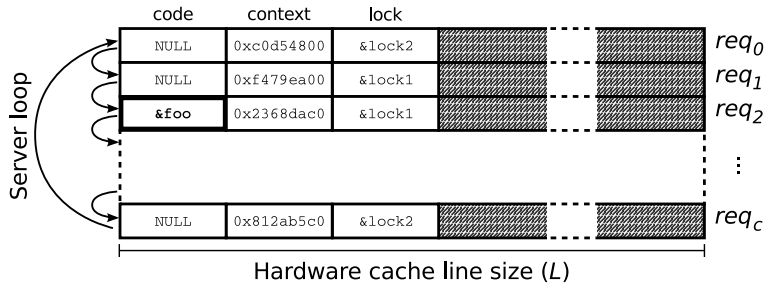


Figure 2: The request array. Client c_2, represented by req_2, has requested the execution of a critical section implemented by function foo.

the *context*, i.e., 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 the lock associated with the critical section.

Client side In order to execute a critical section, a client c_i first writes the address of the context structure into the second word of the structure req_i, then writes the address of the lock into the third word, and finally writes the address of the function that encapsulates the code of the critical section into the first word. The client then actively waits for the first word of req_i to be reset to `NULL`, indicating that the critical section has been executed by the server.

Server side A servicing thread iterates over the elements of the request array, waiting for one of the requests to contain a non-`NULL` value in its first word. When a non-`NULL` value is found, the servicing thread checks if the requested lock is free and, if so, acquires the lock 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 first word to `NULL`, and resumes iterating on the request array.

Management of the request array For efficiency, we have chosen to implement the request structures as an array, with one element for each potential client. Because the number of clients is not known in advance, the constant C is chosen to be very large. Managing the allocation of client identifiers is a classical problem of adaptive long-lived renaming [7], but such renaming incurs the cost of scanning entries to identify those that are unused. To minimize this cost, the server keeps track of the minimum and maximum client identifiers that are currently attributed, and only scans the entries between them.

2.2 Ensuring liveness and responsiveness

The core algorithm, as presented in Section 2.1, only refers to a single serving thread, and thus assumes that the servicing thread is never blocked at the OS level and never spins in an active waitloop. In practice, however, depending on the behavior of the critical sections, both of these conditions may occur. If a critical section tries to acquire a traditional POSIX lock that is already held, performs I/O, waits on a condition variable, or incurs a page fault, the servicing thread will be blocked at the OS level. Moreover, if the critical section tries to acquire another RCL or a spinlock or implements some form of ad hoc synchronization [35], then it may enter an active waitloop. A Linux thread can also be preempted at the OS level due to the expiration of a timeslice. These issues raise problems of *liveness* and *responsiveness*.

Liveness Blocking and waiting within a critical section raise a problem of liveness, because the server is not able to 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 spins in a critical section while waiting for a variable to be set by another critical section that is to run on the same server but is protected by a different lock (*e.g.*, clients C2 and C3 in Figure 3). To ensure liveness, the RCL runtime

ensures that when a servicing thread blocks or waits there always exists at least one other *free* servicing thread that is not currently executing a critical section and that this servicing thread will eventually be elected.

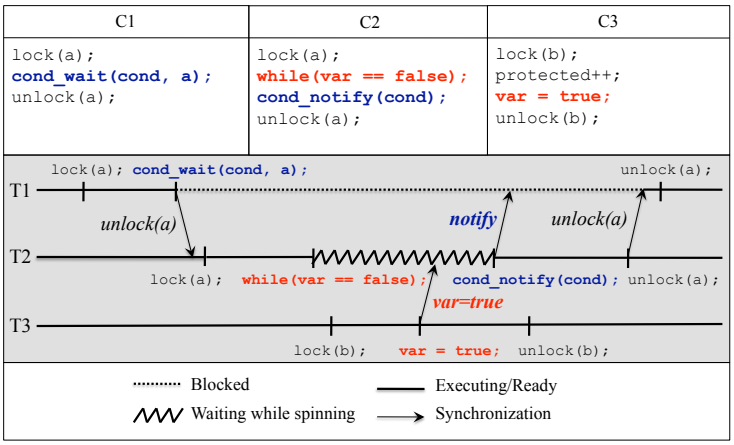


Figure 3: An example of ad hoc synchronization with blocking in I/O and waiting while spinning.

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*,[1] 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 going to sleep, 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 assumes that all servicing threads are either blocked or waiting. In this case, it checks that a free thread exists in the pool of servicing threads and, if not, it adds a new one.

Responsiveness 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. The RCL runtime implements a number of strategies to improve responsiveness, to address issues introduced by the underlying OS and by RCL design decisions.

A well known problem in the use of locks is the danger that the OS will preempt a thread at the expiration of a timeslice while it is executing a critical section, extending the duration of the critical section and thus increasing contention [24]. RCL dedicates a set of threads on dedicated cores to the execution of critical sections, which makes it possible to manage these threads according to a scheduling policy that does not use preemption. We use the Linux FIFO scheduling policy, which both respects priorities, as we require to ensure liveness, and has the property that a thread executes until it is blocked or yields. The use of the FIFO policy nevertheless raises a further liveness issue: if a servicing thread is in an active wait loop, it will not be preempted by the operating system, and a free thread will not be elected. When it detects no progress, the

[1] We used the Linux time-slice value as the timeout.

management thread thus also acts as a scheduler, electing a servicing thread 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: a servicing thread quits the pool whenever it observes that there is at least one other free servicing thread, since that other thread will be able to handle any 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 OS, we can exploit the OS scheduler to elect a new free thread immediately. Concretely, the RCL runtime maintains a *backup thread* that has lower priority than all servicing threads. The FIFO scheduling policy never elects a lower priority thread when a higher priority thread can run, and thus this backup thread is only elected when all servicing threads are blocked. When the backup thread is elected, 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 core and the lock is already owned by another servicing thread, the servicing thread immediately yields the processor in order to allow the owner of the lock to release it.

Example Figure 3 presents a complete example of ad hoc synchronization in a critical section that illustrates a worst case in term of active waiting. Initially, `T1` is the only servicing thread, and handles the request from `C1`. `C1`'s critical section contains a `cond_wait`, causing `T1` to block at the OS level. There is now no runnable servicing thread, and thus the OS scheduler immediately elects the backup thread, which adds a new free servicing thread, `T2`, which immediately preempts the backup thread. `T2` can now treat the request from `C2`. `C2`'s critical section causes `T2` to spin 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 `T1` is blocked in the OS and `T2` is spinning, no servicing thread has been able to set `is_alive`. As there is now no free servicing thread, the management thread creates a new servicing thread, `T3`, and elects it since it is now the only servicing thread that has not recently been elected. At this point, the server has therefore three servicing threads, `T1`, `T2` and `T3`, of which one, `T3`, is free. When `T3` executes the request from `C3`, it modifies the value of `var`.

After executing the request of `C3`, `T3` detects that there are other servicing threads and yields the processor to allow them to run. `T2` is the only other servicing thread that is not blocked at the OS level, and thus it is elected. As the critical section executed by `T3` has set `var`, `T2` can exit its waitloop and unblock the condition variable, causing `T1` to unblock. `C2`'s critical section is then completed, and `T2` observes that the server has two free servicing threads, itself and `T3`. `T2` thus leaves the servicing pool. `T3` is then elected, but observes that there is no pending critical section. Since it furthermore observes that there is another servicing thread and that thread is not free, it yields. `T1` is

then elected, and can complete the request of `C1`. It then observes that there are two free servicing threads, itself and `T3`, and leaves the servicing pool. The server returns to the initial state, with only one servicing thread: `T3`.

2.3 Algorithm details

In summary, a server core executes three kinds of threads: a high priority management thread, medium priority servicing threads, and a low priority backup thread, all scheduled using a FIFO scheduling algorithm. We describe the behavior of these threads, starting with the servicing threads, in more detail.

Serving RCLs A servicing thread executes the server loop, shown in Algorithm 1. This loop consists of two parts: a *fast path*, shown on lines 6-17, which is the only code that is executed when there is only one servicing thread in the pool, and a slower path, additionally including lines 18-29, that is only executed when multiple servicing threads exist to handle blocked critical sections. The fast path, specifically lines 10-16, implements the RCL server loop as described in Section 2.1.

A minimal amount of bookkeeping is required in the server loop. As the servicing thread can potentially be interrupted at any time due to a page fault, some care is needed to ensure the atomicity of these bookkeeping operations. In particular, within the server loop (lines 10-16), since the servicing thread is not free, it must decrement the number of free threads before this code (line 9) and increment the number of free threads afterwards (line 17). These operations are atomic. Additionally, the server loop must test whether the requested lock is held by another servicing thread. This test requires a compare-and-swap (CAS) operation (line 13). Both the atomic operations and the CAS are core-local and thus have a negligible impact on performance. In particular, we have reimplemented our own optimized version of these operations without using the x86 instruction `prefix` that locks the shared memory.

There is usually only one servicing thread in the FIFO queue, unless at least one other servicing thread has blocked and then unblocked. If the active servicing thread detects that other servicing threads are available, it yields the processor (lines 18-29) to allow another servicing thread to be elected, by sleeping if the value of `nb_free` indicates that more than one free servicing thread exists (line 20-27) or by explicitly yielding the processor (line 29). In the former case, the servicing thread takes a lock to avoid races on the variables `nb_servicing` and `is_servicing`. Note that the management of `nb_free` is lock free and thus between the observation of the counter at line 18 and the sleep at line 24, other free threads may begin executing critical sections. In this case, the management thread will eventually activate another free thread and thus ensure the liveness of the server.

Executing a critical section A critical section may try to acquire an RCL, which may be managed by the same core or by another core. In the former case, the servicing thread initializes the context, lock and code pointer, and waits for the code pointer to be cleared (Algorithm 2, lines 9-14). If the RCL is managed by the same core, the servicing thread must actively wait until the lock is free, during which time it repetitively yields, to give the CPU to the thread that owns the lock (lines 16-19).

Management and backup threads As shown in Algorithm 3, the management thread sleeps for the duration `timeout` (line 24) and then checks the flag `is_alive`. If this flag is set, the management thread clears it for the next timeout expiration. If the flag is not set, it means that none of the servicing threads has been able to progress since the previous timeout. In this case, the management thread ensures that at least one free thread exists (lines 9 and 28-43) and then forces the election (lines 12-21) of a thread that has not recently been elected, i.e., that has its `recently_elected` flag unset.

The backup thread (lines 24-26) simply calls a function that checks that there is at least one free thread. Recall that the backup thread is only activated when all the servicing threads are blocked in the kernel.

Algorithm 1: Structures and server thread

```
                  lock_t:        { server_t *server, bool is_locked,
                                   pthread_mutex_t posix_lock };
                  request_t:     { function_t code, pointer_t context, lock_t *lock };
                  thread_t:      { server_t *server, bool recently_elected,
   structures:                    bool is_servicing, thread_t* next};
                  server_t:      { posix_lock_t plock, posix_cond_t pcond,
                                   integer nb_servicing, bool is_alive,
                                   integer nb_threads, integer nb_free,
1                                  request_t *requests, thread_t threads }
2  global variables:    integer client_thread_number;
3  thread local variables:    request_t r;
4  function rcl_servicing_thread(thread_t *t)
5      var server_t* s := t->server;
6      while true do
7          s->is_alive := true;
8          t->recently_elected := true;
           // Serving RCL requests
9          atomic(s->nb_free--);
10         for i := 0 to client_thread_number do
11             r := s->requests[i];
               // Core local CAS, does not require a memory lock
12             if r->code ≠ null
13               and local_CAS(&r->lock->is_locked, false, true) = false then
                   // Execute the critical section
14                 r->code(r->context);
                   // Indicate client execution completion
15                 r->code := null;
16                 r->lock->is_locked := false;
17         atomic(s->nb_free++);
18         if s->nb_servicing > 1 then
19             if s->nb_free > 1 then
                   // Keep only a single free servicing thread
20                 lock(s->plock);
21                 s->nb_servicing--;
22                 t->is_servicing := false;
23                 atomic(s->nb_free--);
                   // Sleep and wait until this servicing thread is needed again
24                 cond_wait(&s->pcond, &s->plock);
25                 t->is_servicing := true;
26                 s->nb_servicing++;
27                 unlock(s->plock);
28             else
                   // There are other busy servicing threads
                   // Gives them a chance to run
29                 yield();
```

Algorithm 2: Executing a critical section

```
1  thread local variables:
2    integer th_client_index;
3    bool is_client;
4    server_t *my_server;
5  function execute_critical_section(lock_t *lock, function_t code, pointer_t context)
6    var request_t *
7        r := &lock->server->requests[th_client_index];
8    if is_client or my_server ≠ lock->server then
       // RCL to a remote core
9      r->lock := lock;
10     r->context := context;
11     r->code := code;
12     while r->code ≠ null do
13       ;
14     return;
15   else
       // Local nested lock, wait until the lock is free
16     while local_CAS(&lock->is_locked, false, true) = true do
17       yield();
       // Execute the critical section
18     code(context);
19     lock->is_locked := false;
20     return;
```

3 RCL and Legacy Applications

Implementing a set of locks using RCL requires assigning the locks to servers and reengineering the source code of the application. If multiple locks are assigned to a single server, their critical sections are effectively serialized. Because the number of servers is fixed, and should generally only be a small portion of the number of cores available on a machine, the developer must carefully choose which locks to implement using RCL. For this, we use the `Mutrace` profiler [22], and propose criteria for interpreting its results. If the results of `Mutrace` indicate that an application uses locks for which RCL is likely to be beneficial, the developer must reengineer the application, to extract each critical section into a separate function that can be passed to a remote server, and to replace the current locking operations by calls to the various functions of the RCL runtime. We have developed a tool that can automate this reengineering process.

3.1 Profiling

`Mutrace` is a dynamically loaded library that intercepts calls to the functions associated with various concurrency libraries. For each lock, `Mutrace` collects information about the properties of the execution of its associated critical sections, including lock contention. The resulting trace can be inspected by the application developer, to determine whether is it beneficial to use RCL in the given application.

Figure 4 shows an extract of the information obtained when profiling the SPLASH-2 applications. The first column identifies the application. The column *Lock* gives the lock name.[2] The column *Cont.* gives a measure of the contention

[2] Mutrace gives a lock number and an indication of where the lock was first accessed; we have replaced this with the lock name, for readability.

Algorithm 3: Management

```
1  function manager_thread(server_t *s)
2      activate_servicing_thread(s);
3      s->is_alive := true;
4      while true do
5          if s->is_alive = true then
6              s->is_alive := false;
7          else
8              s->is_alive := true;
               // Ensure that a thread can handle remote requests in case of a circular
                  path
9              activate_servicing_thread(s);
10             lock(s->plock);
11             var cur := s->threads;
               // Find a thread that has not recently been elected and elect it (≤ 2
                  iterations)
12             while true do
13                 while cur ≠ null do
14                     if cur->is_servicing = true
15                        and cur->recently_elected = false then
16                         cur->recently_elected = true;
17                         elect(cur);
18                         goto end;
19                     cur := cur->next;
                   // All the threads were elected, clear the recently_elected flags
20                 clear_all_recently_elected(s->threads);
21             end:
22             unlock(s->plock);
23         sleep(timeout);
24 function backup_thread(server_t *s)
25     while true do
26         activate_servicing_thread(s);
27 function activate_servicing_thread (server_t *s)
       // if we have no more free threads
28     if s->nb_free ≤ 0 then
           // take the lock
29         lock(s->plock);
30         if s->nb_servicing = s->nb_threads then
               // create a new thread if all threads are already servicing
31             var thread_t *t := new thread;
32             t->next = s->threads;
33             s->threads = t;
34             s->nb_threads++;
35             s->nb_servicing++;
36             t->is_servicing := true;
37             atomic(s->nb_free++);
38             unlock(s->plock);
39             t.start(prio_servicing);
40         else
               // wakeup one sleeping thread
41             atomic(s->nb_free++);
42             cond_signal(s->pcond);
43             unlock(s->plock);
```

on the lock, in terms of the number of times a thread tries to acquire the lock and finds that it is already held. The column *Cont rate* gives the result of dividing the contention by the execution time, giving an estimate of the rate of contention.

Given the information produced by the profiler, we have found the contention rate to be the most reliable indicator of the benefit of implementing a lock as an

Application	Lock	Cont.	**Cont. rate**
Barnes	CellLock	276	**5,498**
FMM	lock_array	484	**1,251**
Ocean Contiguous	error_lock	4,222	**6,200**
Ocean Non Contiguous	error_lock	4,196	**6,455**
Radiosity	q_lock	162,414	**85,107**
	free_interaction_lock	77,056	**40,379**
Raytrace Balls4	ridlock	36,152	**48,988**
	memlock	330	**447**
Raytrace Car	ridlock	24,265	**44,585**
	memlock	263	**483**
Volrend	IndexLock	544	**574**
Water-nsquared	KinetiSumLock	151	**3,715**
Water-spatial	KinetiSumLock	145	**3,508**

Figure 4: Profiling result extract for SPLASH-2

RCL. Figure 4 contains the information about the locks with the highest contention for each application. In our experiments, we have tried implementing each of these locks with an RCL. For our test architecture, a 48-core Opteron, when the contention rate is below $10,000$, changing the lock algorithm does not change the performance: when the contention is small, all lock algorithms are equivalent. When the contention rate is greater than $10,000$, implementing the lock as an RCL always gives a benefit on our hardware. Three locks are in this category: the `ridlock` of the Raytrace application and the `q_lock` and the `free_interaction_lock` of the Radiosity application. Our evaluation, presented in Section 4, shows that replacing these locks significantly improves the performance. The appropriateness of this threshold is confirmed by our results on the memcached application (Section 4.4).

3.2 Reengineering legacy applications

If the results of the profiling show that some locks used by the application can benefit from RCL, the developer must then reengineer each critical section that may be protected by the selected locks as a separate function that can be passed to the lock server. This reengineering is an instance of the "Extract Method" refactoring [10], that extracts a region of code into an independent function. We have implemented this reengineering using the program transformation tool Coccinelle [25], in 2115 lines of code. This reengineering has a negligible impact on performance (on average less than 0.04%).

The main problem in extracting a critical section into a separate function is to create the proper variable environment in which to execute the critical section code. Specifically, the extracted function must receive as arguments the values of the variables that are initialized prior to the critical section and read within the critical section, and must return as results the values of the variables that are updated within the critical section and read afterwards. By default, this information is passed to and from the server in an auxiliary structure. If there is at most one needed argument and at most one needed return value, then these values are instead passed via the client's cache line in the array of requests (Fig. 2), in place of the context value, to improve cache locality. It is also necessary to address the case of a critical section that contains more than one unlock operation, on different control flow paths, which complicates the identification of the end of the extracted function. In this case, the code is transformed such that

the critical section returns a different flag value for each unlock operation that may end the critical section, and then the code following the extracted critical section executes the code following the unlock that is indicated by the flag value.

To illustrate these issues, we consider a critical section from the SPLASH-2 benchmark `raytrace`, shown in Fig. 5. In this code, `ALOCK` acquires a lock from an array that is its first argument, at the index indicated by its second argument, and `AULOCK` releases a lock obtained similarly. The critical section references the variable `pid` (line 4), which is a parameter of the enclosing function, and updates the variable `wpentry` (line 4), which is used after the critical section. Space for these variables is allocated in a context structure that is specific to this critical section (Fig. 6, lines 1-4). The critical section furthermore contains a nested conditional that invokes `AULOCK` to release the lock before returning. In this case, the transformation essentially replaces the inner `AULOCK` by a jump to the final one, and moves the code following the inner `AULOCK` after the critical section, where a flag `ret` is used to determine whether it should be executed. The value of `ret` is returned from the extracted function. Our implementation currently only addresses the case where an inner conditional branch contains a single unlock operation, and the branch ends in return or exit. If these conditions are not met, the user is informed that the code should be transformed manually.

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

Figure 5: Critical section from `raytrace/workpool.C`

Fig. 6 shows the complete result of transforming the code of Fig. 5. The transformation furthermore modifies various other lock manipulation functions to use the RCL runtime. In particular, the function for initializing a thread receives additional arguments allowing the developer to indicate 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 initialization operations should create traditional locks and which ones should use RCLs.

4 Evaluation

We first use a custom microbenchmark to identify the cases in which critical sections execute faster when using RCL than when using traditional locks. We then evaluate the performance gain offered by RCL on the SPLASH-2 benchmark suite [30], and the legacy applications Memcached [21]. Table 1 summarizes

```
1  union instance {
2    struct input { INT pid; } input;
3    struct output { WPJOB *wpentry; } output;
4  };
5
6  void function(void *ctx) {
7    struct output *outcontext = &(((union instance *)ctx)->output);
8    struct input *incontext = &(((union instance *)ctx)->input);
9    WPJOB *wpentry;
10   int ret;
11   INT pid=incontext->pid;
12   ret = 0;
13   /* start of original critical section code */
14   wpentry = gm->workpool[pid][0];
15   if (!wpentry) {
16     gm->wpstat[pid][0] = WPS_EMPTY;
17     /* end of original critical section code */
18     ret = 1;
19     goto done;
20   }
21   gm->workpool[pid][0] = wpentry->next;
22   /* end of original critical section code */
23 done:
24   outcontext->wpentry = wpentry;
25   return (void *)(uintptr_t)ret;
26 }
27
28 INT GetJob(RAYJOB *job, INT pid) {
29   int ret;
30   ...
31   union instance instance = { pid, };
32
33   ret = liblock_exec(&gm->wplock[pid], &instance, &function);
34   wpentry = instance.output.wpentry;
35   if (ret) {
36     if (ret == 1)
37       return (WPS_EMPTY);
38   }
39   ...
40 }
```

Figure 6: Transformed critical section

the properties of these benchmarks. We first summarize the locking strategies considered and then present our results.

Benchmark	Cont.	Shared Data
Custom Micro-Benchmark	All	All
SPLASH2/Raytrace/car (3D Rend.)	High	Low
SPLASH2/Raytrace/balls4 (3D Rend.)	Medium	Low
SPLASH2/Radiosity (Graphics)	Medium	Low
Memcached/Get (On-memory cache)	Medium	Low
Memcached/Set (On-memory cache)	Medium	High

Table 1: Evaluations

All experiments were performed on a Dell PowerEdge R815 server having four 12-core Opteron-6172 2.1GHz CPUs (48 cores in total) and running Linux 2.6.35 (Ubuntu 10.10). All experiments were compiled with GCC 4.4.5 with -O3.

4.1 Locking strategies

We compare RCL with POSIX locks, traditional CAS spinlocks, MCS locks [20, 19] and flat combining locks [13]. The latter two are, to our knowledge, the best current locking algorithms targeting commodity hardware. We give a brief overview of these two algorithms in this section.

MCS locks use a lock-free algorithm to add a thread that is waiting for the lock to the end of a queue. While waiting, a thread spins on a memory location that is shared with the thread associated with the previous queue entry. The latter thread updates this memory location when it releases the lock. MCS thus achieves the responsiveness of spinlocks, but without the access contention induced by spinning on a location that is common to all threads. MCS, however, is subject to a convoy effect [5] when the owner of lock is preempted. He *et al.* have presented a refinement of MCS that avoids the convoy effect [12]. We have not used this algorithm because our benchmarks never use more threads than cores, and thus do not exhibit this phenomenon.

With flat combing [13], the owner of the lock acts temporarily as a server, called the *combiner*, and executes the pending critical sections for the other threads. To allow the owner of the lock to make progress, a new combiner is regularly elected, which involves a concurrent access to a single shared location by all of the threads that want to access the lock. Elections occur frequently at high contention, incurring overhead. Flat combining uses a linked list instead of an array of requests to manage the interaction between the clients and the combiner. The collection of unused elements in this list also incurs overhead. Finally, flat combining is unable to handle blocking within a critical section, because there is only one combiner, rather than a pool of servicing threads, as used in RCL.

We have implemented our own optimized versions of MCS and flat combining locks. In the case of MCS, we ensure that each thread allocates its node on the memory bank associated with its core in order to limit memory access costs. In the case of flat combining, the original implementation[3] focuses on optimizing the combination of operations on specific data structures. We have adapted this implementation to provide a generic interface as needed to support legacy applications. In our adaptation, we have optimized memory alignment to avoid useless cache misses and determined the best values for two key parameters: the number of requests that a combiner serves and the number of iterations that may elapse before an unused request entry is removed from the linked list.

4.2 Microbenchmarks

We have developed a custom microbenchmark in order to measure the performance of RCL relative to the other locks at varying degrees of contention, and with a varying number of shared memory locations being accessed in critical sections. We first measure the average critical section execution time, to evaluate when RCL is faster than other types of locks, and then measure the average number of cache misses per critical section, to understand when the performance gain is due to improved data locality.

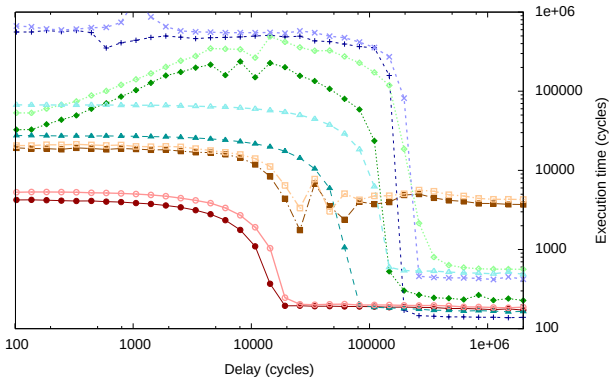
[3] https://sites.google.com/site/cconcurrencypackage/flat-combining-and-the-synchronization-parallelism-tradeoff

Our microbenchmark executes critical sections repeatedly. One core is dedicated to the RCL server; this core is left idle when benchmarking other types of locks. The remaining 47 cores each run a client, i.e., a thread that executes critical sections. Each client waits for a given delay between the end of one critical section and the beginning of the next one: the shorter the delay, the higher the contention. For each delay value, 1000 critical sections are executed. In each critical section, a client references and updates a given number of shared cache lines by incrementing the values in shared memory locations. These locations are scattered across memory in such a way that two such locations are never mapped to the same cache line: thus, to access n shared cache lines, the microbenchmark simply accesses n shared memory locations. In order to ensure that cache line accesses are not pipelined, we construct the address of the next memory location that is accessed using the value read from the current memory location [36].

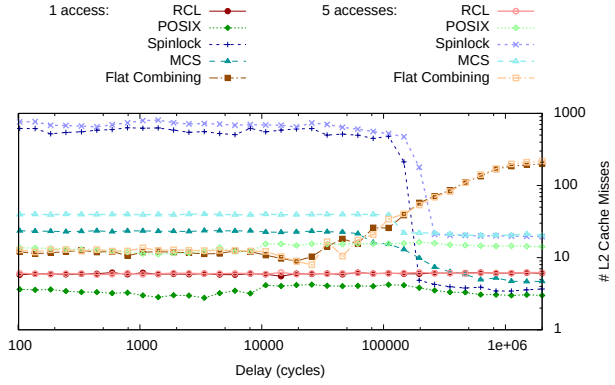
The critical section execution times are shown in Figure 7(a). Under high contention, when the critical section accesses only a single shared cache line (the left side of the graph), RCL is always faster than all the other considered types of locks. Flat combining is the best after RCL, but is still 2.5 times slower. MCS is slower than flat combining. The traditional spinlock is the slowest of all locks under high contention, due to the overhead of cache coherency messages when all threads spin on a compare-and-swap instruction. Finally, POSIX locks are as efficient as MCS locks under very high contention, but their execution time increases as contention decreases. Due to the fact that each critical section is executed locally, the performance of MCS, spinlocks and POSIX locks decreases significantly when 5 cache lines are accessed.

When contention is low (the right side of Figure 7(a)) and the critical section only accesses one cache line, spinlocks, MCS locks and RCL have similar performance. Spinlocks are best under low contention with a critical section execution time of 1260 cycles; RCL and MCS are the next best and are only 15% slower. However, when critical sections access 5 cache lines, the execution time of both traditional spinlocks and MCS locks increases significantly, whereas that of RCL remains stable. This is due to the fact that all critical sections are executed on the same core, thus improving cache locality. The execution time of flat combining also remains stable when the number of memory accesses increases, but it is more than ten times higher than that of RCL. POSIX locks perform better than flat combining but not as well as spinlocks, MCS and RCL.

Figure 7(b) shows the number of L2 cache misses per critical section for each lock. The execution time of each lock is directly correlated with its number of cache misses, except for the POSIX locks, whose overhead is mainly due to the high cost of context switches. Even though the number of cache misses increases as the contention increases for both spinlocks and MCS locks, it remains stable for RCL, which shows how well RCL suited is for highly-contended locks. The number of cache misses when using flat combining increases as the contention decreases, which directly reflects the high execution time of flat combining under low contention. These cache misses are caused by the server when it scans the linked list of requests: accessing each element of this linked list typically incurs a cache miss. This scan also occurs at high contention, but several critical sections are also executed simultaneously, while at low contention, only one critical section is executed for the same number of cache misses.



(a) Critical section execution time



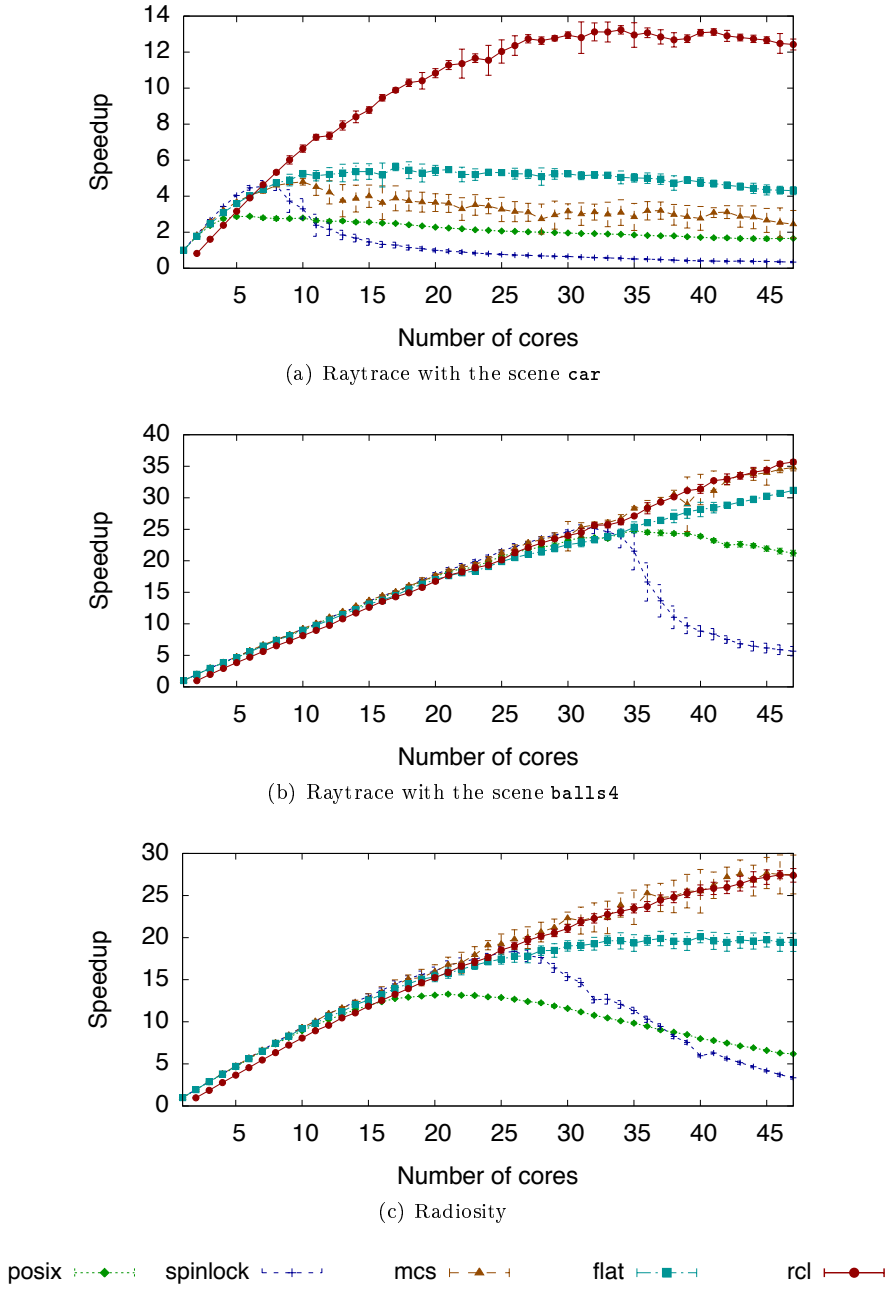
(b) Cache misses

Figure 7: Microbenchmark results. Each data point is the average of 30 runs. y-axis: cycles per critical section for (a) and # of cache misses per critical section for (b).

4.3 SPLASH-2

SPLASH-2 is a classic benchmark suite consisting of a number of legacy multithreaded applications and application kernels. It has been previously used to evaluate locking algorithms [12, 15]. We use SPLASH-2 both to evaluate our reengineering tool and to evaluate the accuracy of our profiler and the performance of RCL. The nine applications and five application kernels included in the SPLASH-2 benchmark suite contain a total of 137 critical sections. All of these critical sections are correctly transformed by our reengineering tool.

As presented in Section 3.1, among all the locks used in SPLASH-2, we have identified only three as having high enough contention to be interesting candidates for RCL: `q_lock` and `free_interaction_lock` for the application Radiosity and `ridlock` for the application Raytrace. For these two applications, we compare the base application that uses POSIX locks (before reengineering) with four modified versions of the application in which the identified locks are implemented as traditional spinlocks, MCS locks, flat combining locks, and RCLs, respectively. For Raytrace, we have also slightly modified the code to solve a false sharing issue for the server-based algorithms flat combining and RCL. The reengineering tool identifies, as a side effect, which potentially shared



(a) Raytrace with the scene `car`

(b) Raytrace with the scene `balls4`

(c) Radiosity

Figure 8: SPLASH-2 results. Each data point is the average of 30 runs.

variables are accessed by each critical section. In one case, one field of a structure is both read and written by the critical section while the other fields are only accessed by the clients. Adding padding that is the size of a cache line around this field has no impact when using POSIX locks, spinlocks, or MCS, but gives a significant benefit for flat combining and RCL, of up to 33% in the

latter case for the high contention scene `car`.

For Raytrace, only one lock is implemented using RCL, and thus only one server is needed. Radiosity, however, creates as many instances of the `q_lock` as the number of threads. Nevertheless, with our current implementation of RCL, we can only select the type of a lock at its initialization site and thus all of the instances of `q_lock` are allocated to a single server core. We could allocate the `free_interaction_lock` to a different server core, but we have found that this gives no performance benefit, and thus we use only one server core in this case as well.

All versions of Raytrace were tested with the benchmark scenes `car` and `balls4` provided with SPLASH-2. The scene `car` incurs higher contention rate than the scene `balls4`, due to the fact that it uses a deeper hierarchy to store its objects. Radiosity was tested with the default input provided for fewer than 64 cores. The results are shown in Figure 8. The x axis gives the total number of cores used: for RCL, N means $N-1$ application threads and one server; for the other locking algorithms, N means N application threads.

On `car`, RCL reaches a maximal speedup of 13.24 with 34 cores while flat combining reach a maximal speedup of 5.63 at 17 cores. The maximal speedup of other locks remains below 5. We have measured the delay between the end of one critical section and the beginning of the next and found that the average delay is $9,240$ cycles. As presented in Figure 7(a), RCL is the only algorithm that scales at this contention rate. On `balls4`, RCL reaches a maximal speedup of 35.61 with 48 cores while MCS reaches a maximal speedup of 34.45 with 48 cores, flat combing reaches a maximal speedup of 30.63 with 48 cores and the other locks remain below a maximal speedup of 26. The average delay between two critical sections is $74,790$ cycles, which is just before the point at which we previously found that MCS starts to perform better than flat combining (Figure 7(a)). Finally, with Radiosity, RCL reaches a maximal speedup of 27.46 with 46 cores, MCS reaches a maximal speedup of 27.66 at 45 cores, while the other locks remain below a maximal speedup of 20.5. The delay between two critical sections is on average $74,648$ cycles at which point only MCS and RCL scale. In summary, in the case of medium contention, as found in Radiosity and in Raytrace on `balls4`, RCL is comparable or slightly better than MCS and clearly better than the remaining types of locks. In the case of high contention, as found in Raytrace on the scene `car`, RCL scales better than all the other approaches.

4.4 Memcached

Memcached [21] is a legacy cache server that is widely used in Web infrastructures that need to support a high load. The lock that we have implemented as an RCL is the "cache lock," that exhibits a high contention rate (see Figure 9(c)). This lock protects all of the accesses to the main hash table when memcached is executed in multi-threaded mode. Memcached contains in total 15 critical sections and these are all correctly transformed by our reengineering tool.

The tests were carried out using memcached 1.4.6 and the library libmemcached 0.49 [18]. Libmemcached provides a client, memslap, which makes it possible to simulate a realistic load on an instance of a memcached server. In our experiments, memslap and memcached execute in parallel on our test ma-

chine, with 24 cores reserved for the execution of memslap and the remaining 24 cores reserved for memcached. This configuration is sufficient to saturate memcached up to 24 cores, regardless of the type of lock used. Memcached is executed in multithreaded mode, using the default parameters. There are only 21 usable cores for worker threads in memcached, because one core is reserved for the RCL server (in case of RCL), and two cores are reserved for the two memcached threads that treat incoming requests and that manage the maintenance of the hash table (increasing or decreasing its size, for example), respectively. Memslap uses one base thread, which has its own core, and 460 threads pinned on the 23 remaining cores for sending requests.

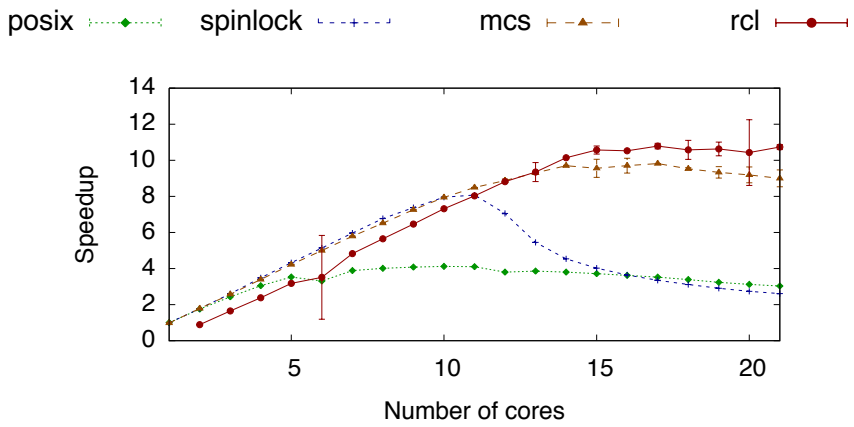
We compare the execution of the original memcached that uses POSIX locks with the version of memcached adapted by our engineering tool where the "cache lock" is replaced respectively by MCS locks, spinlocks and RCL. As for SPLASH-2, N cores for RCL means $N-1$ cores for the application and 1 core for the RCL server. N cores are used for the application for the other locking algorithms. The synchronization between the thread that manages the hash table and the worker threads relies on a condition variable. In particular, the thread that manages the hash table blocks in a critical section while waiting on this condition variable throughout the execution, except when the hash table needs to be resized. This benchmark therefore periodically activates the RCL mechanisms that permit to manage blocking threads. As flat combining does not support any form of blocking within a critical section, we were not able to test it on this application.

Figure 9 shows the performance of memcached with two configurations. In the first configuration (Figure 9(a)), 10,000 elements are first added to the cache and then clients perform only GET requests. All the locks benefit from a good locality as the data accessed by the critical sections are only read. RCL reaches a maximal speedup of 10.79 at 17 cores, MCS reaches a maximal speedup of 9.8 at 17 cores and POSIX locks and spinlocks collapse after 8 cores with a maximal speedup of 8.07, in the case of spinlocks. In the second configuration (Figure 9(b)), clients only add new elements to the cache. In this case, cache locality is an important factor for scaling up; with RCL the written data stay on the server, but with other locks, the data bounce between the cores that own the lock. RCL scales up to 21 cores with a speedup of 4.73, spinlocks scale to 6 cores with a maximal speedup of 2.96, and MCS scales up to 9 cores with a maximal speedup of 2.65. In both GET and SET configurations, when the number of cores increases, RCL has the best performance.

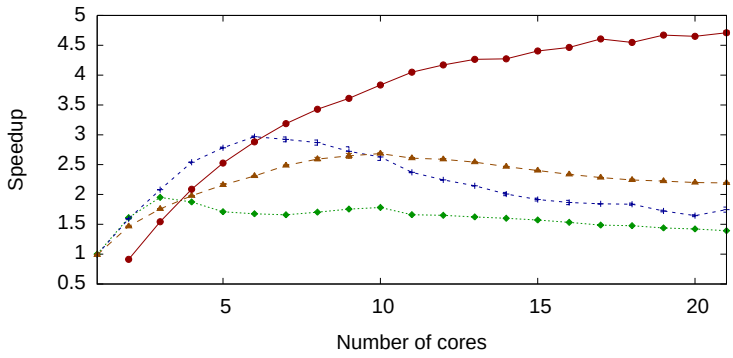
5 Related Work

Many approaches have been proposed to improve locking [1, 8, 13, 15, 28, 31, 33]. Some solutions [8, 28] improve the fairness of locking algorithms or the load they induce on the data bus. Other works switch automatically between blocking locks and spinlocks depending on the contention level [15]. Others, like RCL, address data locality [13, 33].

Suleman *et al.* [33] propose to transform critical sections into remote procedure calls to a powerful server core on an asymmetric multicore. Their communication protocol is implemented in hardware and requires a modified processor. They do not address blocking within critical sections (*cf.*, Section 2.2), relying on static analysis to ensure that critical sections that may be nested within another critical section are never executed on the server. This can be a prob-



(a) GET requests.



(b) SET requests

Experiment	Lock	Cont.	**Cont. rate**
Get	cache lock	5470860	**75690**
Set	cache lock	311349	**29295**

(c) profiling result extract

Figure 9: Performance of memcached, using different types of locks for the "cache lock". Each data point is the average of 5 executions.

lem for legacy applications or libraries, which may block on condition variables. RCL works on legacy hardware and allows blocking within critical sections. Flat combining [13], temporarily transforms the owner of a lock into a server for other critical sections. At low contention, flat combining is not efficient because the server, which could be executing its own code, has to uselessly verify whether pending requests exist, which increases the latency of a critical section. In RCL, the server may also uselessly scan the array of pending requests, but as the server has no other code to execute, this does not incur any overall delay.

Sridharan *et al.* [31] associate an affinity between a core and a lock to increase data locality. They calculate this affinity by intercepting Futex [9] operations in Linux and modify the Linux scheduler to try to schedule the lock requester

to the preferred core of the lock. This technique, however, does not address the access contention that occurs when several cores try to enter their critical sections.

GLocks [1] address the problem of latency due to cache misses of highly-contended locks at the hardware level by building a token-ring between cores. When a core receives the token, if it has a pending critical section, it serves the critical section and then forwards the token to the next core, otherwise it simply forwards the token. GLocks improves the performance of some of the SPLASH-2 benchmarks compared to MCS, principally with Raytrace. GLocks, however, only uses one token and therefore can be used to implement only one lock. It furthermore only addresses access contention and not cache locality. Roy *et al.* [27] have proposed a profiling tool to identify critical sections that work on disjoint data sets, in order to optimize them by increasing parallelism. This approach is complementary to ours, since one of the objectives of RCL is to improve the performance of critical sections that intensively access shared data.

Lock-free structures have been proposed in order to avoid using locks for traditional data structures such as counters, linked lists, stacks, or hashtables [14, 17, 29]. These approaches never block threads. However, such techniques are only applicable to the specific types of data structures considered. For this reason, locks are still commonly used on multicore architectures.

Finally, experimental operating systems designed with manycore architectures in mind use data replication to improve locality [34] and even RPC-like mechanisms to access shared data from remote cores [4, 23, 11, 26, 16]. These solutions, however, require a complete overhaul of the operating system or database design. RCL, on the other hand, can be used with current systems and applications with few modifications.

6 Conclusion

RCL is 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 migrate critical section execution to a server core. We have implemented an RCL runtime for Linux that supports a mixture of RCL and POSIX locks in a single application. To ease the reengineering of legacy applications, we have designed a profiling-based methodology for detecting highly contended locks and implemented a tool that transforms critical sections into remote procedure calls. Our performance evaluations on a legacy benchmark and a legacy application show that RCL improves performance when an application relies on highly contended locks.

In future work, we will consider the design and implementation of an adaptive RCL runtime. Our first goal will be to be able to dynamically switch between locking strategies, so as to dedicate a server core only when a lock is contented. Secondly, we want to be able to migrate an RCL server to another core, when multiple servers are available. One of the challenges in doing that will be the implementation of low-overhead run-time profiling strategies.

Availability The implementation of RCL as well as our test scripts and results are available at `http://rclrepository.gforge.inria.fr`.

References

[1] J. L. Abellán, J. Fernández, and M. E. Acacio. Glocks: Efficient support for highly-contended locks in many-core cmps. In *25th IPDPS*. IEEE, 2011.

[2] A. Agarwal and M. Cherian. Adaptive backoff synchronization techniques. In *ISCA'89*, pages 396–406. ACM, 1989.

[3] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: featherweight synchronization for Java. In *PLDI'98*, pages 258–268. ACM, 1998.

[4] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schuepbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *SOSP'09*, pages 29–44. ACM, 2009.

[5] M. Blasgen, J. Gray, M. Mitoma, and T. Price. The convoy phenomenon. *SIGOPS Oper. Syst. Rev.*, 13:20–25, April 1979.

[6] L. Boguslavsky, K. Harzallah, A. Kreinen, K. Sevcik, and A. Vainshtein. Optimal strategies for spinning and blocking. *Journal of Parallel and Distributed Computing*, 1993.

[7] A. Brodsky, F. Ellen, and P. Woelfel. Fully-adaptive algorithms for long-lived renaming. In *DISC '06*, pages 413–427, 2006.

[8] T. S. Craig. Building FIFO and priority-queueing spin locks from atomic swap. Technical Report TR 93-02-02, Department of Computer Science, University of Washington, Feb. 2003.

[9] U. Drepper and I. Molnar. Native POSIX thread library for Linux. Technical report, RedHat, Inc, 2003.

[10] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[11] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *OSDI'99*, pages 87–100.

[12] B. He, W. N. Scherer III, and M. L. Scott. Preemption adaptivity in time-published queue-based spin locks. In *Proceedings of the 11th International Conference on High Performance Computing*, 2005.

[13] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *SPAA'10*, pages 355–364, 2010.

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

[15] F. R. Johnson, R. Stoica, A. Ailamaki, and T. C. Mowry. Decoupling contention management from scheduling. In *ASPLOS'10*, pages 117–128, 2010.

[16] E. M. C. Jr., P. Das, T. J. LeBlanc, B. D. Marsh, and M. L. Scott. Kernel-kernel communication in a shared-memory multiprocessor. *Concurrency - Practice and Experience*, 5(3):171–191, 1993.

[17] E. Ladan-Mozes and N. Shavit. An optimistic approach to lock-free FIFO queues. *Distributed Computing*, 20(5):323–341, 2008.

[18] Libmemcached. `https://launchpad.net/libmemcached`.

[19] J. M. Mellor-Crummey and M. L. Scott. Synchronization without contention. In *ASPLOS'91*, pages 269–278. ACM.

[20] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM TOCS*, 9(1):21–65, Feb. 1991.

[21] `http://memcached.org`.

[22] `http://0pointer.de/blog/projects/mutrace.html`.

[23] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: heterogeneous multiprocessing with satellite kernels. In *SOSP'09*, pages 221–234, 2009.

[24] J. K. Ousterhout. Scheduling techniques for concurrebt systems. In *ICDCS '82*, pages 22–30, 1982.

[25] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in Linux device drivers. In *EuroSys'08*, pages 247–260.

[26] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *Proceedings of the VLDB Endowment (PVLDB)*, 3(1):928–939, 2010.

[27] A. Roy, S. Hand, and T. Harris. Exploring the limits of disjoint access parallelism. HotPar'09, pages 8–8, 2009.

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

[29] O. Shalev and N. Shavit. Split-ordered lists: Lock-free extensible hash tables. *JACM*, 53(3):379–405, May 2006.

[30] Modified SPLASH-2 (Stanford parallel applications for shared memory 2). `http://www.capsl.udel.edu/splash`.

[31] S. Sridharan, B. Keck, R. Murphy, S. Chandra, and P. Kogge. Thread migration to improve synchronization performance. In *In Workshop on Operating System Interference in High Performance Applications*, 2006.

[32] M. A. Suleman, O. Mutlu, J. A. Joao, Khubaib, and Y. N. Patt. Data marshaling for multi-core architectures. In *ISCA '10*, pages 441–450. ACM, 2010.

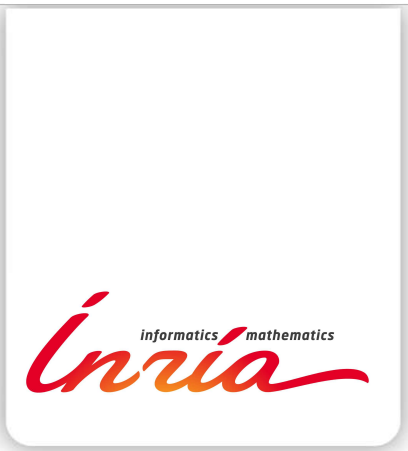
[33] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *ASPLOS'09*, pages 253–264.

[34] S. B. Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: an operating system for many cores. In *OSDI'08*. USENIX Association, 2008.

[35] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma. Ad hoc synchronization considered harmful. In *OSDI'10*, pages 1–8. USENIX Association, 2010.

[36] K. Yotov, K. Pingali, and P. Stodghill. Automatic measurement of memory hierarchy parameters. In *SIGMETRICS '05*, pages 181–192. ACM, 2005.

Contents



RESEARCH CENTRE
PARIS – ROCQUENCOURT

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399