Remote Core Locking: Migrating Critical-Section Execution to Improve the Performance of Multithreaded Applications

Jean-Pierre Lozi Florian David Gaël Thomas Julia Lawall Gilles Muller
LIP6/INRIA
firstname.lastname@lip6.fr

Abstract

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. In this paper, we propose a new lock algorithm, 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 other lock algorithms 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 because such data can typically remain in the server core's cache.

We have developed 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 locks into RCL locks. We have evaluated our approach on 18 applications: Memcached, Berkeley DB, the 9 applications of the SPLASH-2 benchmark suite and the 7 applications of the Phoenix2 benchmark suite. 10 of these applications, including Memcached and Berkeley DB, are unable to scale because of locks, and benefit from RCL. Using RCL locks, we get performance improvements of up to 2.6 times with respect to POSIX locks on Memcached, and up to 14 times with respect to Berkeley DB.

1 Introduction

Over the last twenty years, a number of studies [2, 3, 5, 12, 13, 15, 17, 24, 26, 27] 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 simultaneously try to enter critical sections that are protected by the same lock, causing the cache line containing 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 remains a challenge. These issues imply that some applications that work well on a small number of cores do not scale to the number of cores found in today's multicore architectures.

Recently, several approaches have been proposed to execute a succession of critical sections on a single *server* core to improve cache locality [13, 27]. Such approaches also incorporate a fast transfer of control from other *client* cores to the server, to reduce access contention. Suleman *et al.* [27] propose a hardware-based solution, evaluated in simulation, that introduces new instructions to perform the transfer of control, and uses a special fast core to execute 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 drastically degrades performance at low contention. Furthermore, neither Suleman *et al.*'s algorithm nor Hender *et al.*'s algorithm can accommodate threads that block within a critical section, which makes them unable to support widely used applications such as Memcached.

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 one or several dedicated servers, 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 as well as nested 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 much lower overall overhead.

We propose a methodology along with a set of tools to facilitate the use of RCL in a legacy application. Because RCL serializes critical sections associated with locks managed by the same core, transforming all locks into RCLs on a smaller number of servers could induce false serialization. Therefore, the programmer must first decide which locks should be transformed into RCLs and on which core(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 must be transformed into RCLs. We also have designed an automatic reengineering tool for C programs to transform the code of critical sections so that it can be executed as a remote procedure call on the server core: the code within the critical sections must be extracted as functions and variables referenced or updated by the critical section that are declared by the function containing the critical section code must be sent as arguments, amounting to a context, to the server core. Finally, we have developed a runtime for Linux that is compatible with POSIX threads, and that supports a mixture of RCL and POSIX locks in a single application.

RCL is well-suited to improve the performance of a legacy application in which contended locks are an obstacle to performance, since using RCL enables improving locality and resistance to contention without requiring a deep understanding of the source code. On the other hand, modifying locking schemes to use fine-grained locking or lock-free data structures is complex, requires an overhaul of the source code, and does not improve locality.

We have evaluated the performance of RCL as compared to other locks 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 our profiler, we have identified Memcached, Berkeley DB with two types of TPC-C transactions, two benchmarks in the SPLASH-2 suite, and three benchmarks in the Phoenix2 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 [17] and Flat Combining [13]. Comparisons are made for a same number of cores, which means that there are fewer application threads in the RCL case, since one or more cores are dedicated to RCL servers.

On an Opteron 6172 48-core machine running a 3.0.0 Linux kernel with glibc 2.13, our main results are:

- On our latency microbenchmark, under high contention, RCL is faster than all the other tested approaches, over 3 times faster than the second best approach, Flat Combining, and 4.4 faster than POSIX.

- On our benchmarks, we found that contexts are small, and thus the need to pass a context to the server has only a marginal performance impact.

- On most of our benchmarks, only one lock is frequently used and therefore only one RCL server is needed. The only exception is Berkeley DB which requires two RCL servers to prevent false serialisation.

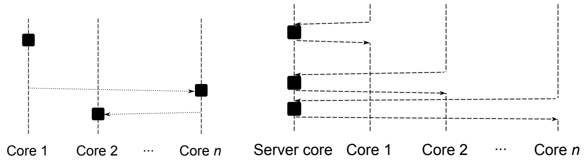
- On Memcached, for Set requests, RCL provides a speedup of up to 2.6 times over POSIX locks, 2.0 times over MCS and 1.9 times over spinlocks.

- For TPC-C Stock Level transactions, RCL provides a speedup of up to 14 times over the original Berkeley DB locks for 40 simultaneous clients and outperforms all other locks for more than 10 clients. Overall, RCL resists better when the number of simultaneous clients increases.

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

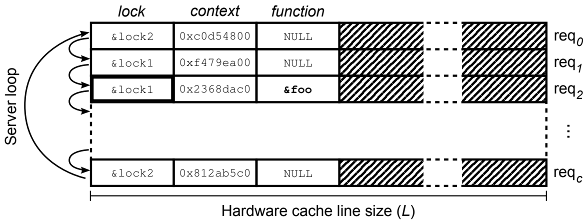
2 RCL Overview

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 (Fig. 1). To use this approach, it is necessary to choose the locks for which RCL is expected to be beneficial and to reengineer the critical sections associated with these locks as remote procedure calls. In this section, we first describe the core RCL algorithm, then present a profiling tool to help the developer choose which locks



(a) POSIX locks (b) RCL

Fig. 1: Critical sections with POSIX locks vs. RCL.



lock	context	function	
&lock2	0xc0d54800	NULL	req_0
&lock1	0xf479ea00	NULL	req_1
&lock1	0x2368dac0	**&foo**	req_2
			⋮
&lock2	0x812ab5c0	NULL	req_c

Fig. 2: The request array. Client c_2 has requested execution of the critical section implemented by `foo`.

to implement using RCL and a reengineering tool that rewrites the associated critical sections.

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 (Fig. 2). This array has size $C \cdot L$ bytes, where C is a constant representing the maximum number of allowed clients (a large number, typically much higher than the number of cores), and L is the size of the hardware cache line. Each request structure req_i is L bytes and allows communication between a specific client i and the server. The array is aligned so that each structure 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 the lock associated with the critical section, (ii) the address of a structure encapsulating 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 a function that encapsulates the critical section for which the client c_i has requested the execution, or `NULL` if no critical section is requested.

Client side To execute a critical section, a client c_i first writes the address of the lock into the first word of the 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 actively waits for the third word of req_i to be reset to `NULL`, indicating that the server has executed the critical section. In order to improve energy efficiency, if there are less clients than the number of cores available, the SSE3 `monitor/mwait` instructions can be used to avoid spinning: the client will sleep and be woken up automatically when the server writes into the third word of req_i.

Server side A servicing thread iterates over the requests, waiting for one of the requests to contain a non-`NULL` value in its third word. When such a 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 third word to `NULL`, and resumes the iteration.

2.2 Profiling

To help the user decide which locks to transform into RCLs, we have designed a profiler that is implemented as a dynamically loaded library and intercepts calls involving POSIX locks, condition variables, and threads. The profiler returns information about the overall percentage of time spent in critical sections, as well as about the percentage of time spent in critical sections for each lock. We define the time spent in a critical section as the total time to acquire the lock (blocking time included), execute the critical section itself, and release the lock. It is measured by reading the cycle counter before and after each critical section, and by comparing the total measured time in critical sections with the total execution time, for each thread. The overall percentage of time spent in critical sections can help identify applications for which using RCL may be beneficial, and the percentage of time spent in critical sections for each lock helps guide the choice of which locks to transform into RCLs. For each lock, the profiler also produces information about the number of cache misses in its critical sections, as these may be reduced by the improved locality of RCL.

Fig. 3 shows the profiling results for 18 applications, including Memcached v1.4.6 (an in-memory cache server), Berkeley DB v5.2.28 (a general-purpose database), the 9 applications of the SPLASH-2 benchmark suite (parallel scientific applications), and the 7 applications of the Phoenix v2.0.0 benchmark suite (MapReduce-based applications) with the "medium" dataset.[1] Raytrace and Memcached are each tested with two different standard working sets, and Berkeley DB is tested with the 5 standard transaction types from TPC-C. A gray box indicates

[1] More information about these applications can be found at the following URLs: `http://memcached.org` (Memcached), `http://www.oracle.com/technetwork/database/berkeleydb` (Berkeley DB), `http://www.capsl.udel.edu/splash` (SPLASH-2) and `http://mapreduce.stanford.edu` (Phoenix2).

that the application has not been run for the number of cores because even at 48 cores, locking is not a problem.

Ten of the tests spend more than 20% of their time in critical sections and thus are candidates for RCL. Indeed, for these tests, the percentage of time spent in critical sections directly depends on the number of cores, indicating that the POSIX locks are one of the main bottlenecks of these applications. We see in Sec. 4.1 that if the percentage of time executing critical sections for a given lock is over 20%, then an RCL will perform better than a POSIX lock, and if it is over 70%, then an RCL will perform better than all other known lock algorithms. We also observe that the critical sections of Memcached/Set incur many cache misses. Finally, Berkeley DB uses hybrid Test-And-Set/POSIX locks, which causes the profiler to underestimate the time spent in critical sections.

2.3 Reengineering legacy applications

If the results of the profiling 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 lock server. This reengineering amounts to an "Extract Method" refactoring [10]. We have implemented this reengineering using the program transformation tool Coccinelle [21], in 2115 lines of code. It has a negligible impact on performance.

The main problem in extracting a critical section into a separate function is to bind the variables needed 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 the critical section, and return the values of variables that are updated in the critical section and read afterwards. Only variables local to the function are concerned; alias analysis is not required because aliases involve addresses that can be referenced from the server. The values are passed to and from the server in an auxiliary structure, or directly in the client's cache line in the request array (Fig. 2) if only one value is required. The reengineering also addresses a common case of fine-grained locking, illustrated in lines 5-9 of Fig. 4, where a conditional 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 unlock ends the critical section, and then the code following the remote procedure call executes the code following the unlock that is indicated by the flag value.

Fig. 5 shows the complete result of transforming the code of Fig. 4. The transformation furthermore modifies various other lock manipulation 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.

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

Fig. 4: Critical section from `raytrace/workpool.c`.

```
union instance {
  struct input { INT pid; } input;
  struct output { WPJOB *wpentry; } output;
};

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

Fig. 5: Transformed critical section.

3 Implementation of the RCL Runtime

Legacy applications may use ad-hoc synchronization mechanisms and rely on libraries that themselves may block or spin. The core algorithm, of Sec. 2.1, only refers to a single servicing thread, and thus requires that this thread is never blocked at the OS level and never spins in an active waitloop. In this section, we describe how the RCL runtime ensures liveness and responsiveness in these cases, and present implementation details.

Application		% in CS = f(# cores)						Lock usage for # max. core			
		1	4	8	16	32	48	Description	#	# L2 cache misses/CS	% in CS
SPLASH-2	Radiosity	4.3%	8.8%	15.6%	43%	79.3%	84.5%	Linked list access	1	1.6	82.0 %
	Raytrace Balls4	0.5%	1.3%	1.9%	3.3%	17%	40.1%	Counter increment	1	1.3	32.32 %
	Raytrace Car	12.2%	25.9%	51.4%	71.9%	85.5%	87.6%	Counter increment	1	0.6	79.95 %
	Barnes						3.1%				
	FMM						5.0%				
	Ocean Cont.					0.3%†					
	Ocean Non Cont.					0.3%†					
	Volrend						6.8%				
	Water-nsquared						3.6%				
	Water-spatial						0.5%				
Phoenix 2	Linear Regression	0.9%	25.2%	43.7%	66.9%	60.8%	83.6%	Task queue access	1	4.0	83.6%
	String Match	0.1%	4.7%	11.7%	24.2%	35.0%	63.4%	Task queue access	1	3.9	63.4%
	Matrix Multiply	0.9%	26.2%	45.9%	64.8%	79.2%	92.7%	Task queue access	1	3.4	92.7%
	Histogram						12.7%				
	PCA						11.6%				
	KMeans						1.5%				
	Word Count						3.2%				
Memcached	Get	6.7%	30.2%	50.1%	76.3%	22 cores: 84.9%‡		Hashtable access	1	3.6	84.7%
	Set	4.8%	28.7%	44.6%	54.4%	22 cores: 55.4%‡		Hashtable access	1	32.7	55.3%
Berkeley DB with TPC-C	Payment						5.80%				
	New Order						1.55%				
	Order Status	0.8%	0.8%	0.3%	2.0%	35.8%	52.9%	DB struct. access	11	4.2	52.9%
	Delivery						1.58%				
	Stock Level	0.0%	0.4%	0.2%	2.2%	0.4%	55.5%	DB struct. access	11	3.2	55.5%

† Number of cores must be a power of 2. ‡ Other cores are executing clients.

Fig. 3: Profiling results for the evaluated applications.

3.1 Ensuring liveness and responsiveness

Three sorts of situations may induce liveness or responsiveness problems. First, the servicing thread could be blocked at the OS level, e.g., because a critical section tries to acquire a POSIX lock that is already held, performs an I/O, or waits on a condition variable. Indeed, we have found that roughly half of the multithreaded applications that use POSIX locks in Debian 6.0.3 (October 2011) also use condition variables. Second, the servicing thread could spin if the critical section tries to acquire a spinlock or a nested RCL, or implements some form of ad hoc synchronization [29]. Finally, a thread could be preempted at the OS level when its timeslice expires [20], or because of a page fault. Blocking and waiting within a critical section risk deadlock, because the server 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.

Ensuring liveness To ensure liveness, the RCL runtime manages a pool of threads on each server such 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 this servicing thread will eventually be elected. 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* (we use the Linux time-slice value) and runs at highest priority. When activated, the management thread checks that at least one of the servicing threads has made progress since its last activation, using a server-global flag *is_alive* that is set by the servicing threads. If it finds that this flag is still cleared when it wakes up, it assumes that all servicing threads are either blocked or waiting and adds a free thread to the pool of servicing threads.

Ensuring responsiveness The RCL runtime implements a number of strategies to improve responsiveness. First, it avoids thread preemption from the OS scheduler by using the POSIX FIFO scheduling policy, which allows a thread to execute until it blocks or yields the processor. Second, it tries to reduce the delay before an unblocked servicing thread is rescheduled by minimizing the number of servicing threads, thus minimizing the length of the FIFO queue. Accordingly, a servicing thread suspends when it observes that there is at least one other free servicing thread, i.e., one other thread able to handle requests. Third, to address the case where all servicing threads are blocked in the OS, the RCL runtime uses a *backup thread*, which has lower priority than all servicing threads, that simply clears the *is_alive* flag and wakes up the management thread. 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, to allow the owner of the lock to release it.

The use of the FIFO policy raises two further liveness issues. First, FIFO scheduling may induce a priority inversion between the backup thread and the servicing threads or between the servicing threads and the management thread. To avoid this problem, the RCL runtime uses only lock-free algorithms and threads never wait on a resource. Second, if a servicing thread is in an active wait loop, it will not be preempted, and a free thread will not be elected. When the management thread detects no progress, i.e., *is_alive* is false, it thus also acts as a scheduler, electing a servicing thread by first decrementing and then incrementing the priorities of all the servicing threads, effectively moving them to the end of the FIFO queue. This is expensive, but is only needed when a thread spins for a long time, which is a sign of poor programming, and is not triggered in our benchmarks.

3.2 Algorithm details

We now describe in detail some issues of the algorithm.

Serving RCLs Alg. 1 shows the code executed by a servicing thread. The *fast path* (lines 6-16) is the only code that is executed when there is only one servicing thread in the pool. A *slow path* (lines 17-24), is additionally executed when there are multiple servicing threads.

Lines 9-15 of the fast path implement the RCL server loop as described in Sec. 2.1 and indicates that the servicing thread is not free by decrementing (line 8) and incrementing (line 16) *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.[2] To avoid the need to reallocate the request array when new client threads are created, the size of the request array is fixed and chosen to be very large (256K), and the client identifier allocator implements an adaptive long-lived renaming algorithm [6] that keeps track of the highest client identifier and tries to reallocate smaller ones.

The slow path is executed if the active servicing thread detects that other servicing threads exist (line 17). If the other servicing threads are all executing critical sections (line 18), the servicing thread simply yields the processor (line 19). Otherwise, it goes to sleep (lines 21-24).

Executing a critical section A client that tries to acquire an RCL or a servicing thread that tries to acquire an RCL managed by another core submits a request and waits for the function pointer to be cleared (Alg. 2, lines 6-9). If the RCL is managed by the same core, the servicing thread must actively wait until the lock is free. During this time it repetitively yields, to give the CPU to the thread that owns the lock (lines 11-12).

Management and backup threads If, on wake up, the management thread observes, based on the value of *is_alive*, that none of the servicing threads has progressed since the previous timeout, then it ensures that at least one free thread exists (Alg. 3, lines 8-19) and forces the election (lines 20-27) of a thread that has not been recently elected. The backup thread (lines 31-34) simply sets *is_alive* to false and wakes up the management thread.

Algorithm 1: Structures and server thread

```
1  structures:
     lock_t:      { server_t* server, bool is_locked };
     request_t:   { function_t* code, void* context, lock_t* lock };
     thread_t:    { server_t* server, int timestamp, bool is_servicing };
     server_t:    { List<thread_t*> all_threads,
                  LockFreeStack<thread_t*> prepared_threads,
                  int number_of_free_threads, number_of_servicing_threads,
                  int timestamp, boolean is_alive, request_t* requests }
2  global variables:    int number_of_clients;
3  function rcl_servicing_thread(thread_t* t)
4      var server_t* s := t->server;
5      while true do
6          s->is_alive := true;
7          t->timestamp := s->timestamp;
           // This thread is not free anymore.
8          local_atomic(s->number_of_free_threads--);
9          for i := 0 to number_of_clients do
10             r := s->requests[i];
11             if r->code ≠ null and
12                 local_CAS(&r->lock->is_locked,false,true) = false
               then
                   // Execute the critical section
13                 r->code(r->context);
                   // Indicate client execution completion
14                 r->code := null;
15                 r->lock->is_locked := false;
           // This thread is now free
16         local_atomic(s->number_of_free_threads++);
           // More than one servicing thread means blocked threads exist
17         if s->number_of_servicing_threads > 1 then
18             if s->number_of_free_threads <= 1 then
19                 yield(); // Allow other busy servicing threads to run
20             else
                   // Keep only one free servicing thread
21                 t->is_servicing := false;
22                 local_atomic(s->number_of_servicing_threads--);
23                 local_atomic_insert(s->prepared_threads,t);
                   // Must be atomic because the manager could wake
                     up the thread before the sleep (use futex).
24                 atomic(if not t->is_servicing then sleep)
```

4 Evaluation

We first present a microbenchmark that identifies when critical sections execute faster with RCL than with the other lock algorithms. We then correlate this information with the results of the profiler, so that a developer can use the profiler to identify which locks to transform into RCLs. Finally, we analyze the performance of the applications identified by the profiler for all lock algorithms. Our evaluations are performed on a 48-core machine having four 12-core Opteron 6172 processors, running Ubuntu 11.10 (Linux 3.0.0), with gcc 4.6.1 and glibc 2.13.

[2] Since a server's atomic operations are core-local, we have implemented optimized atomic CAS and increment operations without using the costly x86 instruction prefix `lock` that cleans up the write buffers.

Algorithm 2: Executing a critical section

```
thread local variables:
    int th_client_index; bool is_server_thread; server_t* my_server;
function execute_cs(lock_t* lock, function_t* code, void* context)
    var request_t * r := &lock->server->requests[th_client_index];
    if ¬is_server_thread or my_server ≠ lock->server then
        // RCL to a remote core
        r->lock := lock; r->context := context; r->code := code;
        while r->code ≠ null do
            ;
        return;
    else
        // Local nested lock, wait until the lock is free
        while local_CAS(&lock->is_locked,false,true) = true do
            // Another thread on the server owns the lock
            yield();
        // Execute the critical section
        code(context);
        lock->is_locked := false;
        return;
```

Algorithm 3: Management

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

4.1 Comparison with other locks

We have developed a custom microbenchmark to measure the performance of RCL relative to four other lock algorithms: CAS Spinlock, POSIX, MCS [18] and Flat Combining [13]. These algorithms are briefly presented in Fig. 6. To our knowledge, MCS and Flat Combining are currently the most efficient.

Spinlock	CAS loop on a shared cache line.
POSIX	CAS, then sleep.
MCS	CAS to insert the pending CS at the end of a shared queue. Busy wait for completion of the previous CS on the queue.
Flat Combining	Periodic CAS to elect a client that acts as a server, periodic collection of unused requests. Provides a generic interface, but not combining, as appropriate to support legacy applications: server only iterates over the list of pending requests.

Fig. 6: The evaluated lock algorithms.

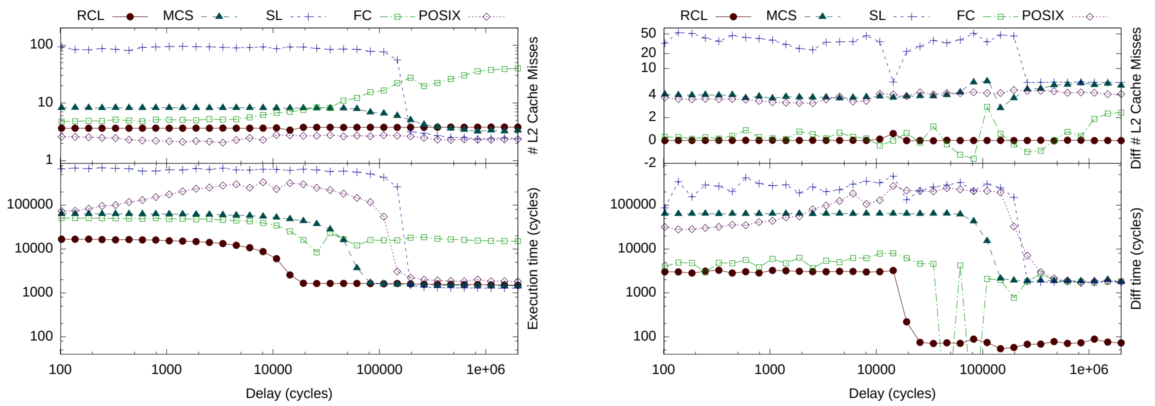
Our microbenchmark executes critical sections repeatedly on all cores, except one that manages the lifecycle of the threads. For RCL, this core also executes the RCL server. We vary the *degree of contention* on the lock by varying the delay between the execution of the critical sections: the shorter the delay, the higher the contention. We also vary the *locality* of the critical sections by varying the number of shared cache lines each one accesses (references and updates). To ensure that cache line accesses are not pipelined, we construct the address of the next memory access from the previously read value [30].

Fig. 7(a) presents the average number of L2 cache misses (top) and the average execution time of a critical section (bottom) over 5000 iterations when critical sections access one shared cache line. This experiment measures the effect of lock access contention. Fig. 7(b) then presents the increase in execution time incurred when each critical section instead accesses 5 cache lines. This experiment measures the effect of data locality of shared cache lines. Highlights are summarized in Fig.7(c).

With one shared cache line, at high contention, RCL has a better execution time than all other lock algorithms, with an improvement of at least 3 times.[3] This improvement is mainly due to the absence of CAS in the lock implementation. Flat Combining is second best, but RCL performs better because Flat Combining has to periodically elect a new combiner. At low contention, RCL is slower than Spinlock by only 209 cycles. This is negligible since the lock is seldom used. In this case, Flat Combining is not efficient because after executing its critical section, the combiner must iterate over the list of requests before resuming its own work.

RCL incurs the same number of cache misses when each critical section accesses 5 cache lines as it does for

[3] Using the SSE3 monitor/mwait instructions on the client side when waiting for a reply from the server, as described in Sec. 2.1, induces a latency overhead of less than than 30% at both high and low contention. This makes the energy-efficient version of RCL quantitatively similar to the original RCL implementation presented here.



(a) Execution with one shared cache line per CS

(b) Difference between one and five cache lines per CS

	High contention (10^2 cycles)			Low contention (2.10^6 cycles)		
	CAS/CS	Execution time (cycles)	Locality (misses)	CAS/CS	Execution time (cycles)	Locality (misses)
Spinlock	N	Bad (672889)	Very Bad (+53.0 misses)	N	Good (1288)	Bad (+5.2)
POSIX	1	Medium (73024)	Bad (+3.8 misses)	1	Medium (1826)	Bad (+4.0)
MCS	1	Medium (63553)	Bad (+4.0 misses)	1	Good (1457)	Bad (+4.8)
Flat Combining	ε	Medium (50447)	Good (+0.3 misses)	1	Bad (15060)	Medium (+2.4)
RCL	0	Good (16682)	Good (+0.0 misses)	0	Good (1494)	Good (+0.0)

(c) Comparison of the lock algorithms

Fig. 7: Microbenchmark results. Each data point is the average of 30 runs.

one cache line, as the data remains on the RCL server. At low contention, each request is served immediately, and the performance difference is also quite low. At higher contention, each critical section has to wait for the others to complete, incurring an increase in execution time of roughly 47 times the increase at low contention. Like RCL, Flat Combining has few or no extra cache misses at high contention, because cache lines stay with the combiner, which acts as a server. At low contention, the number of extra cache misses is variable, because the combiner often has no other critical sections to execute. These extra cache misses increase the execution time. POSIX and MCS have 4 extra cache misses when reading the 4 extra cache lines, and incur a corresponding execution time increase. Finally, Spinlock is particularly degraded at high contention when accessing 5 cache lines, as the longer duration of the critical section increases the amount of time the thread spins, and thus the number of CAS it executes.

To estimate which locks should be transformed into RCLs, we correlate the percentage of time spent in critical sections observed using the profiler with the critical section execution times observed using the microbenchmark. Fig. 8 shows the result of applying the profiler to the microbenchmark in the one cache line case with POSIX locks.[4] To know when RCL becomes better than all other locks, we focus on POSIX and MCS: Flat Combining is always less efficient than RCL and Spinlock is only efficient at very low contention. We have marked the delays at which, as shown in Fig. 7(a), the critical section execution time begins to be significantly higher when using POSIX and MCS than when using RCL. RCL becomes more efficient than POSIX when 20% of the application time is devoted to critical sections, and it becomes more efficient than MCS when this ratio is 70%. These results are preserved, or improved, as the number of accessed cache lines increases, because the execution time increases more for the other algorithms than for RCL.

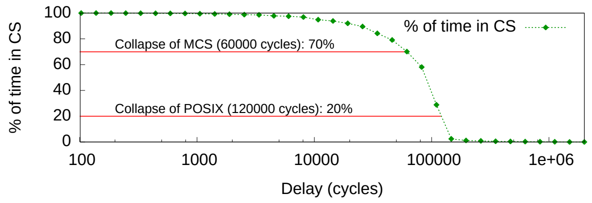
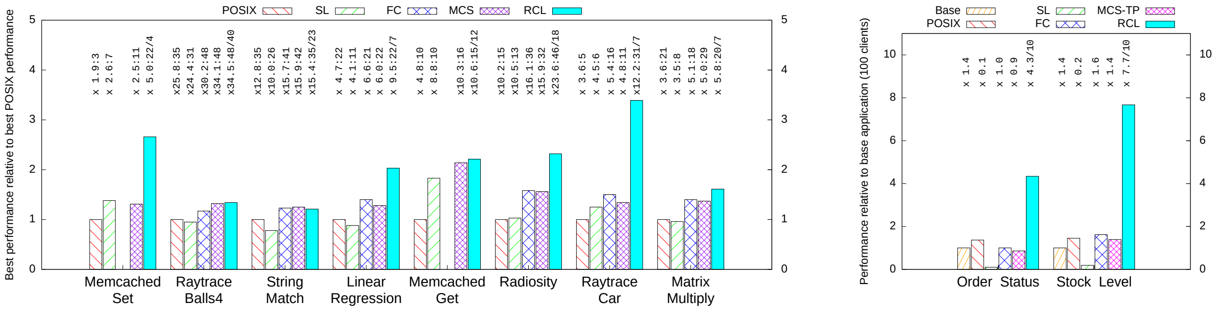


Fig. 8: CS time in the microbenchmark with POSIX locks.

4.2 Application performance

The two metrics offered by the profiler, i.e. 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 (critical section length, interactions between locks, etc.) affect critical section execution. We find, however, that using the time spent in critical sections as our main metric and the number of cache misses in critical sections as a secondary metric works well; the former is a good indicator

[4] Our analysis assumes that the targeted applications use POSIX locks, but a similar analysis could be made for any type of lock.



(a) Applications with one lock bottleneck

(b) Berkeley DB

Fig. 9: Best performance for each type of lock relative to the best performance for POSIX locks.

of contention, and the latter of data locality.

To evaluate the performance of RCL, we have measured the performance of applications listed in Fig. 3 with the lock algorithms listed in Fig. 7. Memcached with Flat Combining is omitted, because it periodically blocks on condition variables, which Flat Combining does not support. We present only the results for the applications (and locks) that the profiler indicates as potentially interesting. Replacing the other locks has no performance impact.

Fig. 9(a) presents the results for all of the applications for which the profiler identified a single lock as the bottleneck. For RCL, each of these applications uses only one server core. Thus, for RCL, we consider that we use N cores if we have $N-1$ threads and 1 server, while we consider that we use N cores if we have N threads for the other lock algorithms. The top of the figure (xα : n/m) reports the improvement α over the execution time of the original application on one core, the number n of cores that gives the shortest execution time (i.e., the scalability peak), and the minimal number m of cores for which RCL is faster than all other locks. The histograms show the ratio of the shortest execution time for each application using POSIX locks to the shortest execution time with each of the other lock algorithms.[5]

Fig. 9(b) presents the results for Berkeley DB with 100 clients (and hence 100 threads) running TPC-C's Order Status and Stock Level transactions. Since MCS cannot handle more than 48 threads, due to the convoy effect, we have also implemented MCS-TP [12], a variation of MCS with a spinning timeout to resist convoys. In the case of RCL, the two most used locks have been placed on two different RCL servers, leaving 46 cores for the clients. Additionally, we study the impact of the number of simultaneous clients on the number of transactions treated per second for Stock Level transactions (see Fig. 11).

Performance analysis For the applications that spend 20-70% of their time in critical sections when using POSIX locks (Raytrace/Balls4, String Match, and Memcached/Set), RCL gives significantly better performance than POSIX locks, but in most cases it gives about the same performance as MCS and Flat Combining, as predicted by our microbenchmark. For Memcached/Set, however, which spends only 54% of the time in critical sections when using POSIX locks, RCL gives a large improvement over all other approaches, because it significantly improves cache locality. When using POSIX locks, Memcached/Set critical sections have on average 32.7 cache misses, which roughly correspond to accesses to 30 shared cache lines, plus the cache misses incurred for the management of POSIX locks. Using RCL, the 30 shared cache lines remain in the server cache. Fig. 10 shows that for Memcached/Set, RCL performs worse than other locks when fewer than four cores are used due to the fact that one core is lost for the server, but from 5 cores onwards, this effect is compensated by the performance improvement offered by RCL.

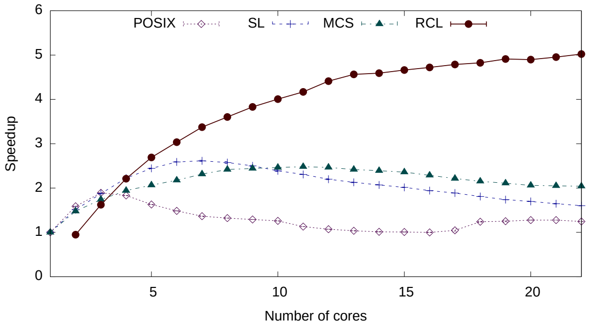


Fig. 10: Memcached/Set speedup.

For most of the applications that spend more than 70% of their time in critical sections when using POSIX locks (Radiosity, Raytrace/Car, and Linear Regression), RCL gives a significant improvement over all the other lock algorithms, again as predicted by our microbenchmark. Matrix Multiply, however, spends over 90% of its time in critical sections when using POSIX locks, but shows

[5] For Memcached, the execution time is the time for processing 10,000 requests.

only a slight performance improvement. This application is intrinsically unable to scale for the considered data set; even though the use of RCL reduces the amount of time spent in critical sections to 1% (Fig. 12), the best resulting speedup is only 5.8 times for 20 cores. Memcached/Get spends more than 80% of its time in critical sections, but is only slightly improved by RCL as compared to MCS. Its critical sections are long and thus acquiring and releasing locks is less of a bottleneck than with other applications.

In the case of Berkeley DB, RCL achieves a speedup of 4.3 for Order Status transactions and 7.7 for Stock Level transactions with respect to the original Berkeley DB implementation for 100 clients. This is better than expected, since, according to our profiler, the percentage of time spent in critical sections is respectively only 53% and 55%, i.e. less than the 70% threshold. This is due to the fact that Berkeley DB uses hybrid Test-And-Set/POSIX locks, and our profiler was designed for POSIX locks: the time spent in the Test-And-Set loop is not included in the "time in critical sections" metric.

When the number of clients increases, the throughput of all implementations degrades. Still, RCL performs better than the other lock algorithms, even though two cores are reserved for the RCL servers and thus do not directly handle requests. In fact, the cost of the two server cores is amortized from 5 clients onwards. The best RCL speedup over the original implementation is for 40 clients with a ratio of 14 times. POSIX is robust for a large number of threads and comes second after RCL. MCS-TP [12] resists convoys but with some overhead. MCS-TP and Flat Combining have comparable performance.

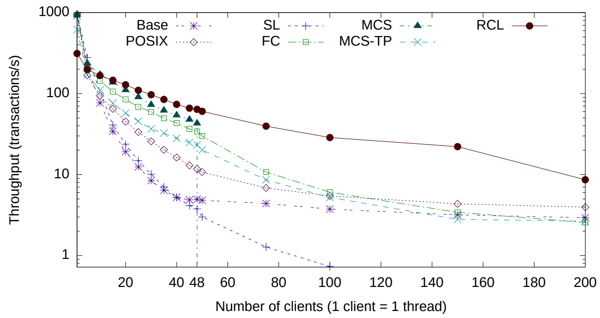


Fig. 11: Berkeley DB/Stock Level throughput.

Locality analysis Figure 12 presents the number of L2 cache misses per critical section observed on the RCL server for the evaluated applications. Critical sections trigger on average fewer than 4 cache misses, of which the communication between the client and the server itself costs one cache miss. Thus, on average, at most 3 cache lines of context information are accessed per critical section. This shows that passing variables to and from the server does not hurt performance in the evaluated applications.

Application	L2 cache misses on the RCL server
Radiosity	2.2
Raytrace/Car	1.8
Raytrace/Balls4	1.8
Linear Regression	2.4
Matrix Multiply	3.2
String Match	3.2
Memcached/Get	N/A†
Memcached/Set	N/A†
Berkeley DB/Order Status	3.3
Berkeley DB/Stock Level	3.6

† We are currently unable to collect L2 cache misses when using blocking on RCL servers.

Fig. 12: Number of L2 cache misses per CS.

False Serialization A difficulty in transforming Berkeley DB for use with RCL is that the call in the source code that allocates the two most used locks also allocates nine other less used locks. The RCL runtime requires that for a given lock allocation site, all allocated locks are implemented in the same way, and thus all 11 locks must be implemented as RCLs. If all 11 locks are on the same server, their critical sections are artificially serialized. To prevent this, the RCL runtime makes it possible to choose the server core where each lock will be dispatched.

To study the impact of this false serialization, we consider two metrics: *false serialization rate* and *use rate*. The false serialization rate is the ratio of the number of iterations over the request array where the server finds critical sections associated with at least two different locks to the number of iterations where at least one critical section is executed.[6] The use rate measures the server workload. It is computed as the total number of executed critical sections divided by the number of iterations where at least one critical section is executed, giving the average number of clients waiting for a critical section on each iteration, which is then divided by the number of cores. 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 mostly spins on the request array, waiting for critical sections to execute.

Fig. 13 shows the false serialization rate and the use rate for Berkeley DB (100 clients, Stock Level): (i) with one server for all locks, and (ii) with two different servers for the two most used locks as previously described. Using one server, the false serialization rate is high and has a significant impact because the use rate is also high. When using two servers, the use rate of the two servers goes down to 5% which means that they are no longer saturated and that false serialization is eliminated. This allows us to improve the throughput by 50%.

[6] We do not divide by the total number of iterations, because there are many iterations in application startup and shutdown that execute no critical sections and have no impact on the overall performance.

	False serialization rate	Use rate	Transactions/s
One server	91%	81%	18.9
Two servers	<1% / <1%	5%/5%	28.7

Fig. 13: Impact of false serialization with RCL.

5 Related Work

Many approaches have been proposed to improve locking [1, 8, 13, 15, 24, 26, 27]. Some improve the fairness of lock algorithms or reduce the data bus load [8, 24]. Others switch automatically between blocking locks and spinlocks depending on the contention rate [15]. Others, like RCL, address data locality [13, 27].

GLocks [1] addresses at the hardware level the problem of latency due to cache misses of highly-contended locks by using a token-ring between cores. When a core receives the token, it serves a pending critical section, if it has one, and then forwards the token. However, only one token is used, so only one lock can be implemented. Suleman *et al.* [27] transform critical sections into remote procedure calls to a powerful server core on an asymmetric multicore. Their communication protocol is also implemented in hardware and requires a modified processor. They do not address blocking within critical sections, which can be a problem with legacy library code. RCL works on legacy hardware and allows blocking.

Flat Combining [13], temporarily transforms the owner of a lock into a server for other critical sections. Flat Combining is unable to handle blocking in a critical section, because there is only one combiner. At low contention, Flat Combining is not efficient because the combiner has to check whether pending requests exist, in addition to executing its own code. In RCL, the server may also uselessly scan the array of pending requests, but as the server has no other code to execute, it does not incur any overall delay. Sridharan *et al.* [26] increase data locality by associating an affinity between a core and a lock. The affinity is determined by intercepting Futex [9] operations and the Linux scheduler is modified so as to schedule the lock requester to the preferred core of the lock. This technique does not address the access contention that occurs when several cores try to enter their critical sections.

Roy *et al.* [23] 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. 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, 16, 25]. 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 and databases designed with manycore architectures in mind use data replication to improve locality [28] and even RPC-like mechanisms to access shared data from remote cores [4, 7, 11, 19, 22]. 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 legacy benchmarks and widely used legacy applications 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. Second, we want to be able to migrate locks between multiple servers, to dynamically balance the load and avoid false serialization. One of the challenges will be to implement low-overhead run-time profiling and migration strategies. Finally, we will explore the possibilities of RCL for designing new applications.

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

Acknowledgments We would like to thank Alexandra Fedorova and our shepherd Wolfgang Schröder-Preikschat for their insightful comments and suggestions.

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*, 2011.

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

[3] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: featherweight synchronization for Java. In *PLDI'98*, pages 258–268, 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, 2009.

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

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

[7] E. M. Chaves 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.

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

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

[18] J. M. Mellor-Crummey and M. L. Scott. Synchronization without contention. In *ASPLOS*, pages 269–278. ACM, 1991.

[19] 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.

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

[21] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in Linux device drivers. In *EuroSys*, pages 247–260, 2008.

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

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

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

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

[26] 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.

[27] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *ASPLOS*, pages 253–264, 2009.

[28] 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*.

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

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