Remote Core Locking: Performance through Serialization

Michael Gebhard
michael.gebhard@fau.de

ABSTRACT

Today's many-core systems require scalable applications. Software needs to be able to use the concurrent computation potential offered by a large number of cores. However the scalability of many applications is limited by data locality and the performance of lock algorithms. To improve both locality and lock performance, Lozi *et al.* propose the concept of Remote Core Locking (RCL) [3]. This paper summarises the functionality of RCL and analyses the benchmarks conducted by Lozi *et al.* On the one hand, RCL improves the locality of shared data by migrating critical sections to a dedicated server core. RCL also offers higher scalability, as no communication between all threads is required. Instead all threads communicate pairwise with the server core. On the other hand RCL has drawbacks including false serialisation of independent critical sections, as well as reduced locality of private client data. Lozi *et al.*'s benchmarks partly substantiate the impact of these advantages and drawbacks. Nevertheless RCL allows for significant performance improvements on some applications.

CCS CONCEPTS

•Computing methodologies →Concurrent computing methodologies;

1 INTRODUCTION

The performance of concurrent applications on today's many-core systems relies on the scalability of these applications. To use the computation potential of many core hardware, applications must achieve a high level of parallelism. However, many concurrent algorithms and data structures depend on locks as a means of synchronisation. These locks guard critical sections. By concept critical sections provide linearisable access to shared data. In order to access shared data, an algorithm must first acquire the respective lock. Acquiring locks introduces contention between the threads of a concurrent program [5]. Accessing shared data in the respective critical section induces locality issues between the current thread and the thread that previously held the lock. According to Verghese *et al.* data locality is potentially the most important performance issue on shared memory architectures [6]. Remote core locking (RCL) provides both reduced contention as well as increased shared data locality. RCL migrates critical sections to a dedicated server core and replaces lock based synchronisation by a request array. Each thread has one request structure in this array. Threads thus no longer compete for a lock. Instead, each thread requests access to a critical section by writing to its own request structure. The execution of critical section on the dedicated core causes the shared data to remain local to this core and thus improves the locality of this shared data. Similar approaches to critical section migration are taken by Hendler *et al.*'s flat combining (FC) [1] and Suleman *et al.*'s accelerated critical sections (ACS) [4]. First, FC does not propose a general lock implementation. Instead it implements certain concurrent data structures. The locking functionality of FC is similar to RCL. In difference to RCL, FC uses the knowledge of the implemented data structure in its critical section server. This allows the FC server to combine concurrent requests to reduce their overall execution time. A general lock implementation like RCL cannot provide this feature. Furthermore, FC does not have a dedicated server core for a critical section. Instead, the role of the server is passed on between normal clients. This impedes data locality compared to RCL. Second, Suleman *et al.*'s ACS is a hardware based algorithm, unlike RCL and FC, which are entirely implemented in software. Suleman *et al.*'s approach describes an asymmetric many core architecture with new instructions. These instructions implement requests similar to those used in RCL. ACS, FC and RCL have one decisive functional difference. RCL can handle blocking inside critical sections, whereas FC and ACS require non-blocking critical sections. This makes RCL less intrusive to use in legacy applications. Lozi *et al.* also developed two tools to assist programmers in transforming applications to use RCL. First, a profiler that selects which locks of an application might profit from using RCL. Second, a reengineering tool that can automatically rewrite critical sections to use RCL. Given these two assets, changing a legacy application into one using RCL locks requires very little understanding of the application code. This paper further discusses the implementation of RCL, its advantages and drawbacks, an evaluation of Lozi *et al.*'s benchmarks and finally an outlook on possible future improvements of RCL.

2 RCL IMPLEMENTATION

A RCL lock consists of two parts. One part is a server that offers execution of critical sections. The other part consists

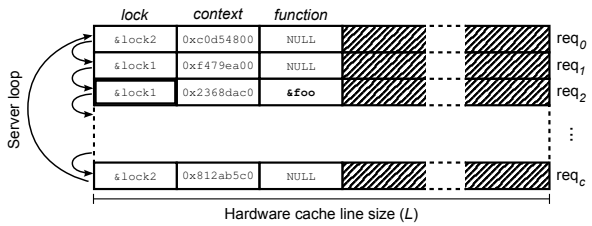


Figure 1: The request array. Client c_2 has requested execution of the critical section implemented by *foo* [3].

of several clients that requests such executions. This section first presents the structures used to communicate between the client and the server, then the transformation of an ordinary lock into an RCL lock in a client and finally the implementation of the server.

Client-Server-Communication

The server for an RCL lock manages an array of request structures. This array contains one request for each possible client, as shown in figure 1. Such a request consists of three content fields and padding to occupy exactly one hardware cache line. The first field points to the actual lock guarding the critical section. The second field points to a context structure given by the client. The last field points to the function that encapsulates the critical section. The context contains the parameters for this function. In figure 1 client c_2 requests execution of the function *foo*. *foo* contains a critical section provided by the shown server. This section is guarded by the lock *lock1*. To request the execution, c_2 must perform four steps. First, store the context in a context structure. Second, store the address of *lock1* to its request line. Third, write the address of the context to its request. Last, store the address of *foo* to its request. The change of the function field is recognised by the server. The function address must be the last of these writes to the request line. The order of writes ensures that the server reads all updated values of this line. Finally, the client must wait for the server to execute its request, by spinning on the function field until the server resets it to `NULL`. On the other side of the communication, the server continuously awaits requests (Listing 3). It iterates over the request array and executes pending requests. The server identifies pending requests by the value of their function fields. A request line with a function address unequal to `NULL` represents a pending request. After finishing the execution of a request, the server writes `NULL` to the request's function field and thus signals completion to the client. The server then continues to iterate over the request array.

Listing 1: Critical section from raytrace/workpool.c [3].

```
1  GetJob(..., int pid) {
2    [...]
3    lock(global_lock(pid))
4    wpentry = get_workpool_entry(pid)
5    if not wpentry exists then
6      state = WORK_POOL_EMPTY
7      unlock(global_lock(pid))
8      return WORK_POOL_EMPTY
9    set_workpool_entry(wpentry->next)
10   unlock(global_lock(pid))
11   [...]
12 }
```

Listing 2: Transformed critical section [3].

```
1  function(void *context) {
2    context->wpentry = get_workpool_entry(context->pid)
3    if not context->wpentry exists then
4      state = WORK_POOL_EMPTY
5      return 1
6    set_workpool_entry(wpentry->next)
7    return 0
8  }
9
10 GetJob(..., int pid) {
11   [...]
12   context = { pid }
13   ret = rcl_request(global_lock(pid), context, &function)
14   if ret == 1 then
15     return WORK_POOL_EMPTY
16   [...]
17 }
```

Client Side

On the client side the programmer transforms an ordinary lock into an RCL lock. Listing 1 shows a critical section before and listing 2 after transformation. The lock guards a critical section. The client encapsulates this critical section in a closure [2]. The closure contains the code of the critical section and all necessary parameters. This implements migration of this section to the server. The client builds a context data structure (line 12, listing 2) containing the parameters for this extracted function. The context combines all local variables required by the closure. The client requests a server to execute the closure and continues when the server signals completion of the closure. In case of fine-grained locking, one critical section can have more then one code paths to release its lock. The server must tell the client on which path to continue. The server encodes this in the return value of the critical section. The client then switches over this return value (line 14, listing 2) and continues on the correct path.

Server Side

The server consists of a management thread, a backup thread and at least one servicing thread. Servicing threads execute

Listing 3: RCL server: Servicing thread [3].

```
1  servicing_thread() {
2    while true do
3      timestamp = server->timestamp
4      server->alive = 1
5      foreach request in server->request_array do
6        if request->function != NULL and
7              local_CAS(request->lock, FREE, LOCKED) then
8          if request->function != NULL then
9            request->context =
10             request->function(request->context)
11           request->function = NULL
12         unlock(request->lock)
13 }
```

Listing 4: RCL server: Management thread [3].

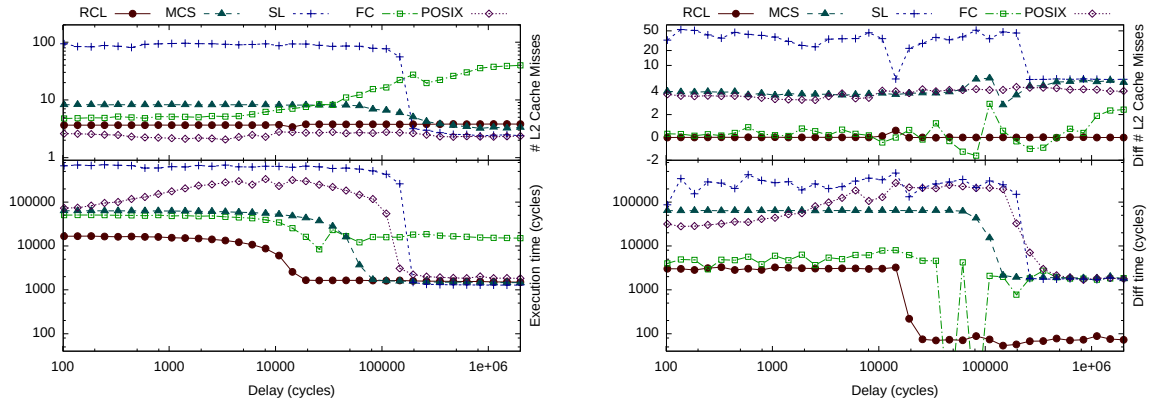
```
1  management_thread(...) {
2    while true do
3      if server->alive == 0 then
4        if free_threads < 1 then
5          spawn_servicing_thread()
6          atomic(free_threads++)
7        server->alive = 1
8        loop: while true do
9          foreach cur in servicing_threads do
10           if cur->timestamp < server->timestamp then
11             cur->timestamp = server->timestamp
12             elect(cur)
13             break loop
14         server->timestamp++
15     else
16       server->alive = 0
17     sleep_until(timeout)
18 }
```

requests containing a non-NULL function field as listing 3 shows. If a servicing thread blocks or spins during execution of a request, the server must ensure another servicing thread is available to execute new incoming requests. Servicing threads signal their availability by setting the *alive* flag of the server (line 4, listing 3) every time they finish iterating over the request array. The first purpose of the management thread is to ensure at least one servicing thread is available at all times. When all servicing threads are blocked, the backup thread is scheduled by the operating system and wakes up the management thread. A spinning servicing thread prevents the backup thread from being scheduled. Therefore the management thread must regularly check availability of the servicing threads. Lines 16 and 3-5 respectively show the management thread sleeping for a fixed amount of time and spawning a new servicing thread if necessary. The management runs at highest priority and thus preempts any other thread when it wakes up.

Running several servicing threads introduces two new problems. First the threads need to synchronise execution of requests. Hence a servicing thread acquires the lock associated with a request before executing it and releases the lock afterwards (lines 7, 12, listing 3). Each critical section provided by the server has one unique lock. The addresses of these locks are stored in each request of the respective critical section. The locks are only acquired by the servicing threads running on the server core and are implemented as a local CAS operation. Unlike ordinary locks, this local CAS operation has negligible performance impact and does not influence the scalability of the number of clients. Second the server must ensure responsiveness of all servicing threads. To avoid preemption of a servicing thread while it is executing a critical section, Lozi *et al.* the POSIX FIFO scheduling policy of the operating system scheduler. This allows the running servicing thread to execute requests uninterruptedly until it is blocked or until the management thread wakes up. If the running servicing thread spins within a critical section, the management thread must instruct the operating system to schedule a different servicing thread in order to avoid starvation. The management thread uses timestamps to guarantee that every servicing thread is scheduled eventually (lines 10-14, listing 4). Each servicing thread and the server itself has one timestamp. The management thread first selects the first thread with a lower timestamp than the server. Second, the management thread sets this servicing thread's timestamp to the one of the server and elects this servicing thread to be scheduled. Third, if no servicing thread has a lower timestamp, the management thread increments the server's timestamp.

3 ADVANTAGES AND DRAWBACKS

When a multithreaded application accesses shared data, the application must synchronise the data between its threads. Synchronisation can be done through a critical section guarded by an ordinary lock. In this case one thread acquires the lock and then modifies the data. The modification is done in the cache of the core the thread is running on. When another thread running on a different core then acquires the lock, this core's cache is invalid. The CPU must copy the data from the first cache to the second. According to Lozi *et al.* [3] this lack of locality is the main disadvantage of ordinary locks compared to RCL locks. A multithreaded application using RCL also has to synchronise its shared data. This application also uses critical sections guarded by locks. But the threads of this application request the execution of a critical section from an RCL server core. The CPU has to copy this request from the core of the client thread to the server core. A server thread then modifies the shared data. The shared data is only modified by the threads of this server. The server threads always run on one specific server core. Therefore the shared data in this core's cache remains valid. This allows the RCL



(a) Execution with one shared cache line per CS

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

Figure 2: Results of the microbenchmark [3].

servicing threads to execute critical sections without waiting for the CPU to copy shared data between caches. However the CPU still has to copy the request from the client core to the server core. This request is the private data of the thread running on the client core. The private data consists of the request line itself, the referenced context structure and the code of the function that is requested. The code of the requested function never changes. The code therefore can remain valid in the server's cache. The request line is a single cache line that the CPU must copy to the server for every request. The size and validity of the context structure depend on the application. Suleman *et al.* conclude that their RCL like algorithm can improve overall performance of an application, when the improved locality of shared data outweighs the decreased locality of private data [4]. RCL also comes with very little synchronisation overhead. For clients, the synchronisation overhead consists of writing one cache line. For the server, the overhead consists of one management thread, that wakes up regularly and iterates once over the list of servicing threads, and the servicing threads that each iterate over the request array of constant size. Especially no global CAS operation is necessary when executing a critical section. The CAS in line 7 of the servicing thread (listing 3) operates on cache local memory. It is therefore much faster than CAS operation accessing shared memory. Above all other overhead the main disadvantage of RCL in overhead comes from the server itself taking up one core on which no clients can run. On the other hand many concurrent algorithms are not perfectly scalable. This intrinsically limits the number of threads an application needs to reach its scalability peak. Today's many core systems offer enough cores to assign one core to every thread of such an application. This makes the disadvantage of a dedicated server core insignificant. Another disadvantage of RCL is the possibility of false

Listing 5: Critical section in microbenchmark [3].

```
1  thread() {
2    while true do
3      foreach line in context_cache_lines do
4        *line++
5      rcl_request(global_lock,
6                  context_cache_lines,
7                  critical_section)
8  }
9  critical_section(void *context_cache_lines) {
10   foreach line in context_cache_lines do
11     *line++
12   foreach line in shared_cache_lines do
13     *line++
14 }
```

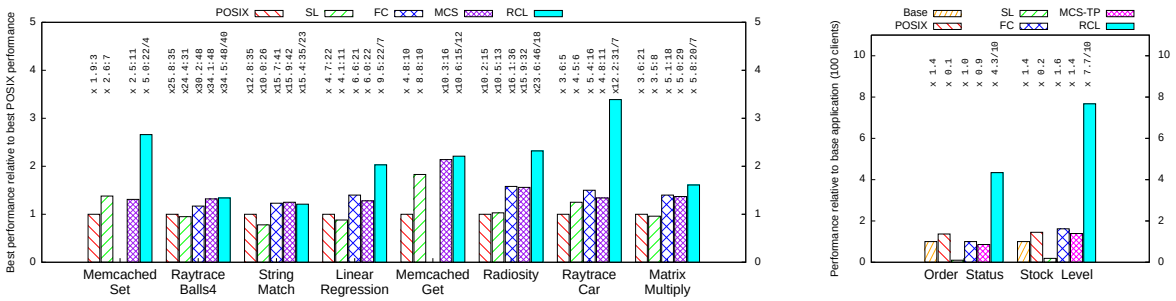
serialisation. This occurs when one RCL server core serves multiple independent critical sections. The server core can be busy with serving many requests for one critical section. This delays the execution of a request for a different critical section. An application using ordinary locks is able to concurrently execute such independent critical sections.

4 EVALUATION

Lozi *et al.* evaluated the impact of the advantages and drawbacks of RCL. Lozi *et al.* conducted 18 benchmarks commonly used to test the performance of lock implementations as well as one self-made microbenchmark.

Microbenchmark

The microbenchmark runs 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 [3]. Each lock is tested with 48 threads. Each thread modifies context cache lines, executes its critical section and then spins for a fixed



(a) Applications with one lock bottleneck

(b) Berkeley DB

Figure 3: Results of legacy benchmarks.

amount of time. The spinning time is used to simulate different levels of contention. For RCL 47 threads are spawned to leave one for the server. The critical section accesses both the context cache lines and the shared data cache lines. Accessing a cache line means incrementing its content. The critical section iterates over all cache lines it has to access as listing 5 shows. Lozi *et al.* executed the microbenchmark with one and five shared cache lines, but do not mention the number of context cache lines. The RCL server also only serves one critical section. This prevents false serialisation. Figure 2 shows the results of the microbenchmark. For one cache line access per critical section at high contention RCL outperforms all other tested locks. Lozi *et al.* conclude this is due to the absence of CAS in the RCL implementation. At low contention RCL performs comparable to all other locks except flat combining. Lozi *et al.* give no explanation why looses its performance advantage. When accessing five cache lines per critical section RCL outperforms all other locks on all contention levels. According to Lozi *et al.* this is due to the improved locality of shared data in RCL and the increase in cache misses in other lock implementations.

Legacy Benchmarks

The remaining 18 benchmarks include Memcached, Berkeley DB, the 9 applications of the SPLASH-2 benchmark suite and the 7 applications of the Phoenix2 benchmark suite. Lozi *et al.*'s profiler identifies 8 of these applications to have a single lock as bottleneck, which can be improved by RCL. The results of these 8 benchmarks are shown in figure 3 (a). 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 [3]. m is especially also given when RCL is not faster than all other locks. This is seen on String Match where RCL is outperformed by flat combining and MCS. Lozi *et al.* give no explanation why RCL performs badly on this benchmark. On Memcached Set, Linear Regression, Radiosity and Raytrace Car RCL performs significantly better than all other lock implementations. On the remaining 3 benchmarks RCL performs comparable to MCS, even though the profiler identifies these applications to be improvable with RCL. Berkeley DB Order Status and Stock Level are not identified by the profiler. Regardless RCL outperforms all other locks by a large margin on these two benchmarks as figure 3 (b) shows. According to Lozi *et al.* this is because their profiler cannot measure the hybrid Test-And-Set/POSIX locks used in Berkeley DB.

In summary the benchmarks presented by Lozi *et al.* show varying improvements from RCL. Even in the worst shown case RCL still performs comparable to all other locks. Lozi *et al.* do not present the results of the remaining 8 benchmarks. They state transformation to RCL has no performance impact on these benchmarks.

5 CONCLUSIONS

RCL is a lock implementation with minimal synchronisation overhead. It improves shared data locality by serialising critical sections on a dedicated core. The improved locality allows RCL to outperform all other tested ordinary lock implementations in certain scenarios. Lozi *et al.* developed a profiling and a reengineering tool. The profiling tool decides which locks in an application are likely to be improvable with RCL. Even though Lozi *et al.*'s benchmarks do not prove their profiler to be highly reliable. The reengineering tool then transforms these locks into RCL locks. RCL performs comparable or better than all lock implementations presented in Lozi *et al.*'s benchmarks. In the future RCL might be improved by dynamically adjusting itself at runtime. This includes migrating locks between several server for dynamic load balancing and avoidance of false serialisation, as well as switching between RCL and ordinary locking at runtime [3]. Some locks can only benefit from RCL at certain points during runtime. Dynamic adjustments would allow RCL to also

improve these locks. RCL might also be improvable by incorporating the combining feature of flat combining [1].

REFERENCES

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

[2] P. J. Landin. 1964. The mechanical evaluation of expressions. *Comput. J.* (1964), 308–320.

[3] Jean-Pierre Lozi, Florian David, Gal Thomas, Julia Lawall, and Gilles Muller. 2012. *Remote Core Locking: Migrating Critical-Section Execution to Improve the Performance of Multithreaded Applications.* https://www.usenix.org/system/files/conference/atc12/atc12-final237.pdf.

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

[5] Nathan R. Tallent, John M. Mellor-Crummey, and Allan Porterfield. 2010. Analyzing Lock Contention in Multithreaded Applications. *PPoPP* (2010).

[6] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. 1996. Operating system support for improving data locality on CC-NUMA compute servers. *ACM SIGPLAN* (1996), 279–289.