Remote Core Locking: Performance through Serialization

Michael Gebhard
michael.gebhard@fau.de

ABSTRACT
Today’s many core systems require scalable applications. The software needs to be able to use the concurrent computation potential offered by a large number of cores. 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 lock algorithm RCL [3]. This paper summarises the functionality of RCL and analyses the benchmarks conducted by Lozi et al. On the one hand RCL improves shared data locality by migrating critical sections to a dedicated server. RCL also improves locking performance by replacing lock acquisition by a remote procedure call to this server. 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 argue 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. Many concurrent algorithms and data structures depend on locks as means of synchronisation. These locks guard critical sections. The 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]. Once a thread acquires a lock it enters the respective critical section. In this critical section the thread accesses shared data. Accessing shared data 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. Remote core locking (RCL) provides both reduced contention as well as increased shared data locality. RCL replaces locks by a request array. Each thread has one request structure in this array. Threads no longer compete for a lock. Instead each thread requests access to a critical section by writing to its own request structure. To improve data locality RCL executes critical sections on a dedicated core. This leads to the shared data remaining local to this core. 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]. FC does not propose a general lock implementation. It rather 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. The server combines contemporaneous requests and optimises them. This is a feature that cannot be provided by a general lock implementation. 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. 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 RCL like requests. 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
An RCL lock consists of two parts. One part is a server that offers execution of critical sections. The other part consists of several clients that requests such executions. This section
first elaborates on the structures used to communicate between the client and the server, then on transformation of an ordinary lock into an RCL lock in a client and finally on the implementation of the server and its issues.

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 is the critical section. The mentioned context contains the parameters for this function. In Figure 1 a client \( c_i \) may request execution of the function \( foo \). \( foo \) contains a critical section served by this server. This section is guarded by the lock \( lock_1 \). To request the execution \( c_i \) must perform four steps. First, store the context in a context structure. Second, store the address of \( lock_1 \) 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 will be recognised by the server. The function address must be the last of these writes to the request line. This ensures that the server reads all updated values of this line. Finally the client must wait for the server to execute this request. The client does so by spinning on the function field until it is reset to NULL. On the other side of the communication the server awaits requests (Listing 3). The server iterates over the request array and executes pending requests. The server identifies pending requests by the value of their function fields. Any request line with a function address unequal to NULL is pending. The server may be unable to execute a pending request. This happens if the lock of this request is already taken. The server skips these requests. After the execution of a request the server writes NULL to the request’s function field. This signals completion to the client. The server then continues to iterate over the request array.

![Figure 1: The request array. Client \( c_2 \) has requested execution of the critical section implemented by \( foo \) [3].](image)

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

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

```GetJob(..., int pid) {
    lock(global_lock(pid))
    wpentry = get_workpool_entry(pid)
    if not wpentry exists then
        state = WORK_POOL_EMPTY
        unlock(global_lock(pid))
    return WORK_POOL_EMPTY
    set_workpool_entry(wpentry->next)
    unlock(global_lock(pid))
    return wpentry->next
}
```

Listing 2: Transformed critical section [3].

```c
Listing 2: Transformed critical section [3].

function(void *context) {
    context->wpentry = get_workpool_entry(context->pid)
    if not context->wpentry exists then
        state = WORK_POOL_EMPTY
        return 1
    set_workpool_entry(wpentry->next)
    return 0
}
```

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 it is signalled by the server. In case of fine-grained locking, one critical section can have more than 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.
Remote Core Locking: Performance through Serialization

Listing 3: RCL server: Servicing thread [3].
```
servicing_thread() {
  while true do
    timestamp = server->timestamp
    server->alive = 1
    foreach request in server->request_array do
      if request->function != NULL and
        local_CAS(request->lock, FREE, LOCKED) then
        if request->function != NULL then
          request->function(request->context)
          unlock(request->lock)
        end
      end
    end
}
```

Listing 4: RCL server: Management thread [3].
```
management_thread(...) {
  while true do
    if server->alive == 0 then
      if free_threads < 1 then
        spawn_servicing_thread()
        atomic(free_threads++)
        server->alive = 1
      end
      loop: while true do
        foreach cur in servicing_threads do
          if cur->timestamp < server->timestamp then
            elect(cur)
            break loop
          end
        end
      end
    else
      server->alive = 0
      sleep_until(timeout)
  end
}
```

Server Side
The server consists of a management thread, a backup thread and at least one servicing thread. Servicing threads execute requests containing a non-NULL function field as listing 3 shows. To avoid race conditions between servicing threads, they acquire the lock of a request and check again if the request is pending (lines 6-8). If the request is still pending the servicing thread can safely execute it.

The management thread ensures liveness and responsiveness. Issues with either of these arise when a servicing thread cannot continue execution. A servicing thread may be blocked in a critical section. Blocking can be caused by performing I/O or waiting for a condition variable or a lock. A servicing thread can also spin in a critical section. Spinning can be caused by a spin lock or a nested RCL. In case of a nested RCL the servicing thread would spin on a request as described in the section on Client-Servicing-Communication. The management thread ensures that at least one servicing thread is available. A servicing thread is available if it is taking incoming requests. All servicing threads set the alive flag (line 4, listing 3) when they are done executing a request.

The management thread regularly checks this flag to see if any servicing thread has made progress. To guarantee that the management thread runs regularly, it waits for timeouts (line 16, listing 4) and runs at highest priority.

If all servicing threads are blocked the backup thread ensures that the management thread runs immediately. The backup thread runs at lowest priority. If the backup thread is scheduled, it wakes up the management thread and yields. The management thread spawns a new servicing thread if the alive flag is not set (lines 4-6). This ensures liveness.

However spinning servicing threads may cause starvation and therefore cause unresponsiveness. To avoid starvation the management thread also works as a scheduler. The management thread elects a servicing thread to run in a FIFO manner (lines 9-13). Each servicing thread and the management thread have a timestamp. If a servicing thread has a lower timestamp than the manager that thread is elected. The management thread updates the timestamp of the elected thread and schedules it (line 11). If no servicing thread has a lower timestamp the manager increases its own timestamp (line 13) and starts over to elect a servicing thread. This ensures that every thread runs eventually and therefore guarantees responsiveness.

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 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 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.

Listing 5: Critical section in microbenchmark [3].

```c
thread() {
  while true do
    foreach line in context_cache_lines do
      line++
      rcl_request(global_lock,
                  context_cache_lines,
                  critical_section)
  end
  critical_section(void *context_cache_lines) {
    foreach line in context_cache_lines do
      line++
  end
  foreach line in shared_cache_lines do
    line++
  end
}
```

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 selfmade 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 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 loses 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 (\( x/a : n/m \)) reports the improvement \( a \) 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 Berkley 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 improving these locks with RCL. RCL might also be improvable by incorporating the combining feature of flat combining [1].

**REFERENCES**


