# Seminars

l.andrea195

May 2020

## 1 Introduction

In this paper the authors propose a new lock algorithm called Remote Core Locking. It's presented in 2012 in Boston. In the last 20 years, several approaches have been developped for optimizing critical section because many application nowadays obtain best performance with a lower number of cores than those are available on today's multicore architecture. Performance are influenced by the lock algorithm in particular they should improve access contation and cache misses.

How is possible to see in this pictures: Authors make a lot of experiments with 18 application whose the amount of time spent in critical section grows when the number of cores used increase. In the above picture instead we could see the memcached application where a get operation works in best performance with 10 cores instead a set with 2. Test are done with Opteron 6172 with 48 core running at 3.0.0 Linux kernel.
The time spent in critical sections and the number of cache misses are not the only metrics used for evaluate performance of locks usage into a critical section, because there're also many other factors. How we can see at right a Memcache representation where RCL lock result as the best. Furthermore Flat Combining is omitted, because it periodically blocks because it does not support condition variable. Memcached/Set, spends only 54% of the time in critical sections when using POSIX locks. RCL gives a large improvement due improvement cache locality.
In the last years there will be presented different solution to this problem:

- Execution a succession of critical sections on a single server core to improve cache locality
- Insert a fast transfer of control from other client cores to the server, to reduce access contention

- Hardware-based solution whose perform fast transfer of control,
- Software solution where the server is an ordinary client thread and the role of server is distributed by the client thread approach with introduce more overhead

Authors design a solution which try to reduce the problem of bus saturation and improve cache locality for having better performance on time execution.

The following blocking techinque is implemented entirely in software on a x86 architecture. The purpose it to improve performance of the execution of the critical section into application that have multi core architectures. RCL locks get better performance than other kind of locks:(POSIX, CAS SPIN-LOCK, MCS, FLAT COMBINIG).

RCL replace lock acquisition with an optimized remote procedure call, to a dedicated server core. The importance of this factor is the advantage of storing shared-information in the sever core 's cache. It's important because we resolve the problem of need to transfer data protected by a lock to a core when it acquire the lock because the shared-data is transfer to the server into a dedicated cache line for each client to achieve busy-wait synchronization.
In the following picture we could see the requests made by the clients to the server which contain the role to handler of the critical section.
The communication between client and server is done through an array of C*L bytes where C is a number higher than the number of core and L is the size of an hardware cache line.The cache line identify a request made by the client to server and it is composed by three box:the address of the lock, the address of the context, the address of the function that need the critical section. The function box is form by the NULL value in the case no critical section are required.

- From client side after it makes the request and write on cache line, it wait until the third box value is reset to NULL in case of saving power on client it is going to sleep until the reset of the function box ,with SSE3 monitor/mwait.
- From server side it iterates the over requests waiting for one of the requests to contain a non-NULL value in its third word.

For implement RCL algorithm:

- We need to identify the lock that we want to transform from POSIX to RCL

- We need to identify the core that we want to transform core responasble of running the server
- PROFILER is the element used for evaluated which lock and the core should be chosen according **locks frequently used and time spent in critical section**
- Reeingeeniring Tool Coccinelle used for transforming critical section to remote procedure call, in a way that it' seen as separate function in which the main problem is the shared variable: Values are passed to and from the server through an auxiliary structure or client's cache line

The core algorithm is done by a server thread which shouldn't be blocked at OS level and never spin to a waitloop. There're three situation that induce to a deadlock because the server is unable to execute critical section of other locks.

- The thread could be blocked at the OS level because a critical section tries to acquire a POSIX lock, which is already held
- The thread could spin if the critical section try to acquire a spinlock
- The thread could be preempted at the OS level when its timeslice expires or caused by page fault

The runtime ensure responsiveness and liveness respectively avoiding the block of thread at OS level or inversion priority and managing at run time a pool of threads for each server : if the servicing thread is blocked/waited, replace it with another in the pool . RCL runtime which is supported by the Posix lock,manage a pool of thread for each server and in the case a servicing thread blocks or waits there's always a free thread which come from an help. For being sure of the existence of a free thread when is required we use an highest priority thread called "management thread" which is activated at each timeout and check if there's at least one progress since its last activation, otherwise it tries to modify the priority.There's also a backup thread at lowest priority used in the cAse of block of all therads in the OS and woke up the management thread. RCL runtime implements a POSIX FIFO scheduling policy for avoid the thread preemption then to reduce the delay for minimizing the length of the FIFO queue. Using FIFO policy could induce priority inversion between threads for avoid it we use free lock algorithm.
With one shared cache line Rcl has a better execution than all other lock algorithms with an improvement of at least 3 times. At low contetion each

request is served immediately so there're low difference performance and the number of miss is variable. At higher contetion each critical section has to wait for the other to complete For estimating which locks should be transformed into RCLs we based on % time spent in critical section using the profiler. In the below picture we could see how in high contention($10^2$ cycles) and low contention($2x10^6$ cycles) RCL works better: it's possible to see in comparison with others locks.Each test is with an average of 30 runs).

In the RCL higher is the contation less is the delay. Locality changes according the number of cache lines. With one shared cache line at high contention, RCL has a better execution time than all other lock algorithm RCL perform better than Flat Combining, best of the other, which has to periodically elect a new combiner. RCL, at low contention, is slower than Spinlock.

For adapting Berkley DB application to the usage of RCL you need to allocate the 2 most used lock and then other 9. All 11 locks should be implemented as RCLs on the same server. Their critical sections (refer to 11 locks) are artificially serialized. To prevent it the server core is chosen where each lock is dispatched . Now we focus the impact of the serialization with two metrics:

- False serialization 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.

- Use rate: 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.

It's important how change the rate between one or 2 different server: High rate with 1 and elimination of false serialization and increasing of throughput of 50 %

RCL powerful is when an application relies on highly conteded locks then we aspect to:

DESIGN NEW APPLICATION WITH THESE STRATEGIES
CONSIDER THE DESIGN AND THE IMPLEMENTATION OF AN ADAPTIVE RCL RUNTIME.
SYSTEM ABLE TO DYNAMICALLY SWITCH BETWEEN LOCKING STRATEGIES

CAPABILITY TO MIGRATE LOCKS BETWEEN MULTIPLE SERVERS FOR BALANCING DYNAMICALLY THE LOAD AND AVOID FALSE SERIALIZATION.

How we can see we've better performance into RCLs refering to Execution time and locality miss In the comparison with others locks:

- in high contention refer to latency microbenchmark
  - 3.3 over Flat combining
  - 4.4 over a Posix
- On memcached RCL provides a speed up to:
  - 2.6 over Posix
  - 2.0 over MCS
  - 1.9 over Spinlock

Flat combining is always less efficient than RCL and Spinlock is only efficient at very slow contention. Time spent in critical section is consider the prime estimator and the number of cache miss as the second but they don't determine whether an application will benefit from RCL, there're also other measurements. RCL gives significantly better performance than POSIX lock for example in Memcached/Set RCL imporove performance in terms of cache locality. Time spent in critical section is consider the prime estimator and the number of cache miss as the second but they don't determine whether an application will benefit from RCL, there're also other measurements. How it's possible to see RCL performance are evaluated in terms of throughput better than the other lock algorithms in particular when the number of client increases