Remote Core Locking: Migrating Critical-Section Execution to Improve the Performance of Multithreaded Applications, Jean-Pierre Lozi, USENIX ATC 2012

Abstract

- 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.

RCL

- 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.
- RCL transforms critical sections into RPC (Remote Procedure Calls) handled by a single server thread on a dedicated server co

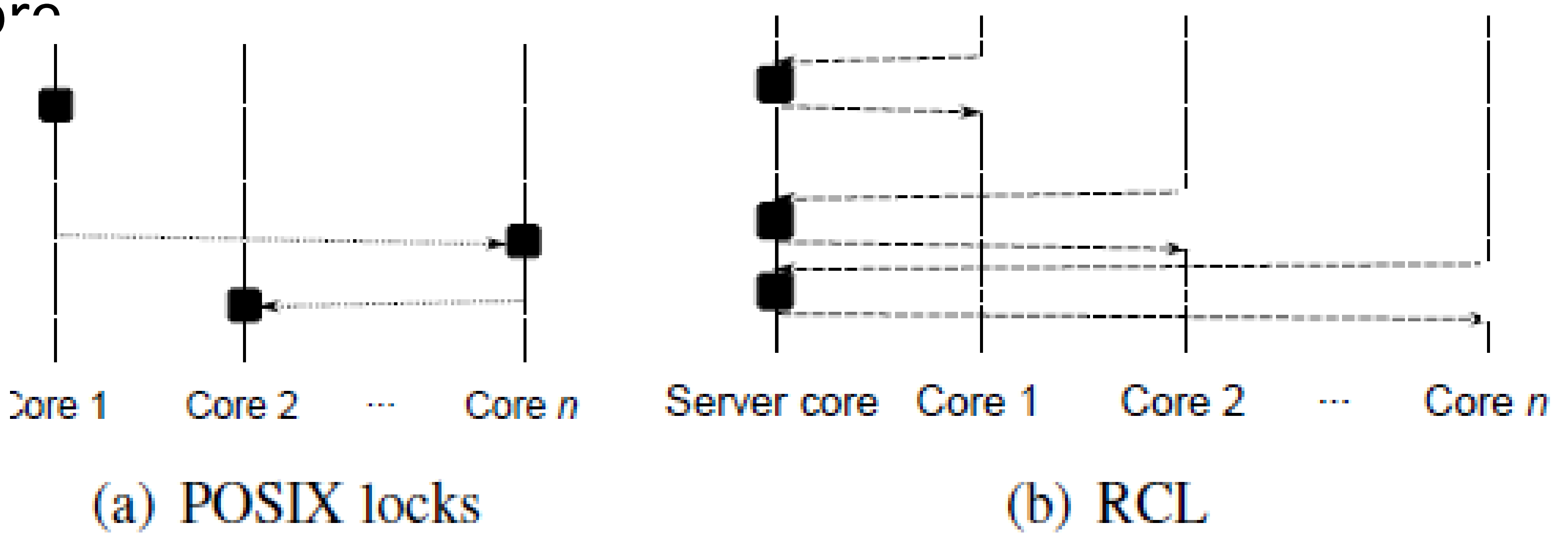

Fig. 1: Critical sections with POSIX locks vs. RCL.

- Shared data: 2D array workpool[][]
- Original code: workpool[][] migrates frequently between different cores, using cache coherence protocol
- RCL: workpool[][] stays on the server core, accessed only by the single server thread.

```
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); }
  . . .
}
```

```
INT GetJob(RAYJOB *job, INT pid) {
  . . .
  ALOCK(gm->wplock, pid) /* lock acquisition */
  wpentry = gm->workpool[pid][0];
  if (!wpentry) {
    gm->wpstat[pid][0] = WPS_EMPTY;
    AULOCK(gm->wplock, pid) /* lock release */
    return (WPS_EMPTY);
  }
  gm->workpool[pid][0] = wpentry->next;
  AULOCK(gm->wplock, pid) /* lock release */
  . . .
}
```

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

Fig. 5: Transformed critical section.

Using Elimination and Delegation to Implement a Scalable NUMA-Friendly Stack

Irina Calciu et al, HotPar 2013

Abstract

- Emerging cache-coherent non-uniform memory access (cc-NUMA) architectures provide cache coherence across hundreds of cores. These architectures change how applications perform: while local memory accesses can be fast, remote memory accesses suffer from high access times and increased interconnect contention. Because of these costs, performance of legacy code on NUMA systems is often worse than their uniform memory counterparts despite the potential for increased parallelism.
- We explore these effects on prior implementations of concurrent stacks and propose the first NUMA-friendly stack design that improves data locality and minimizes interconnect contention. We achieve this by using a dedicated server thread that performs all operations requested by the client threads. Data is kept in the cache local to the server thread thereby restricting cache-to-cache traffic to messages exchanged between the clients and the server. In addition, we match reciprocal operations (pushes and pops) by using the rendezvous elimination algorithm before sending requests to the server. This has the dual effect of avoiding unnecessary interconnect traffic and reducing the number of operations that change the data structure. The result of combining elimination and delegation is a scalable and highly parallel stack that outperforms all previous stack implementations on NUMA systems.

Delegation

- We use one dedicated server thread that accepts push and pop requests from many client threads. The communication is implemented in shared memory, using one location (which we call slot) for each client. The server loops through all the slots, collecting and processing requests and writing the responses back to the slots. The clients post requests to their private slots and spin-wait on that slot until a response is provided by the server
- Optimization: assign several threads to the same slot by thread id, using a spinlock to protect each slot

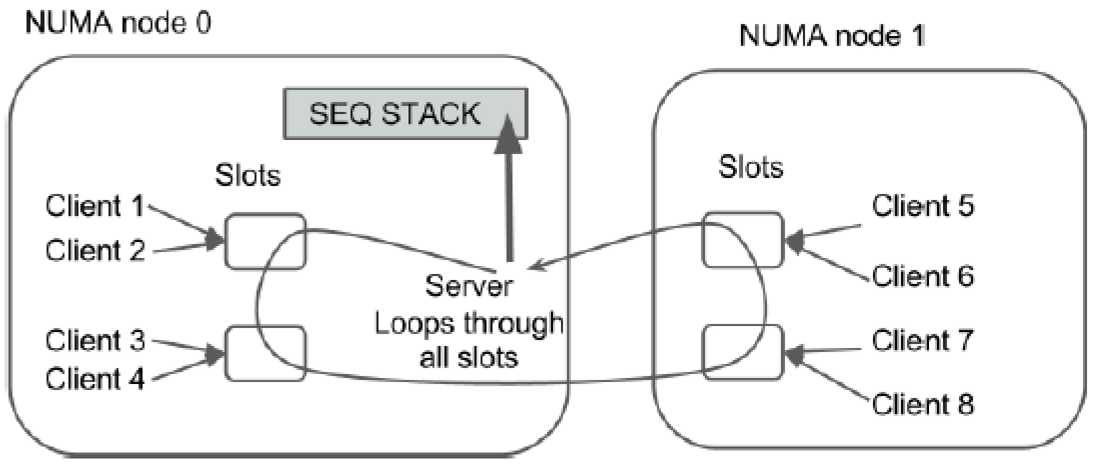

Figure 2: Clients posts their requests in shared local slots and wait for the server to process them. The server loops through all the slots, processes requests as it finds them and immediately posts back the response in the same slot. The sequential stack is only accessed by the server thread; therefore, the top part of the stack remains in the server's L1 cache or LLC all throughout execution.

Elimination

- Uses an additional data structure to allow threads performing push operations to meet threads performing pop operations and exchange their arguments. This is equivalent to the push being executed on the stack and immediately being followed by a pop.
- If one thread fails to find its inverse operation being performed by another thread, then the elimination attempt times out and the thread accesses the stack directly.
- Works best when the number of inverse operations are roughly equivalent.
- Combine elimination w/ delegation: Threads first try to eliminate and, if they time out, they delegate their operation to the server thread. Delegation ensures that the data remains in the server's cache, while elimination enables parallelism, thus making the NUMA-friendly stack more scalable.

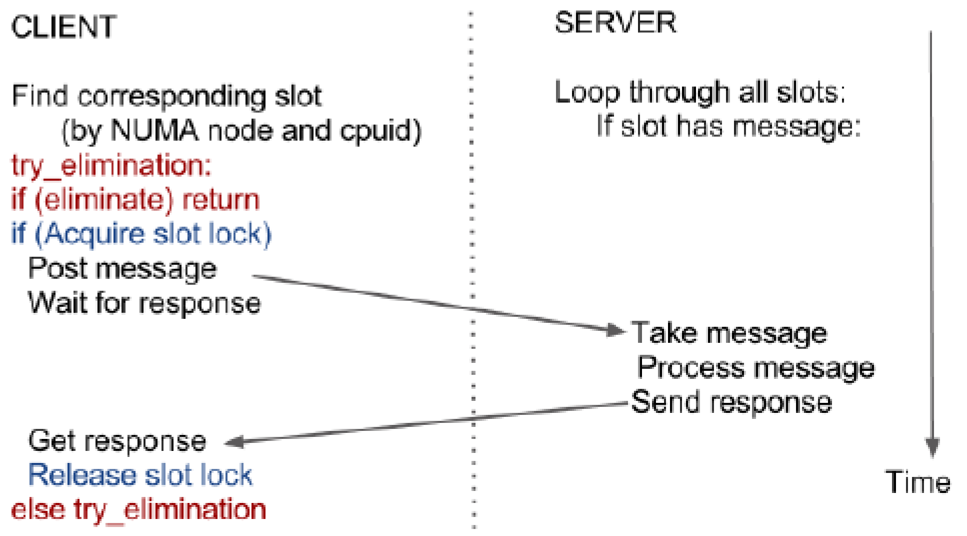

Figure 3: Communication protocol between a client thread and the server thread using slots

(a) (Black) Single thread per slot: each thread posts requests in its private slot, without any synchronization.

(b) (Blue) Multiple threads per slot: threads share slots, so they need to acquire the slot's spinlock before writing the request.

(c) (Red) Elimination: Threads first try to eliminate; if they fail they then try to acquire the slot spinlock and submit a request, but if the lock is already taken, they go back to the elimination structure; they continue this loop until either they eliminate, or they acquire the spinlock.

Local vs. Global Elimination

- The initial stage of elimination can still cause a number of invalidations between different NUMA nodes' caches because each of the threads accesses the same shared structure when performing elimination.
- To overcome this bottleneck, our NUMA-friendly stack splits the single elimination data structure into several local structures, equal to the number of NUMA nodes.
- To minimize interconnect contention, we limit elimination to occur only between those threads located on the same socket.

Perf. Comparisons

- nstack el: NUMA stack with local elimination and delegation
- nstack: delegation only, no elimination
- nstack el gl: global elimination and delegation
- nstack el st: local elimination and single thread per slot
- rendezvous stack: global elimination and direct access. (no delegation)
- lfstack: lock-free stack
- rendezvous loc: local elimination and direct access, based on lfstack
- fcstack: flat combining stack, based on lfstack
  - A thread that acquires a lock for a data structure executes operations for itself and also for other threads that are waiting on the same lock. The server is an ordinary client thread, and the role of server is handed off between clients periodically.
  - Outperforms the rendezvous stack when there is no significant potential for elimination (i.e., in unbalanced workloads).

Comparisons between [Lozi12] and [Calciu13]

- [Lozi12] is more general, and can handle arbitrary critical section code; [Calciu13] focuses on stack data structure with push()/pop() operations, which enables “elimination”, when a push() and a pop() cancel each other.
- [Lozi12] is a form of “delegation” in [Calciu13]; [Calciu13] has both “delegation” and (local and global) “elimination”.
- Both require source code modifications.