RS3LAB

École Polytechnique Fédérale de Lausanne

Server-sided range locking is better than client-sided locking

by Jeremy Dezalos

Master Thesis Report

Prof. Dr. Sanidhya Kashyap
Thesis Advisor

Vishal Gupta
Thesis Supervisor

EPFL IC IINFCOM
RS3Lab, INN 240
Station 14, 1015 Lausanne
Switzerland

February 13, 2024

Abstract

Distributed range locking allows remote multiple processes to access shared memory while keeping it consistant. It is a big contention point because every process must perform this procedure to read or write data. This is why optimizing range lock acquisition is crucial to have good performances. Recent studies found that using a system where only the clients are active when locking is faster compared when the server actively locks ranges for the clients.
In this report, we investigate new ways to implement server-sided range locks. Using combination, our best implementation achieved a 1.76x speedup compared to the best performing client-sided locking algorithm.

Contents

Chapter 1

Introduction

RDMA (Remote direct memory access) has become widely used in distributed computing, especially in computer clusters, in the recent years due to the improved throughput and latency it provides compared to older communication protocols. It allows multiple processes to access memory on a remote machine concurrently.
A problem that arises from this setup is how to manage the memory in order to avoid unsafe memory accesses. The usual method to solve this problem is to use locks, giving only one process at a time the rights to read and modify data. Many techniques exist to solve this problem, locking the whole memory for one thread is a simple solution, albeit an unefficient one. Designing faster algorithms requires locking only the memory that a process needs.
[6] and [9] each propose a way to use RDMA that use only the resources of the client while the server only stores the memory, made available by RDMA. They do not use the server's CPU resources for the locking process because they suppose that the server's CPU is the bottleneck of the algorithm. They find that locking mechanisms that are client-oriented achieve better throughput compared to designs that use server resources to lock memory. However, the reference implementations of server-sided locking only include designs that allow all server threads to lock the memory. That means they have not properly shown that client-sided locking is better because other server-sided locking algorithms were not included.
In a previous work [4], we found that combination, a way that lets only one server thread lock the memory while the other server threads wait until the operation is complete achieves better results compared to traditional locking techniques. This result was, however, found in a setting that used only one machine. To adapt this design into a distributed setup, one server thread will be dedicated to be the combiner while all the other server threads will handle requests from the clients.
However, there is not only one way to manage clients' requests. When a client sends a request to lock a range, it must wait until the server is done locking that range, after that the client must know when it can proceed to its critical section. We designed two ways to retrieve this information. The first method is to get the information the same way the client sends the request by waiting until the server gets back to him. For second method, we designed is a mix between

server-sided and client-sided locking in which the client will poll via one-sided RDMA operations on a flag stored in the server's memory in order to reduce server's CPU consumption.
In this report, we will show that server-sided range locking can be faster than client-based range locking in a distributed environment.
Our experiments showed that our best implementation performed 1.76 times better than the best client-sided locking algorithm on a realistic benchmark.
In this report, we present two different implementations of the combination technique in a distributed setting. The difference between the two of them is how the client threads obtain the information that the range they want to lock has been effectively locked. We compare our two designs against the best performing server-sided and client-sided locking algorithms using a micro-benchmark and a benchmark from Filebench and interpret the results.

Chapter 2

Background

2.1 Combination

In concurrency, the usual way processes lock memory is that they compete to modify a shared data structure and acquire the lock. This is the most instinctive way to acquire a range because all processes try to get access to the memory as fast as they can. A drawback of this is that the structure suffers heavy contention so its line in the cache is often invalidated, which slows down the program.
Another way to lock memory is called *combination*. Introduced by [5], combination limits the access of the data structure that holds the locks to only one process at a time. This process is called the *combiner*. The other processes manage a linked list using atomic operations to store the ranges they want to lock and wait for the combiner to execute their request.

2.2 RDMA

Remote direct memory access, or RDMA, allows two computers to send data to each other without having to rely on the processor, the operating system or needing to switch the context. RDMA works because, in both computers, the network interface card (or NIC), are running a protocol that lets a machine read inside the main memory of the other without needing to spend time on copying the data in a buffer to be sent over the network. Because of these properties, RDMA provides higher throughput and lower latency compared to the traditional transport protocols.
RDMA defines a *queue pair* by being a pair of a *send queue* and a *receive queue* and this queue pair is also associated with a *completion queue*. After the two machines connect their queue pair, they have two choices to communicate which can either be:

2.2.1 Two-sided communication

Two-sided communication involves both the sender and the receiver. First, the receiver puts inside its receive queue a *receive request* containing a pointer in its main memory that the sender will write in by putting a *send request* inside its own send queue. This will consume the receive request from the receiver and it will be able to read what the sender has written. For this method to work, the receiver must always post enough receive requests or else the send requests will be dropped.
Using two-sided communication is easy to perform because it resembles the flow of socket networking: the passive side listens and the active side sends data. However the receiver must be active because it constantly needs to put receive requests in its receive queue to function which uses its CPU resources.

2.2.2 One-sided communication

One-sided communication needs only one party for the communication to succeed. The client can read data inside the main memory of the server, write inside the memory of the server or perform atomic operations, such as compare-and-swap (CAS) or fetch-and-add (FAA). To perform any of these operations, the client puts a send request containing all the necessary operations inside its send queue.
Using one-sided communication, the server does not have to do anything at all, which frees up CPU resources. A drawback of this design is that the client cannot know when the operations are completed. A way to circumvent this problem is by polling the completion queue because each time an operation completes, an entry is added in the completion queue. This solves the problem of feedback but forces the client to wait until the operation finishes to move on, which increases latency.

2.2.3 Batching operations

When we have multiple RDMA operations to execute that do not depend on each other, we use a technique called *batching*. Instead of posting one work request, poll and repeat for the next operation. We can post several work requests in the *send queue* at once and then poll on the completion queue. This reduces latency because those operations are sent in parallel.
An additional optimization for batching is to put all the work requests into a linked list which is then put in the send queue. This reduces latency even more because it allows the driver to perform optimizations.

2.3 RPC

A Remote procedure call, or RPC, is a protocol to communicate between two processes, local or remote. A RPC starts when the client sends a request to the server and, then, the server responds. The client blocks while the server is busy processing the request.

2.3.1 eRPC

eRPC[7] is a state-of-the-art tool to use RPCs in datacenter environments that we use to run server-sided locking algorithms. The communication happens between two machines, the server and the client. eRPC supports UDP over lossy Ethernet but in our case we use it with the two-sided communication verbs from RDMA and the network link protocol is Infiniband.

2.4 Lamport's bakery algorithm

Lamport's bakery algorithm is an algorithm that provides mutual exclusion to processes that want to access the same resource. The algorithm maintains two atomic counters. A process enters the bakery takes a ticket from the *ticket counter* and then increments it. The process must wait until the *current counter* displays the same number as the one on the ticket. When the two numbers match, the process can access the shared resource. When it is done with it, the process increments the current counter. It is easy to see that this algorithm provides mutual exclusion because only one thread can access the resource at the same [1].

[1]Assuming that all the operations performed are atomic

Chapter 3

Related Work

3.1 Static segmentation

Static segmentation [9] is an client-sided range locking algorithm that uses the one-sided operations provided by RDMA. Their idea is to segment the shared memory into segments at the start of the program. Each segment covers a small portion of the shared memory and will be associated with two instances of the Lamport's bakery algorithm. One is handling the read-only requests and the other is handling the write requests[1].
When a process wants to lock a range, it must lock every segment that covers the desired range. To lock a segment, the process first increments using an RDMA FAA either the read-only or the write ticket counter. Using the value returned from the FAA they know if the range is free or not. If the range is not free they issue a RDMA read on the counters until it can proceed. Every time the process find out the range is not free, it waits for a few microseconds. Finally, to unlock the range it increments the current counter associated with the ticket counter they first incremented.
Optimistically, the minimum number of operations to lock and unlock a range is 2 times the number of segments covering the desired range.

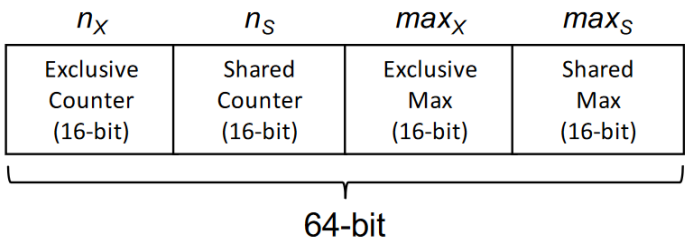

Figure 1: An image from the paper illustrating static segmentation's way to store counters.

[1] Each counter is contained in 2 bytes and the 4 counters from the two instances are packed into an 8 byte number which is optimal because RDMA atomic operation only support 8 byte numbers

3.2 Citron

Citron [6] is another client-sided range locking algorithm that uses the one-sided operations provided by RDMA. Citron maintains a fixed tree where each node contains 8 bytes of data, covers part of the shared memory where each child node contains a fourth of the range covered by its parent. That means that non-leaf nodes have exactly 4 children and that the root node contains the entire shared memory.
The idea is to exclude other processes by checking different conditions which when put together provide mutual exclusion. Each condition is a combination of which node in the tree the other process wants to lock[2] and when the other process wants to lock the node[3].

3.2.1 Detailed algorithm description

In internal nodes, there are two instances of the Lamport's bakery algorithm and a one-bit flag. One instance of the bakery algorithm, the T instance, keeps track of many processes are trying to lock the node concurrently and the other, the D instance, is used by the descendants of the node to notify that they are trying to lock a node. The one-bit flag is set to one if the node is currently locked. A node being locked means that the entire range covered by the node is locked by the process.
In leaf nodes, each of the 64 bits is a flag that are set to one if the process currently is owning the lock of the associated memory unit.

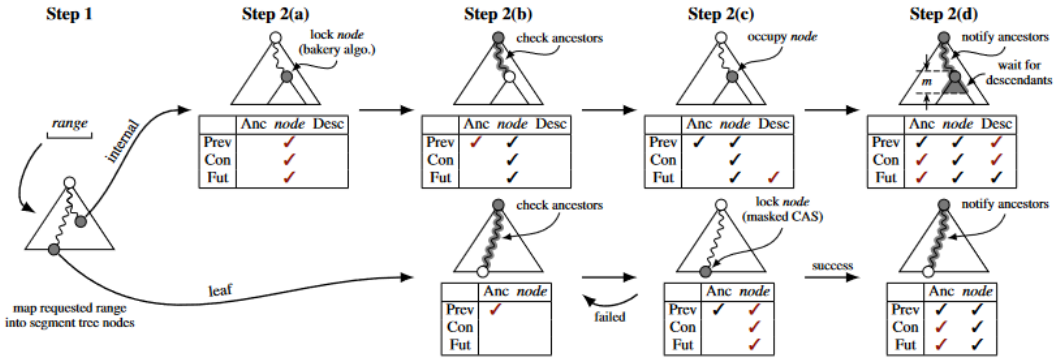

Figure 2: An image from the Citron paper illustrating all the cases to cover when locking a node.

The first step of the client is to split the range it wants to lock into k different nodes. This problem is solved by solving a tree knapsack problem. Then, the procedure to lock a node differs if the node is a leaf or not.
To lock an internal node the procedure is as follows:

[2] Either an ancestor, a descendant or the same node
[3] Is it concurrently, is the node already locked or will the node be locked in the future?

- Step 2a: Increment, using a masked FAA, T's ticket counter of the node we want to lock and wait until the current counter matches the number we obtained by doing RDMA reads.
- Step 2b: Wait until the ancestor nodes are unlocked by checking if the *occupied* flag is zero.
- Step 2c: Set the *occupied* bit of the node to lock to one.
- Step 2d: Using a masked FAA, increment the D ticket of some of the node ancestors. Those ancestors are spaced by m layers in the tree. Then it reads all of its descendants within m layers continuously. Reading is done when all of the polled nodes' D counter matches their D ticket. This makes sure that, if there is a descendant currently locking a node, there will be a collision between the notified ancestors and the polled sub-tree.

To lock a leaf node the procedure is as follows:

- Step 2b: Wait until all ancestor nodes are unlocked by issuing RDMA reads.
- Step 2c: Switch the bits of the associated memory units we want to lock using a masked CAS.
- Step 2d: Using a masked FAA, increment the D ticket of some of the node ancestors. Those ancestors are spaced by m layers in the tree.

Unlocking a node requires incrementing the D counter of every m ancestor and, depending if the node is a leaf or not, setting the bits previously changed to one in the leaf to zero or removing the occupied flag and incrementing the T counter.

Chapter 4

Design

Our design uses the idea of *combination* to implement a server-sided, distributed range lock algorithm.
In our design, only one machine can be the server. The server possesses the memory shared by every client. To manage this shared memory, it has a data structure that stores every lock that is currently taken. In a previous report that also focused on combination for range locks but in a local setting, the data structure that attained the best throughput was a red-black tree. This result was found using a nano-benchmark where read-only operations happen 99% of the time and write operations happen only the rest 1%. The ranges to be locked by the processes were completely random.
In order to work, our design needs at least two processes. One process, that from now on will be called the *combiner*, is responsible to gather all the lock or unlock requests and execute them sequentially and is the sole process that can modify the tree. All the other processes of the server, called *handlers*, are dedicated to handle incoming RPCs from the clients and create the associated lock and unlock requests.
In our designs we assumed a client can lock only one range at a time.

4.1 The client

A client sends an RPC request to lock a range and blocks until it has the confirmation that the range is acquired. The way the client obtains this information differs between our two different designs but the idea is similar. The client will poll the server to know if the operation is finished. This is where our two designs diverge, here's the two ways the client can poll:

1. The client polls using RPC requests and blocks until the response is received. If the lock is still not acquired, it polls again.

2. The client polls using RDMA reads on a dedicated byte in the memory of the server until the flag contained is one.

After being done with the lock, a client sends an RPC request to unlock the range and can *instantly* send a new RPC request to lock a range to increase the throughput.

4.2 Communication between client and server

We defined RPC requests to be 8 bytes long and RPC responses 4 bytes long. There are three different types of requests a client can create. To differentiate between them, the client sets a 2-bit header of the request as follows:

Bits	Mode
00	Lock mode
01	Unlock mode
10	Poll mode

Table 1: Different headers of the RPC requests and their signification.

The modes lock and unlock are used by both of our designs whereas the poll mode is only used by the design that polls using RPCs.
After the header, the structure of the request vary depending on the mode.

Lock mode	
Bits	Interpretation
63..62	Mode
61..42	Unused
41	Flip
40..33	Client id
32	Read only
31..16	Range end
15..0	Range start

Unlock mode	
Bits	Interpretation
63..62	Mode
61..9	Unused
8	Flip
7..0	Client id

Poll mode	
Bits	Interpretation
63..62	Mode
61..8	Unused
7..0	Client id

Tables 2 to 4: Content of an RPC request in function of its type.

This protocol assumes there is at most 2^{16} memory units managed by the server but this limit is easy to expand by changing the size of the fields *range start* and *range end*, which signify the included bounds of the range to be locked. The *read only* bit is one when the operation will not

modify the memory. *Flip* is a one bit value that changes each request. This value is used by the server to avoid data races because a client does not wait for the confirmation of the unlock of the range to send a new locking request.
The handlers usually return empty responses when their work is done. When a client is polling with eRPC, the last bit in the response will be set to one when the lock is acquired.

4.3 The server

4.3.1 Useful structures

Server threads must be able to pass information to each other. The following two structures facilitate achieving this goal.

Algorithm 1 Structure containing all the information of a request

```
struct RequestNode
    int rangeStart
    int rangeEnd
    int blockingRanges
    int owner
    bool readOnly
    bool lockOperation
    bool flip
    RequestNode* next
```

Here are the details of the fields of algorithm 1:

- *RangeStart* and *rangeEnd* are the bounds of the range to be locked.
- *BlockingRanges* stores the number of locks that do not permit the range to be acquired at the moment.
- *ReadOnly* indicates if the operation is read only.
- *LockOperation* differentiates between locks and unlocks.
- *Flip* is the flip value from the client's request.
- *Next* is used to form a linked list of request nodes that the combiner can easily exploit.

Here are the details of the fields of algorithm 2:

- *Tree* is an initially empty red-black tree that stores the locks currently taken, it associates a range locked to a RequestNode.

Algorithm 2 Data available to every server process

```
struct ServerSharedInfo
    RBTree tree
    RequestNode* head
    atomic RequestNode* tail
    RequestNode* locks[2 * NumClients]
    bool ready[NumClients]
```

- *Head* is the pointer to the first element of the linked list and is initially empty.
- *Tail* is an atomic pointer to the last element of the linked list.
- *Locks* is an array of two times the number of clients. This array is used to store the last lock request of each client. This information is used when unlocking to find the node's information because a client does not specify them again when sending an unlock request. The array can store two elements per client, to avoid data races because a unlock request and a lock request from the same client can be processed at the same time. If the handler handling the lock request is faster than the one handling the unlock request it can overwrite the value inside *locks* that the unlock procedure needs. To avoid that, a lock can be stored in two different slots depending on *flip*[1].
- *Ready*'s value which is associated with the id of a client is one when the last lock request received is complete. When the client is polling with RDMA, this is the memory that the clients will poll on. In the case the clients poll using eRPC, an handler will check on this value to determine its response.

4.3.2 The handlers

When receiving an RPC request that aims to lock or unlock a range, an available handler[2] puts it at the *tail* by performing an atomic exchange with the old *tail* and sets the *next* field of the old tail to the node. This effectively creates a linked list of nodes.
At the beginning of the program, the tail is *null* so a process that swapped its node with a *null* pointer knows it is the head of the linked list and so it also stores the node in *head* which will be used by the combiner.
Finally, in the case of a locking request, the handler resets the *ready* field and puts the lock in the *locks* array according to the *flip* bit that was received.

Algorithm 3 Functions executed by handlers

```
addRequest(SharedServerInfo* data, RequestNode* lock)
    RequestNode* prev = tree->tail.exchange(lock)
    if(prev != null):
        prev->next = lock
    else:
        data->head = lock

lock(SharedServerInfo* data, RequestNode* lock)
    data->ready[lock->owner] = 0
    data->locks[2 * lock->owner + flip] = lock
    addRequest(data, lock)

unlock(SharedServerInfo* data, RequestNode* lock)
    addRequest(data, lock)
```

Algorithm 4 Combiner's code

```
 1: RunCombiner(ServerSharedInfo* data):
 2:     RequestNode* currentNode = null
 3:     while(data->head == null):
 4:         wait()
 5:     currentNode = data->head
 6:     while(true):
 7:         if(currentNode->lockOperation):
 8:             currentNode->blockingRanges = CountBlockingRanges(data->tree, currentNode)
 9:             data->tree.insertNode(currentNode)
10:             if(currentNode->blockingRanges == 0):
11:                 data->ready[currentNode->owner] = 1
12:         else:
13:             RequestNode* nodeToBeUnlocked = data->locks[2 * currentNode->owner + flip]
14:             data->tree.removeNode(nodeToBeUnlocked)
15:             DecrementBlockingRanges(data->tree, currentNode)
16:         while(currentNode->next == null):
17:             wait()
18:         currentNode = currentNode->next
```

4.3.3 The combiner

Algorithm description

The process that runs the combiner and does nothing else. At first, the combiner waits until the first node has been put it the linked list. It is able to notice that because the *head* pointer is not *null* anymore (lines 3-4).
After setting the head of the linked list as the current node (line 5), the combiner enters a while loop and will stay inside it until the end of the program. If the current node is a locking operation, the combiner:

- Counts how many ranges overlap with the range to be locked[3] (line 8).
- Inserts the range associated with the node inside the tree (line 9).
- Sets *ready* to one if no existing range in the tree blocks the range to be acquired so that the handlers or the clients[4] have the information that the request is complete (lines 10-11).

If the current node is an unlocking operation instead then the combiner:

- Gets the node from *locks* that was previously inserted in the tree and remove it from the tree (lines 13-14).
- Decrements the blocking ranges by one of other ranges that overlap with range to be unlocked (line 15).

After executing one operation, the combiner waits until a new node is in the linked list, stores it and the loops restarts.

[1] Since *flip* alternates between zero and one at every operation of a given client, the lock data can never be overwritten

[2] Which is selected by eRPC.

[3] Read-only operations already present in the tree overlapping with a new read-only operations will not be counted in the blocked ranges counter because these ranges can be locked in parallel.

[4] Depending if the clients poll using eRPC or RDMA.

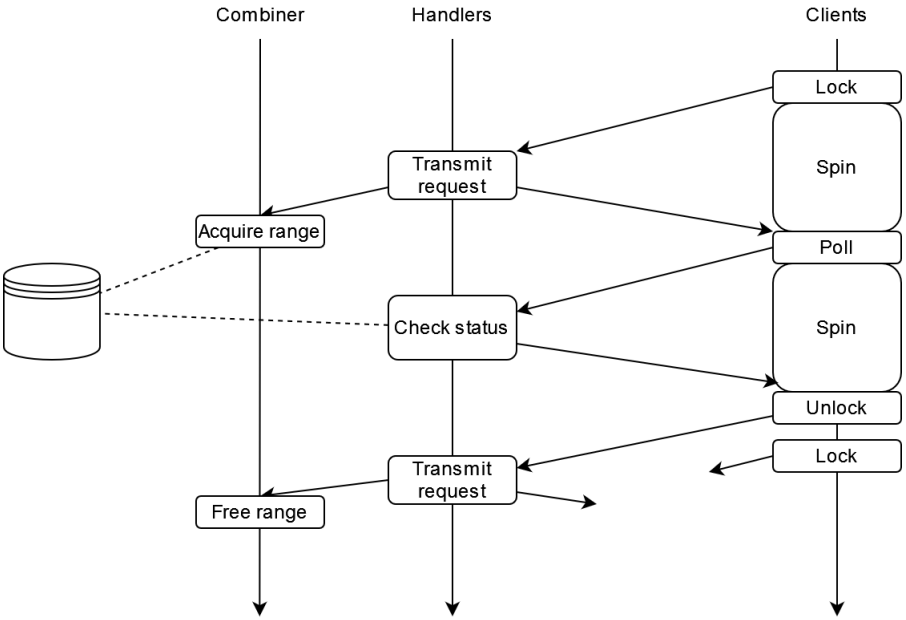

Figure 3: Diagram summarizing the flow of the combiner polling with RPCs. [5]

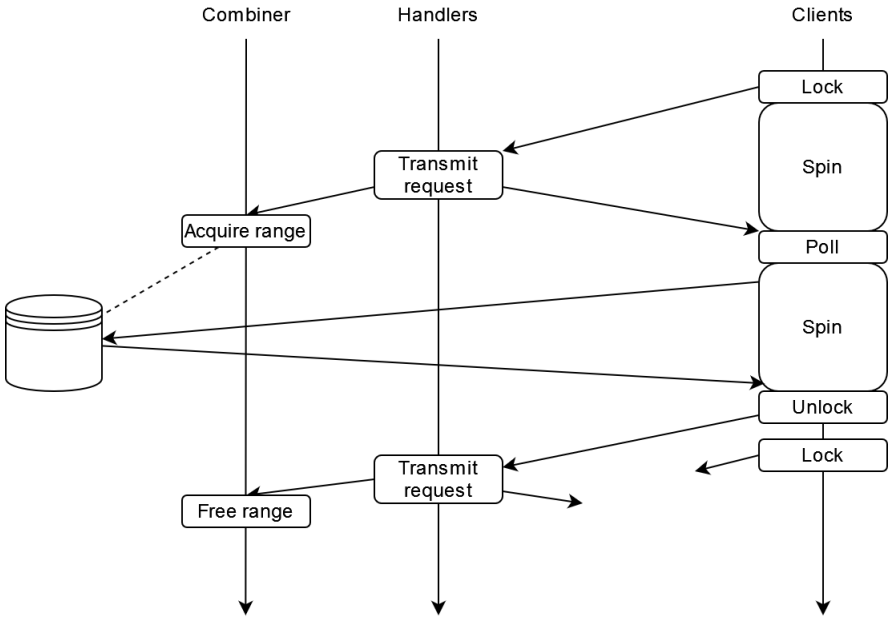

Figure 4: Diagram summarizing the flow of the combiner polling with RDMA. [6]

[5]The database symbol represents the table storing the status of the locking procedure of every process.
[6]See previous footnote.

Chapter 5

Evaluation

5.1 The setup

In every benchmark, we compared two types of implementations. The first contains the server-sided locking algorithms which we ran using eRPC. Here are the implementations that follow this paradigm:

- Interval tree: a simple red-black tree stores all the ranges. All server threads can access it but must acquire a shared lock before modifying it.
- Combiner where the client is polling using eRPC as described in section 4.
- Combiner where the client is polling using RDMA as described in section 4.

Both combiners run with 18 server threads. The interval tree runs only with one server thread. Experiments have shown that passing the tree to multiple processes causes a lot of time spent doing cache coherence. Out of 1,2,4,8 and 18 server processes, the best throughput was achieved consistently when the server had only one server thread. That means that the time spent on cache coherence is greater than the time lost on queuing.

The second type of implementations are where only the clients are active while the server does nothing apart from hosting the memory. All implementations in this category use one-sided RDMA operations to communicate with the server:

- Static segmentation as described in section 3.1. The code was available online [1] but unfortunately used the RDMA experimental verbs. To use those, it is required to downgrade the OFED driver into an anterior version than the first version supported by our NIC. We thus had to re-implement this reference implementation ourselves.

- Citron as described in section 3.2. The code wasn't made public so we had to re-implement it ourselves. Further more, the algorithm described in the paper also used experimental verbs, especially *masked compare-and-swap* which is required to lock part of leaf nodes. Since we couldn't implement it that way, we decided that all nodes were considered internal nodes. This decision effectively downgrades the algorithm because we now have to lock an entire node even if we want to only lock a part of it.

The RDMA library, which we later modified, was found in the github repository of DrTM[8].

Each machine we used for our benchmarks were equipped with an Inter Xeon Platinum 8468 running at 2.101 GHz, contained 2 NUMA nodes that each contain 48 cores. The operating system was Ubuntu 22.04 with a linux kernel 5.15.0-84-generic. An Nvidia Bluefield-2, a Mellanox ConnectX-6 NIC, is installed which enables RDMA in our system.

Every implementation presented here is available at [2].

5.2 Benchmarks

5.2.1 Network latency

We first show the latency of our network.

Operation	**50% latency [us]**	**99% latency [us]**
eRPC	3.52	4.18
RDMA read	2.74	2.86
RDMA atomic	2.79	3.29

Table 5: 50th and 99th latency percentile of network operations

We obtained these numbers by:

- For eRPC, we sent a 8-byte RPC request and received a 4-byte response, which are the same size that we used in all our designs that depend on eRPC to communicate. The time displayed is the time between sending the request and receiving the response.
- For RDMA, we obtained these numbers by using the commands `ib_read_lat` and `ib_atomic_lat` provided by the infiniband fabric utilities. The value for both *read* and *atomic* has been obtained from reading a 8-byte value. Atomic represents both *compare-and-swap* and *fetch-and-add*. We did not measure the latency of other verbs because we did not use other RDMA verbs in our implementations.

5.2.2 Micro-benchmark

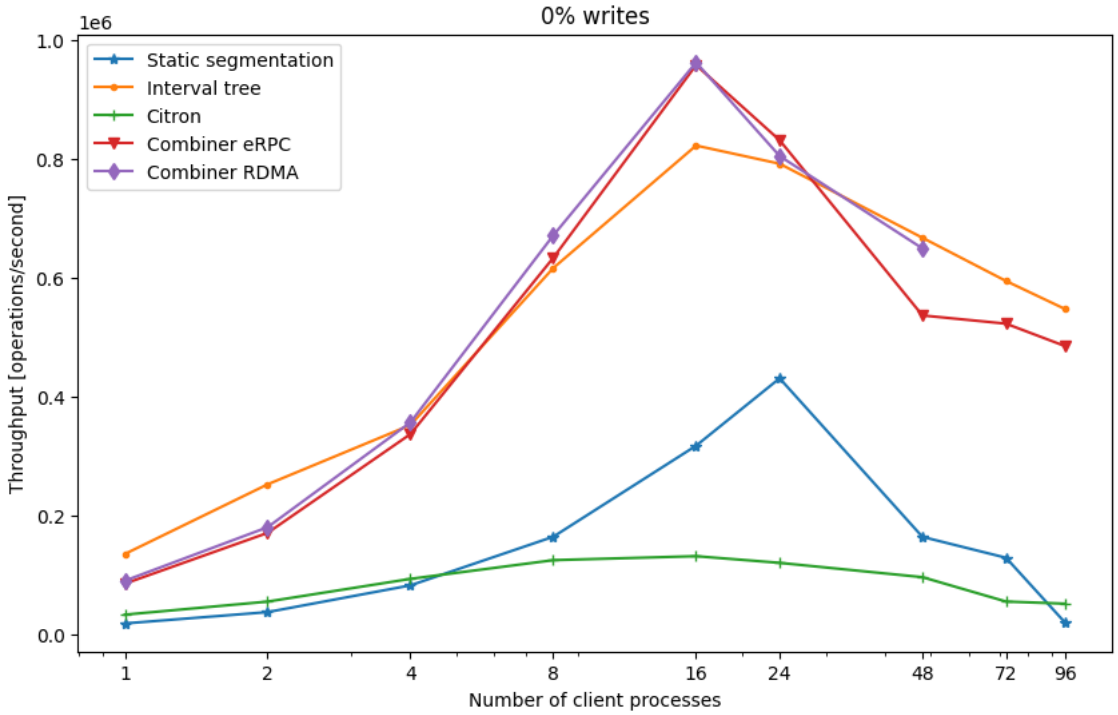

Figure 5: Throughput with 0% writes in function of the number of clients running.

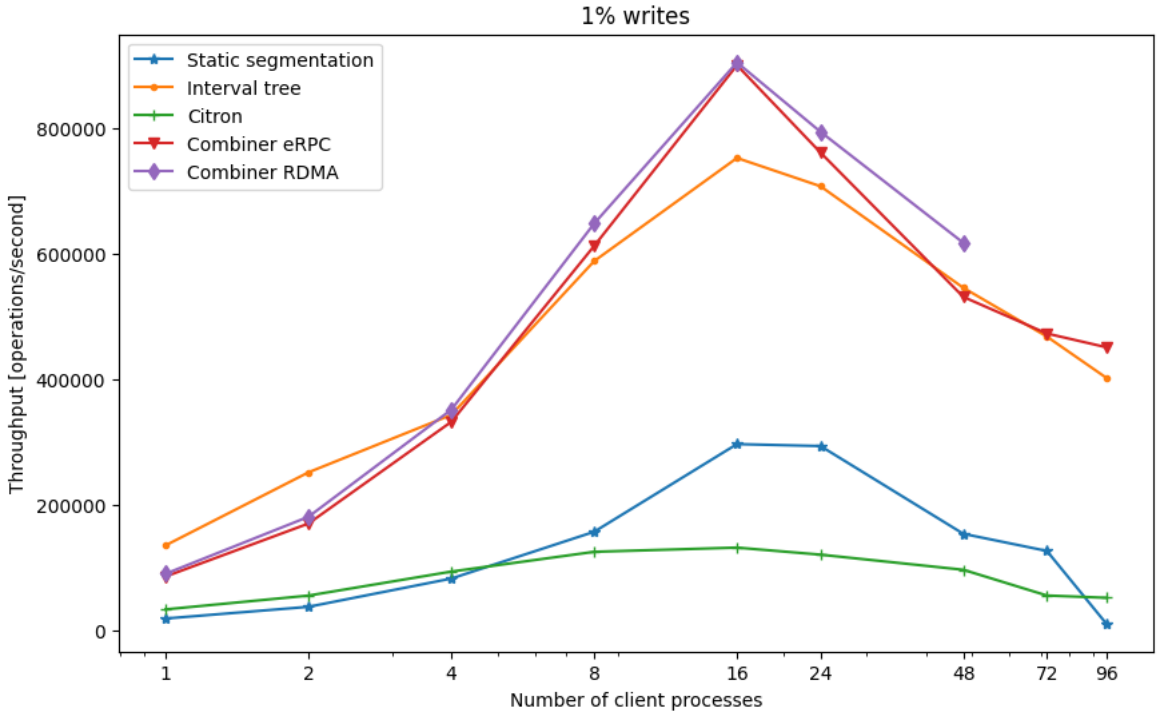

Figure 6: Throughput with 1% writes in function of the number of clients running.

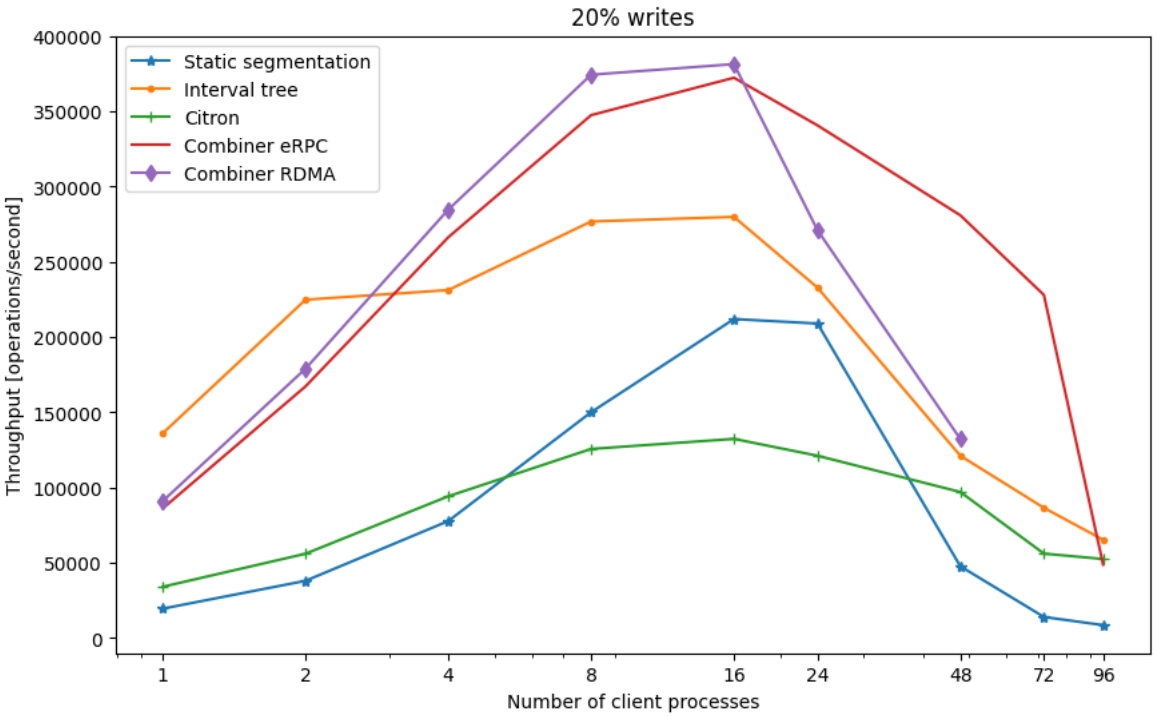

Figure 7: Throughput with 20% writes in function of the number of clients running.

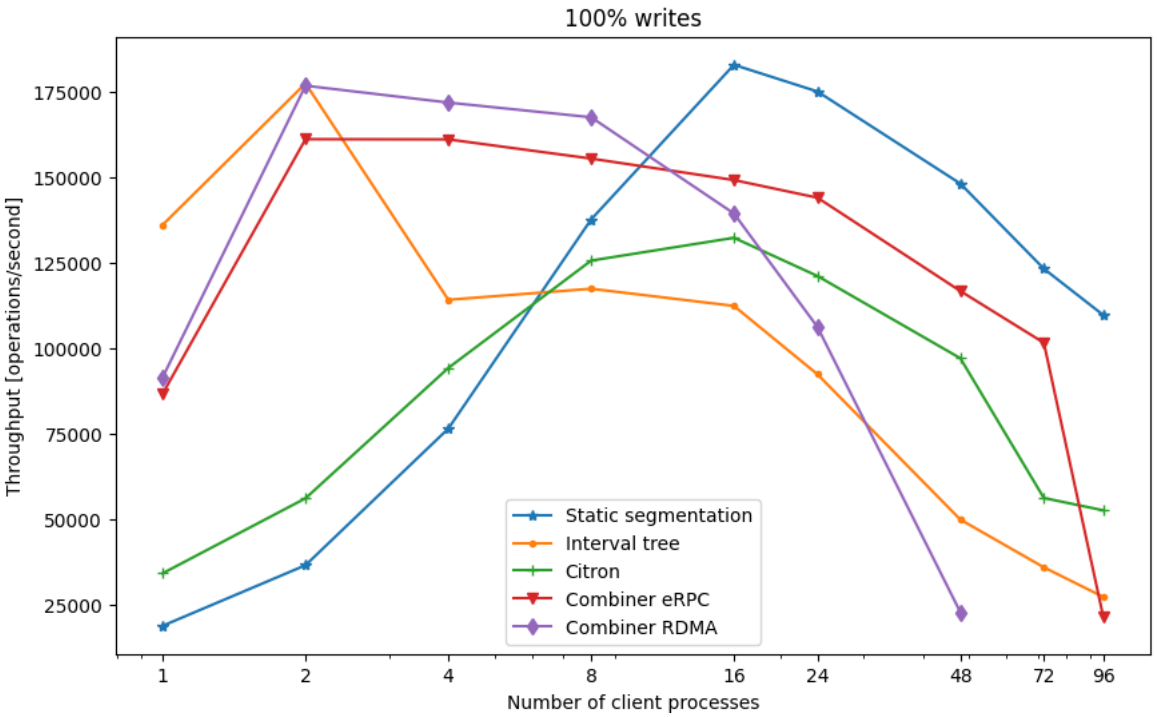

Figure 8: Throughput with 100% writes in function of the number of clients running.

The total number of memory units in the system was 2^{16}. Every implementation except static segmentation acquired a random sized range at a random point of the shared memory. We decided that static segmentation would at most lock 8 different segments of memory[1], also we arbitrarily set a segment's size to 2^8 memory units.
In implementations that communicate using eRPC, when the total number of clients is smaller or equal to the number of cores in one NUMA node, every client was hosted by the same node. In the other case, the clients were split in halves in the two nodes of the machine.
Also note that the combiner polling with RDMA is missing the values from 72 and 96 clients because of connectivity issues with RMDA. We had to run two instances of the program on the same machine, because eRPC allows programs to run on only one NUMA node, which causes issues with our RDMA library.

Citron does not differentiate between read and write operations, this is why the its results are the same in the four diagrams. Let's also repeat that it does not work at its full potential because of the missing experimental verbs. As the paper recommends, we batch some operations together[2] to reduce the number of RDMA round trips. Each batch of operations would supposedly be executed in parallel. We managed to reduce to two round trips for a lock operation however the median latency for locking when there is only one client is around 25 microseconds, which is higher than the expected two round trips of around 6 microseconds we found in 5.2.1. By measuring how long it takes to perform batched RDMA operations we notice that each additional RDMA operation performed beyond the first adds a smaller cost than first. We suppose this is because the remote NIC must still perform the queued requests from the client sequentially. This hypothesis would explain why the latency is a bit lower: the requests are sent in parallel but the NIC still executes them sequentially.
Also our implementation for Citron tends to starve some processes when there is a lot of clients, which affects the throughput negatively. The results displayed here[3] were measured by reducing the total runtime of the stress test to reduce the probability of starvation.

[1] If we let the range size to be random, from 16 clients onwards, the throughput would be close to zero because of the very large number of segments to lock and the contention between clients.
[2] By posting multiple work requests before polling the completion queue.
[3] When starvation does not occur.

0% writes

When there are no write operations in the system, the server-sided range locks perform better than the client-sided ones. Interval tree performs better than combination when there is only one or two clients. This is because the handler executes the operation and transmits the results directly to the client while with both combiners, the handler transmits the request to the combiner and directly answers the client. This causes extra latency because the client must poll at least once. Combination polling with RDMA has a slightly better throughput than the design polling with eRPC. This is to be expected because the latency of RDMA is lower than the latency of eRPC.
Every implementation peaks at around 16 to 24 clients processes. Static segmentation must lock a high number of segments to lock one range, thus it must do a lot of RDMA FAA, which increases the latency. Finally Citron must execute a lot of RDMA operations and, as was previously mentioned, even if they are batched they are not completely executed in parallel, which increases the latency.

1% writes

When there is only 1% of write operations in the system, the results are similar compared to the 0% write scenario. The difference is that the interval tree performs worse when there is a lot of clients processes running because now clients start to poll which reduces the server resources allocated for handling new locking requests.

20% writes

When there is 20% of write operations in the system, the server-sided range locks perform still better than the client-sided ones but their advance shrinks. The trend of the interval tree we noticed at 1% writes continues. Also both combiners' throughput drop with a high client count and the one polling with RDMA handles contention worse than the combiner polling with eRPC. Static segmentation is less affected by the contention's rise and Citron is unaffected.

100% writes

Finally when there is only write operations, no type of implementation performs clearly better than the other.
Interval tree is now almost always answering poll requests form the clients, which reduces its throughput, and its sole server process cannot keep up with all the requests coming in. This increases the latency due to the higher queuing time.
For static segmentation, we notice that for a high number of clients, 100% writes yields better

throughput than 20% writes. After investigating, we found that the backoff time when there is a conflict is equal to the number of writers waiting multiplied by a constant time plus the number of reader waiting multiplied by a constant time[4]. This is designed to let the readers retry faster than the writers. When removing this advantage for the readers, the throughput for 100% writes with a high number of clients reverts to very low performance, which we would expect from such contention.
Combination with RDMA performs worse than combination with eRPC. We hypothesize this is because the data is too close in the memory (the flags for two processes are separated by only one byte), future work can investigate the importance of cache coherence and how to store the memory more efficiently.

[4]This part of the sum is only counted when the waiting operation is a writer

5.2.3 Filebench OLTP

We ran a benchmark that had a similar workload than OLTP from Filebench[3]. This workload separates the memory, which also contains 2^{16} memory units, into 8 data files and one log file. All of these files are of equal length. We ran it on two machines, one is the server and the other the client. The clients ran 48 processes concurrently, all of them contained inside the same NUMA node. The client processes could either be a *reader*, a *writer* or a *log writer*.
Out of the 48 client processes, 39 of them are readers. A reader performs random read-only operations on data files, the ranges were always the same: 2^6 memory units.
9 client processes are writers. Whenever reader processes perform 1000 reads, each writer process performs 100 write operations on the data files. The writers lock sequentially each data file to even out the traffic on the memory. Inside a data file, a writer lock a random range of 2^6 memory units.
The workload contains only one log writer, it performs one lock operation on the log file for every 3200 reads. The start of the range inside the log file is random and its size is 2^{11} memory units.

Lock design	**Throughput [ops/sec]**	**50% latency [us]**	**99% latency [us]**
Combiner eRPC	286491	156.8	182.4
Combiner RDMA	147576	264.0	592.0
Interval tree	124603	366.4	403.2
Static segmentation	161530	150.1	232.8
Citron	162720	26.1	223.4

Table 6: Throughput, 50th and 99th latency percentile of the different implementations with the OLTP benchmark.

The combiner polling with eRPC achieved the best throughput of all the implementations tested which is due to its low tail latency.
Combination with RDMA performs worse than combiner with eRPC in every aspect. This supports our previous observation that under high contention, this design performs worse than others.
Interval tree's latency is also relatively constant but having only one server thread and having to acquire a lock on the tree severely increases the latency.
Citron has the lowest median latency compared to the others but suffers from high tail latency. We can see that Citron is very efficient at locking small ranges thanks to its tree structure but is prone to starvation under high contention.
Static segmentation performs better compared to the micro-benchmark because it has to lock only two segments to acquire a range compared to eight before. Its median latency is higher than Citron's and its 99% latency is the same than Citron. That means that its tail latency is lower than Citron's.

Chapter 6

Conclusion

In this report, we presented two designs of distributed range locks in which the server is active in the process of acquiring locks. Both of these used the principle of combination. The first design made the clients poll until acquisition of the range using RDMA. This design is not efficient under high contention but performs well when contention is low. The second design we proposed had the clients poll with RPCs. This design had 1.76x more throughput than client-sided locking algorithms when running a realistic benchmark.
Future work should investigate the memory layout of the memory exposed by RDMA because in our implementations we did not add padding between structures in the memory. This would explain why implementations using RDMA perform worse the implementations using RPCs under high contention because of the false sharing caused by having structures too close in memory. Also in our experiments we used 18 server processes for the combiners to replicate the experiment in the Citron paper, future experiments should modify this number to saturate the server with requests to compare properly server-sided and client-sided range locks. Another interesting evolution would be to offload the memory stored by the server directly into a Smart-NIC. This would reduce latency even more because it would remove data transfers between the remote NIC and the remote memory.

Bibliography

[1] URL: https://github.com/dongyoungy/rdma_2pc.

[2] URL: https://github.com/rs3lab/range-lock/tree/distributed.

[3] URL: https://github.com/filebench/filebench.

[4] Jeremy Dezalos. "Scalable range locking using combination and delegation". In: 2023. URL: https://github.com/rs3lab/range-lock/blob/distributed/doc/Master_semester_project.pdf.

[5] Panagiota Fatourou and Nikolaos D. Kallimanis. "Revisiting the combining synchronization technique". In: *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '12. New Orleans, Louisiana, USA: Association for Computing Machinery, 2012, pp. 257–266. ISBN: 9781450311601. DOI: 10.1145/2145816.2145849. URL: https://doi.org/10.1145/2145816.2145849.

[6] Jian Gao, Youyou Lu, Minhui Xie, Qing Wang, and Jiwu Shu. "Citron: Distributed Range Lock Management with One-sided RDMA". In: *21st USENIX Conference on File and Storage Technologies (FAST 23)*. Santa Clara, CA: USENIX Association, 2023, pp. 297–314. ISBN: 978-1-939133-32-8. URL: https://www.usenix.org/conference/fast23/presentation/gao.

[7] Anuj Kalia, Michael Kaminsky, and David G. Andersen. "Datacenter RPCs can be general and fast". In: *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*. NSDI'19. Boston, MA, USA: USENIX Association, 2019, pp. 1–16. ISBN: 9781931971492.

[8] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. "Fast in-memory transaction processing using RDMA and HTM". In: *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP '15. Monterey, California: Association for Computing Machinery, 2015, pp. 87–104. ISBN: 9781450338349. DOI: 10.1145/2815400.2815419. URL: https://doi.org/10.1145/2815400.2815419.

[9] Dong Young Yoon, Mosharaf Chowdhury, and Barzan Mozafari. "Distributed Lock Management with RDMA: Decentralization without Starvation". In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. Houston, TX, USA: Association for Computing Machinery, 2018, pp. 1571–1586. ISBN: 9781450347037. DOI: 10.1145/3183713.3196890. URL: https://doi.org/10.1145/3183713.3196890.