RS3LAB

École Polytechnique Fédérale de Lausanne

Scalable range locking using combination and delegation

by Jeremy Dezalos

Master Semester Project Report

Prof. Dr. Sanidhya Kashyap
Thesis Advisor

Vishal Gupta
Thesis Supervisor

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

January 18, 2024

Abstract

In the context of concurrency, range locks are a synchronization tool that protect a section of the memory shared by multiple processes and assign it to a single thread. Because every process must pass through the same procedure in order to acquire a part of the memory, range locks have the possibility to be a massive contention point and effort should be invested to reduce it. Until now, range locks have only been implemented by letting all processes lock and free the memory. In this report, we present two techniques to implement range locks: combination and delegation. Their common point is that only one process is allowed to acquire and free range locks while the others can only submit requests to them. We achieved up to a 3x speedup compared to other implementations that do not use one of these two techniques.

Contents

Chapter 1

Introduction

Modern computers are now all equipped with multiple processors to keep increasing their speed to compensate for our inability to create processors that run at a higher frequency. This new trick enabled those computers to perform better than their predecessors. However this had a cost: processors could not be left to their own devices lest they would interfere with each other by reading protected, old or invalid values in the memory. This setup of multiple processors running in concurrency has to be strictly controlled to make sure every process runs on valid input, while maintaining a high speed.

Range locks are a synchronization structure that provides concurrent access to different regions in memory shared by different processors. The goal of this structure is to enable fine-grained access to the memory for the processors without making processes wait for each other if they do not use the same memory regions. Thus, if two processors want to lock disjoint parts of their shared memory, the range lock will let them proceed. However, if two processors want to lock ranges that overlap then one of them will have to spin until the range has been freed by the processor that acquired the range. Making one thread wait is unavoidable as the memory locked is considered dirty and cannot be modified because when a thread locks a resource it assumes that no other processor will interact with it during the critical section.

However, creating range locks that are fair, efficient and scalable is far from trivial. The main problem to solve is how to manage the data structure that will hold the information necessary for the range lock and how the different processors will access it. Contention on the lock is not the only factor that slows down the execution of the program, contention on the data structure must be kept to a minimum to reduce the time processors spend spinning to access it.

Until now, implementing range locks has only been done by allowing every processor to share and modify the concurrent data structure all at the same time such as [1], which proposed a new data structure to store the locked ranges. However, new ways of sharing this data structure are rarely explored.

This does not mean that research of new ways to execute critical operations efficiently has never been done. [2] introduces the idea of a *combiner*. A combiner in the context of concurrency is a special process that receives requests from all the other processes running concurrently and executes them sequentially. The combiner runs until no requests are left or when it executed a fixed number of operation and then finally stops which gives a chance for another processor to become the combiner. [4] goes a step further by introducing *delegation*. This time, a single will execute all the operations requested by the processors in the system, the *delegate*, and uses the fact that CPU cores in a same socket shares a L1 cache to execute the requests coming from the same socket together.

In this paper, we propose the following thesis: range locking is faster using a *combiner* or a *delegate*. We verify our thesis by adapting the concepts of the combiner and delegation to the context of range locks by using two different data structures: a range tree and a linked list. We then benchmark and compare our designs with existing range lock implementations.

Our designs managed to perform up to a 3x speedup compared to classical range lock designs. The fastest combination of design and data structure depends on the number of writes done by the processors.

Chapter 2

Background

2.1 Critical sections

In concurrent programming, a *critical section* is a block of instructions a process executes that, usually, modifies memory shared with other processors. If multiple processors execute the same critical section concurrently, then the program becomes susceptible to data races and eventually the memory becomes corrupt. This is why it is primordial to make sure only one processor can access a critical section at a time. Range locks are a way to handle this problem by excluding all but one process to access specific chunks of the memory. However one could ask themselves...

2.1.1 Why not simply lock the whole memory ?

Locking the whole structure so that only one process can modify the memory at a time, more commonly called *coarse-grain locking* has the particularity to be extremely easy to implement: make the processes compete for the lock using a compare-and-swap and you are done. Unfortunately it is as fast as having only one process in the system: the first process to obtain the lock executes the critical section and frees the lock and only then another process can obtain the lock. It is easy to see from this simple example that only one process can work at a time. So the number of processes in the system does make the program scale when using coarse-grain locking.

2.2 Computer architecture

The next important concept to address is computer architecture and more specifically how to take into account the architecture of a computer in order to extract the best performance for our

program.

2.2.1 Cache

Reading or writing on the main memory takes time. Caches are widely used to have faster reads or writes. The principle of a cache is the following: when reading data from the memory, the line of memory where the data resides is copied into the cache. This line will stay for a time, depending on the cache policy, and will be copied back into the memory when leaving the cache. The point of introducing a cache to the architecture is that it greatly shortens the time needed to read a line that is already stored in a cache.

It is possible to have multiple processors share a single socket. This means those processors sharing the socket also share the same cache. The downside is that they possess less cache space as they all share the same space but, if they operate on the same memory regions, then much less time is needed to read it.

Chapter 3

Related Work

Scalable Range Locks for Scalable Address Spaces and Beyond [1], published in 2020, presents a new alternative for range locking in the Linux kernel. Their design, consisting of storing ranges in a linked list, contrasts with the existing implementation of range locks in the kernel which uses red-black trees. `mmap_sem` is considered as "one of the most intractable contention points in the memory-management subsystem". Range locks must be as efficient as possible to not cause further contention. They present a reader-writer range lock which performed better than the one present in the kernel regardless of the write percentage and if threads lock disjoint ranges or not.

Revisiting the Combining Synchronization Technique [2], published in 2012, explores the concept of *combining*. The idea is the following: only one thread can execute operations at a time on the whole shared memory. The difference between this technique and a coarse lock affecting the entire shared memory is that the combiner not only executes its own request but the requests of every process present in the system. They find that using a combiner is faster than using fine-grained thread synchronization because threads spend less CPU cycles stalling and have less cache misses.

ffwd: delegation is (much) faster than you think [4], published in 2017, takes the idea of combining one step further. In [2], the process being the combiner changes during the execution of the program whereas in [4] the delegate does not change. The most important change from [2] is that [4] takes into account the structure of the memory, in particular the CPU sockets. They proposed an algorithm that operates over one CPU socket at a time. This tends to minimize the amount of cache coherence on the interconnect. The size of the requests and responses are adapted to the size of the cache lines which lowers false sharing. Finally, by making all processes in the same socket share response, cache coherence within the socket is also reduced.

Chapter 4

Design

In this section, we present our two algorithms and describe them, one implementing combining and the other implementing delegation in the context of range locking.

4.1 Combiner based range locking

4.1.1 Algorithms

Algorithm 1.1: Locking request

```
   shared Node *Tail = NULL
   lock(int start, int size):
 1: Node *myNode, xchgdNode
 2: myNode->wait = true
 3: myNode->completed = false
 4: myNode->start = start
 5: myNode->end = start + size
 6: myNode->next = NULL
 7: xchgdNode = swap(Tail, myNode)
 8: if xchgdNode != NULL then
 9:     xchgdNode->next = myNode
10:     while myNode->wait do
11:         nop
12:     end while
13:     if myNode->completed then
14:         return
15:     else
```

```
16:         combiner(myNode)
17:         return
18:     end if
19: else
20:     combiner(myNode)
21:     return
22: end if
```

Algorithm 1.2: Freeing request

```
free(Node *node)
1:  addNodeToReleaseBuffer(node)
2:  if Tail == NULL then
3:      if CAS(Tail, NULL, node) == NULL then
4:          executeFreeRequests()
5:          Node* nextNode = node->next
6:          if nextNode == NULL then
7:              if CAS(Tail, node, NULL) == node then
8:                  return
9:              end if
10:             while nextNode == NULL do
11:                 nextNode = node->next
12:                 nop
13:             end while
14:         end if
15:         nextNode->wait = false
16:     end if
17: end if
```

Algorithm 1.3: The combiner

```
counter = 0
combiner(Node *node):
1:  while true do
2:      counter++
3:      executeFreeRequests()
4:      insertNode()
5:      blocking ranges = countBlockingRanges()
6:      if blocking ranges == 0 then
7:          node->completed = true
8:          node->wait = false
```

```
 9:      end if
10:      if node->next == NULL || node->next->next == NULL || counter >= h then
11:          break
12:      end if
13:      node = node->next
14:  end while
15:  Node *nextNode = node->next
16:  if nextNode == NULL then
17:      if CAS(Tail, node, NULL) == node then
18:          return
19:      end if
20:      while nextNode == NULL do
21:          nextNode = node->next
22:          nop
23:      end while
24:      nextNode->wait = false
25:  end if
```

4.1.2 Acquiring a range

The following section describes algorithm 1.1

To design a range lock using the combiner idea, we first started from the algorithm *DSM-synch* presented in [2]. Processes that want to acquire a range lock write their request in a structure called *Node* (lines 2-6). Then, to implement our combiner idea, they will compete to obtain the combiner role by atomically swapping this Node with a shared object called *Tail* (line 7). Tail is originally set to NULL when no combiner is running, so a process that obtains NULL from Tail knows it is now the new combiner (lines 8+20). Processes that did not succeed in obtaining the combiner role deposit their request in the *next* field of the Node they obtained (line 9).

By first swapping their node to obtain another and then putting a pointer of their original node inside it, processes manage to create a linked list of all requests pending that the combiner must execute. They will then spin on the *wait* field of their Node until it changes to *false* (line 10). The *completed* field indicates that either the combiner executed their request and they are free to return or that the combiner stopped before executing the request and they must become the new combiner (lines 13+16).

4.1.3 Freeing a range

The following section describes algorithm 1.2

Freeing a range is non blocking operation, meaning that a process will never spin on a condition. A process that wants to free a range stores the lock they acquired previously in a *release buffer* (line 1) and they try to acquire they combiner role using a compare and swap (line 3). If they do not succeed, they return without waiting the completion of their request. However if they succeed, they execute all the requests that free ranges (line 4). The process must then reset the Tail to NULL or give the combiner to a node that is waiting. In the case there is no node waiting (line 6), the process changes the Tail back to NULL using a compare and swap (line 7) and return if successful (line 8). If some process is appending a node, we wait until it finishes. (lines 10-13). Finally in the case there is a node waiting, we change its *wait* field to false to signal that it is the new combiner (line 15)

4.1.4 The combiner

The following section describes algorithm 1.3

The combiner is in charge to execute all requests regardless if they are locking or freeing operations. Since we made sure the combiner is the sole process executing the code, it can execute the requests sequentially.

Each time the combiner executes its loop (line 1), it increments by one a counter to keep track how long it has been running (line 2). To lock a range, the combiner first inserts the new range in its data structure holding all locks taken by the different processes (line 4). After inserting, it checks if there are ranges that are already inside the data structure that overlap the new range. For each overlapping range, the combiner increments by 1 a counter called *blocking ranges* (line 5). We note that our algorithm takes into account if the locking operation is for a read operation or not. When counting the blocking ranges if a read lock overlaps with another read lock then it is not counted. If this counter is still at zero after scanning the data structure (line 6), the new lock does not overlap with any other range and the locking operation is considered as complete. To signal this information to the waiting process, it sets the *wait* and *completed* field of the waiting process to false and true, respectively (lines 7-8).

Between each locking operation, the combiner will check if there are freeing requests pending and if that is the case, it will execute them all at once (line 3). We do this to not delay the acquisition of ranges that could be acquired if the free requests were to be executed. To free a range, the combiner takes the next request in the *release buffer*, deletes it from the data structure and decrements the counter *blocking ranges* to all the ranges inside that overlapped with it.

The combiner will keep executing requests until one or more of the three following conditions are met (line 10).

1. The next request in queue is NULL.
2. The second next request in queue is NULL.

3. The combiner has already executed a fixed number of operations.

The first condition is trivial to explain: if there are no more requests to execute, the combiner is not useful anymore. The second condition forbids on accessing a node whose next field has not yet been updated. Finally, the third condition forces the combiner process to stop to allow it to propose other requests, creating a fairer algorithm.

If none of these conditions are met, the combiner moves on to the next request (line 13).

The method the combiner uses the same algorithm described in algorithm 2.2 to transfer its role (lines 16-25) and has already been described in section 4.1.3

Data structure containing the ranges

We remark that until here, the data structure meant to hold the information about the locks taken was never explained in detail. This is because any data structure can fit inside our algorithm. In our case, we implemented an interval tree and a linked list. The tree automatically balances itself and doesn't need anything else from our part. To store a range in a linked-list, we decided to store them by sorting them by the beginning of their range. This choice makes the algorithm of counting and decrementing the blocking ranges faster because we don't have to scan the entire list until the end to find overlapping ranges compared to populating the list without any order.

What will differ from these two data structures when benchmarking is how they support concurrent accesses and how fast they can find overlapping ranges.

4.2 Delegation based range locking

To implement delegation based range locking, we combined the ideas proposed by [2] and [4]. The principal difference between our design and [4] is that the *delegate*, the process responsible to execute all requests, will change during the execution of the program to give all processes the opportunity to propose requests. This contrasts with [4] that decided to keep the same delegate throughout the execution of the program.

In this system, processes manipulate two types of structures *Requests* and *Responses*. A process can enter its request in a zone in memory that is dedicated for it. The delegate will then read the request, execute it and then write the necessary information in the response, which the waiting will read to obtain its answer. *Toggle bits* are present in both structures *Request* and *Response*. The way for processes to notify the delegate they are waiting is to flip the toggle bit in the request so that the toggle bits in the request and the responses associated to the process will be different.

To reduce inter-socket delay when performing IO operations on multiple CPUs, the delegate will iterate over the CPU sockets and execute only requests created by processes inside the current socket. We also coalesced all responses in a socket into one 128-byte response shared by all processes in the same socket: toggle bits become a single integer where each bit corresponds to a toggle bit for one process. Sharing a single response reduces the amount of cache coherence. Each request is 64 bytes, which is exactly one cache line. When fetching one request, the next 64 bytes in the memory are pre-fetched. We use this to our advantage by storing every request in a single block in memory.

4.2.1 Algorithms

Algorithm 2.1: Request creation

```
shared int shared object = -1
Request request
Response response
lock(start, size, reader):
1:  requests[cpuid]->start = start
2:  requests[cpuid]->last = start + size
3:  requests[cpuid]->reader = reader
4:  requests[cpuid]->toggle = 1 - request->toggle
5:  if CAS(shared object, -1, cpuid) == -1 then
6:      delegate()
7:  else
8:      while requests[cpuid]->toggle != getMyBit(response->toggle) do
9:          nop
10:     end while
11:     if getMyBit(response->combiner) then
12:         getMyBit(response->combiner) = false
13:         delegate()
14:     end if
15: end if
```

Algorithm 2.2: The delegate

```
bool answerIsReady[numberOfCores] = false
bool wait[numberOfCores] = false
int currentSocket
counter = 0
delegate():
1:  while true do
```

```
2:      i = processId foreach process in current socket
3:      if requests[i]->toggle == getBit(response->toggle, i) || wait[i] then
4:          continue
5:      end if
6:      counter++
7:      wait[i] = true
8:      if locking request then
9:          insertNode()
10:         blocking ranges = countBlockingRanges()
11:         if blocking ranges == 0 then
12:             answerIsReady[i] = true
13:         end if
14:     else
15:         deleteNode()
16:         decrementBlockingRanges()
17:         answerIsReady[i] = true
18:     end if
19:     end foreach
20:     i = processId foreach process in current socket
21:     if answerIsReady[i] == false then
22:         continue
23:     end if
24:     wait[i] = false
25:     requests[i] = newRangeLock
26:     answerIsReady[i] = false
27:     end foreach
28:     changeToggleBitsInResponse()
29:     if counter >= MaxRequests || checkIfNoNewRequests() then
30:         break
31:     end if
32:     currentSocket = currentSocket++ mod numberOfSockets
33: end while
34: i = processId foreach process in the system
35: if answerIsReady[i] then
36:     wait[i] = false
37:     requests[i] = newRangeLock
38:     answerisReady[i] = false
39:     if i mod coresPerSocket == coresPerSocket - 1 then
40:         changeToggleBitsInResponse()
41:     end if
42: end if
43: end foreach
```

```
44: newCombinerId = -1
45: newCombinerId = findProcessWaiting(wait)
46: if newCombinerId != -1 then
47:     getBit(response->combiner, newCombinerId) = true
48: else
49:     xchg(shared object, -1)
50:     i = processId foreach process in the system
51:     if requests[i]->toggle != getBit(response) then
52:         newCombinerId = i
53:     end if
54:     end foreach
55:     if newCombinerId != -1 then
56:         if CAS(shared object, -1, newCombinerId) == -1 then
57:             getBit(response->combiner, newCombinerId) = true
58:         end if
59:     end if
60: end if
```

4.2.2 Creating a request

The following section describes algorithm 2.1

To create a request, a process must first select if it wants to lock or free a range, enter the boundary of the range and if it is a read operation (lines 1-3). Finally it flips the toggle bit in the request to notify the delegate (line 4). It will then try to become the delegate by performing a compare and swap on an object shared by all processes (line 5). If the operation is successful, the process becomes the delegate (line 6). If not, it spins until the toggle bit from the response is flipped by the delegate or if the delegate stopped and designated the process to become the new delegate (lines 8-9).

If the process does not receive in response the command to become the delegate (line 11), the operation is complete and the process can return. If not, then the process resets the response (line 12) and becomes the delegate (line 13).

4.2.3 The delegate

The following section describes algorithm 2.2

The delegate has two phases: the normal sequence (lines 1-33) and the stopping sequence (lines 34-60)

Normal sequence

During the normal sequence, it iterates over all CPU sockets in the system. In each socket, it checks and executes the requests of the processes one at a time (line 2). It first polls the toggle bits from the request and the response. If they are the same, it goes to the next process in the socket (lines 3-5). If not, it then checks if the request has already been executed but could not return because the range is already locked by another process. If that is the case then the current process is ignored (lines 3-5).

The process is now considered *waiting* to avoid that an operation is executed twice (line 7). If the request is a locking operation (line 8), the range is inserted in the data structure (line 9), the blocking ranges are counted (line 10) and if the no other range overlaps (line 11), the delegate will write the response after executing all requests from the socket (line 12).

If the request is a freeing operation, the range is deleted from the data structure (line 15), all overlapping ranges have their counter of blocking ranges decremented by one (line 16). The delegate will then write the answer after going over every process in the socket (line 17). If a process that was blocked by the deleted range gets their counter down to zero, then the delegate with fill the response whenever it is turn of the process' socket.

When responding the delegation goes over the socket again (line 20) considering only processes whose answers are ready (lines 21-23), the delegate first places the allocated locks in the respective requests (line 25) and then flips the toggle bits in the response all at the same time (line 28).

The delegate will keep executing the normal sequence until:

1. The delegate scanned every process in every socket and no new requests were found.
2. The delegate has already executed a fixed number of operations.

If one of these conditions is met, the delegate will begin the stopping sequence after having responded to the socket it is currently dealing with (lines 29-31). If not the delegate goes to the next socket and the loop starts again (line 32).

Stopping sequence

The goal of the stopping sequence is to stop the delegate in a clean manner and assign the role to new process if necessary.

The delegate checks if it has responses it has not delivered yet (line 35). The procedure is the same as in the normal sequence: after checking the whole socket, write the acquired locks in the request and flip the toggle bits in the response (lines 36-41).

Finally, the delegate must either choose a successor from the processes that are waiting for their request to finish or return if there is no such process. The delegate first checks if a process is blocked by an overlapping range (line 45). If a process is waiting, the role of delegate is assigned to it (line 47). If that is not the case, it is still possible that a process entered a request and haven't been noticed and we must account for that. We first set back the shared object to its initial state before polling for new requests (line 49). Then either we detect a new request and assign the process that requested it as the new delegate using a compare and swap (lines 50-55), or the process did the compare and swap successfully (line 56 is not satisfied) and no other action is needed from the old delegate.

Chapter 5

Evaluation

The following results were obtained using a Intel(R) Xeon(R) Platinum 8276L CPU @ 2.20GHz with 224 cores. The system contains 8 sockets, each socket containing 28 cores.

Each range lock implementation ran for 30 seconds, each process alternated between locking a range and freeing it. The range lock covered 1000 bytes of memory and each process tries to allocate a random range. We tested different types of ratios of write operations to read operations: 0, 0.2, 0.5 and 1. We pinned processes to cores as it yielded more operations per second. Seven different implementations were tested:

5.1 Results

1. RWLock, the lock described in the *Related work* chapter proposed by [1]
2. ExclusionLock, a simpler variant proposed by [1], which does not include differentiation between reader and writer
3. SegmentLock, a method, proposed by [3], that divides the entire range into segments, each segment having to be locked separately
4. CombinerTree, combination using a red-black tree
5. CombinerList, combination using a linked list
6. DelegationTree, delegation using a red-black tree
7. DelegationList, delegation using a linked list

The first three are here for reference and do not use combination or delegation, the fourth has been implemented by our supervisor, Vishal Gupta, with minor modifications from our part and the last three have been fully implemented by us.

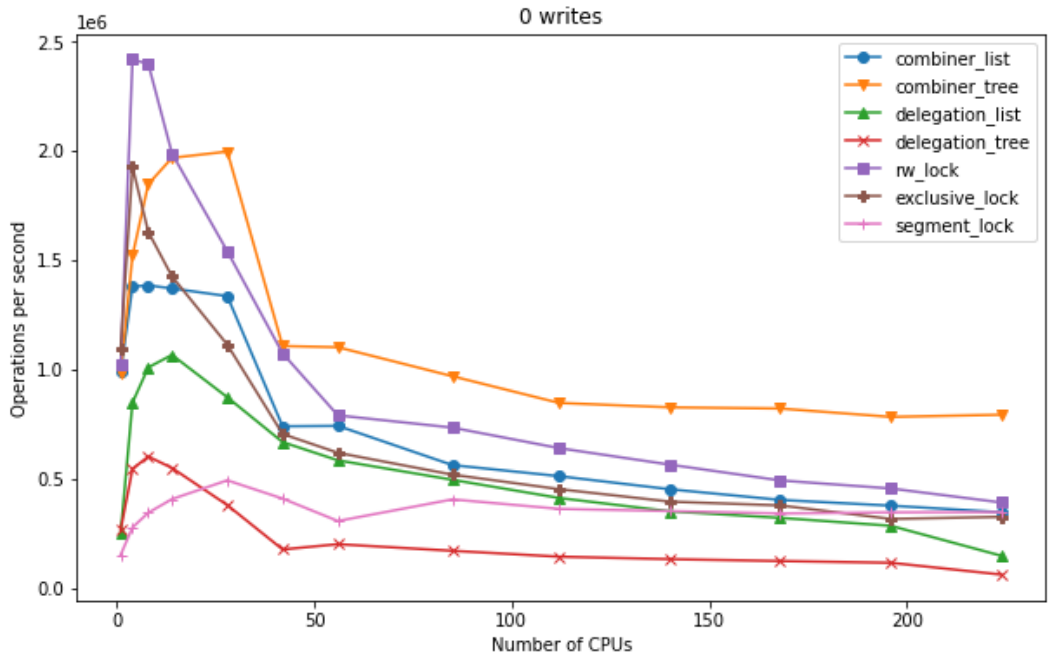

Figure 1: Throughput with 0% writes in function of the number of processes running

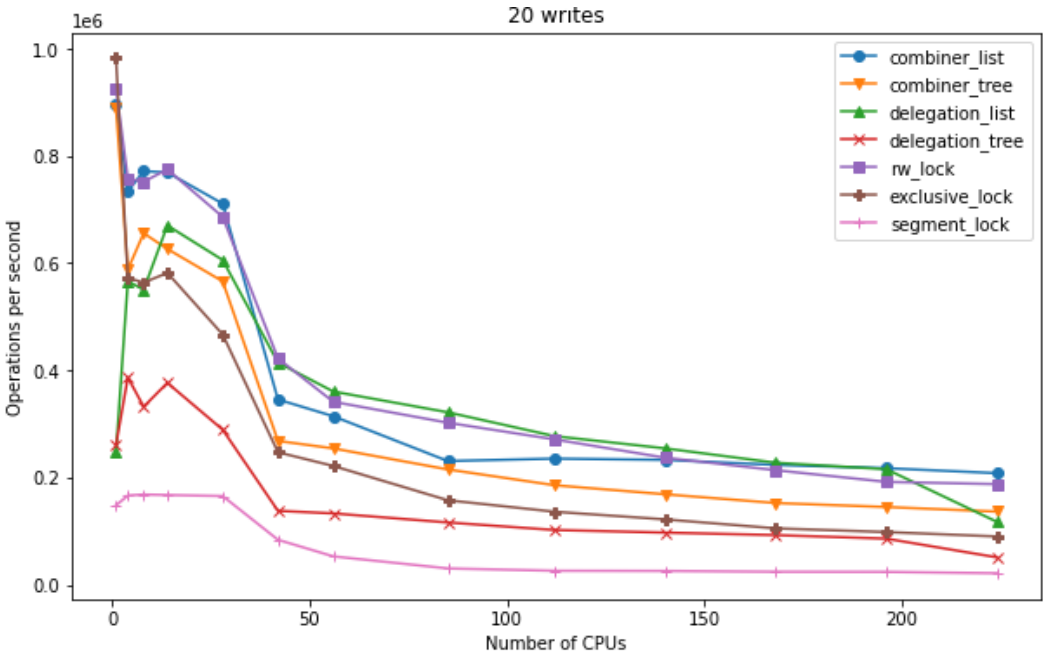

Figure 2: Throughput with 20% writes in function of the number of processes running

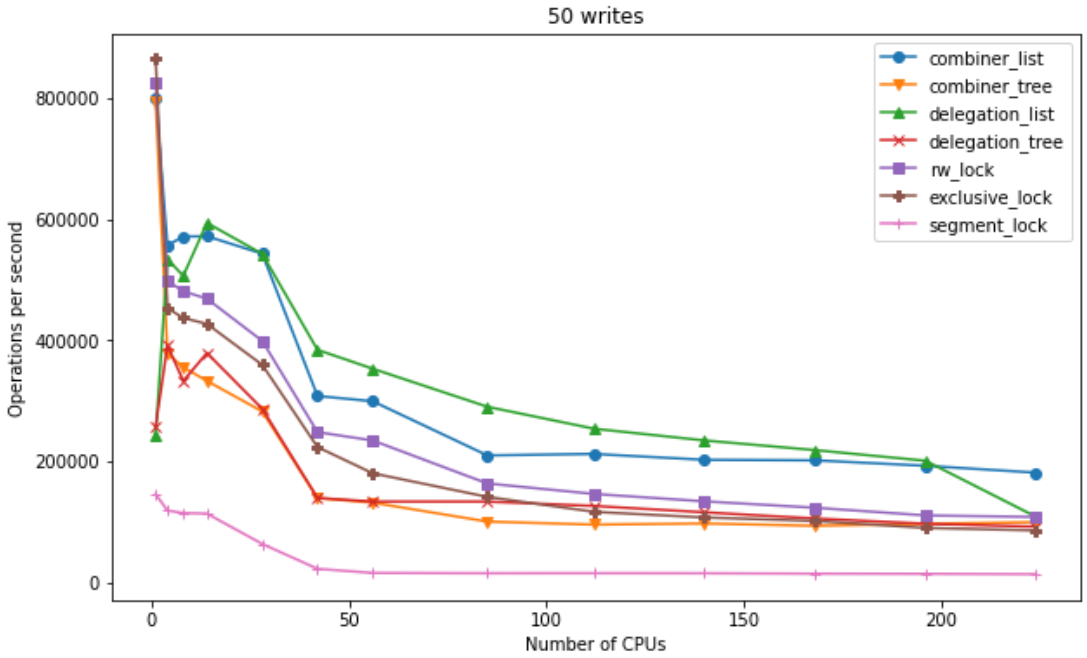

Figure 3: Throughput with 50% writes in function of the number of processes running

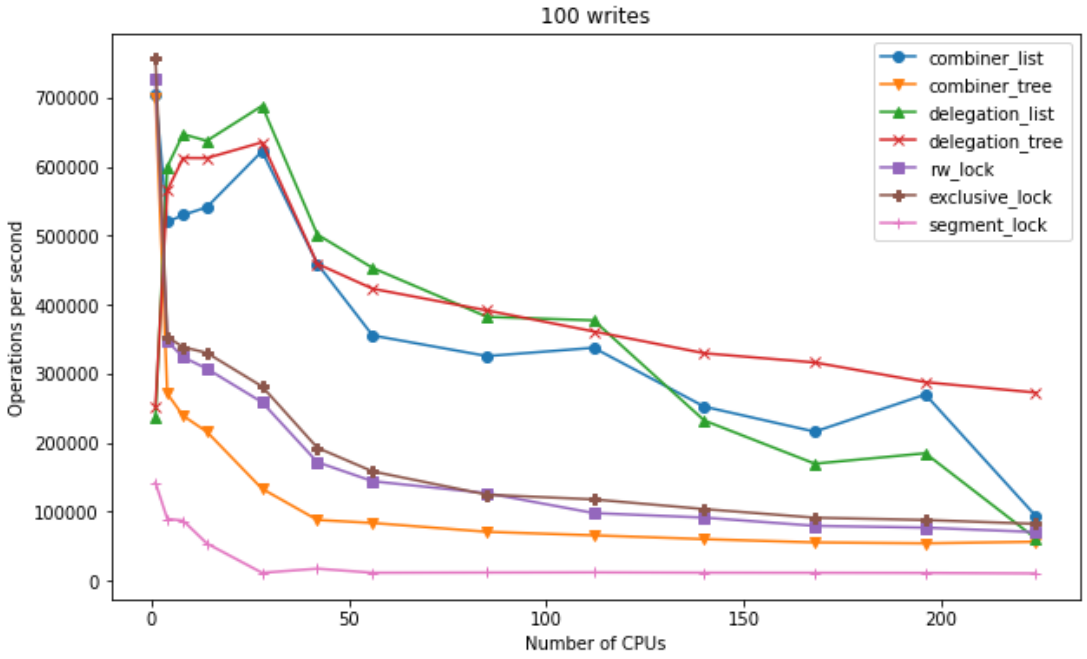

Figure 4: Throughput with 100% writes in function of the number of processes running

To validate our results, we measure the average time to execute an operation. We compare here delegation using a red-black tree and the segment lock with 100% writes. We use this setup because it contains the widest gap between two range locks in order to illustrate the efficiency of our design.

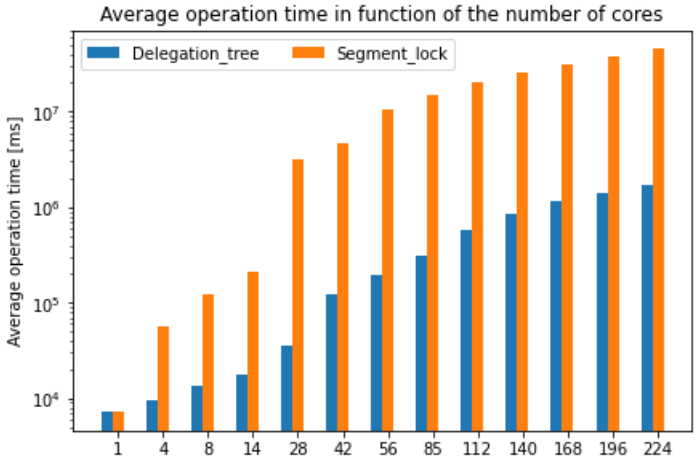

Figure 5: Average operation time with 100% writes

In this figure, the average operation time means the time it took on average to acquire and free the lock (this does not count the time a process held the lock.)

5.2 Discussion

Let us first discuss the similarities of our results no matter what the percentage of writes is. We can see that the reference implementations always share the same order of efficiency: *rwLock* is faster than *exclusiveLock* and *exclusiveLock* is faster than *segmentLock*. *RwLock* contains more optimizations than *exclusiveLock* so it is coherent to have it run faster. *SegmentLock* has the disadvantage to have to lock each segment using a compare and swap. Even though it works well when there are few overlapping ranges, that is not the case in our experiment which reduces its performance drastically. We also notice a sharp decrease when there is more than 28 processes in the system. 28 is also the number of cores contained in each socket so the decrease of the number of operations after 28 processes is caused by the inter-connect latency.

Let us now examine the differences. As was previously, and implicitly, mentioned, our designs behave very differently when we change the number of write operations. Some perform really well when there is a low amount of writes and poorly when there is only writes and vice versa for other designs. The 0% write scenario is the only case in which the throughput increased for a while when adding processes. It did not happen in the other cases because having multiple processes in the systems doing write operations forces processes to spend time on cache coherence because writing invalidates cache lines. All our designs (except the combiner using a tree) perform better when there is only write operations compared to when there is only half the operations that are writes. Finally we note that delegation or combination do not perform better than the other in every scenario.

Overall, our designs achieved better results compared to the reference implementations. For

instance, delegation using a tree achieved 3x the throughput of the best reference implementation when the number of processes is high. However we have not found a *one-implementation-fits-all* for every scenario. More work is needed to establish why some situations favors more a lock than another.

Chapter 6

Conclusion

In this report, we presented two designs and implementations of range locks. One used combination, the other delegation. We tested both of them with two different under-laying data structures: a red-black tree and a linked list with multiple percentages of write operations. We found that our implementations yielded up to a 3x speedup compared to the reference implementations that do not use combination or delegation. Some designs worked better when the writes are low or high but no implementation performed better compared to others in every situation.

Bibliography

[1] Dave Dice Alex Kogan and Shady Issa. “Scalable Range Locks for Scalable Address Spaces and Beyond”. In: *Fifteenth European Conference on Computer Systems (EuroSys ’20)*. 2020. DOI: https://doi.org/10.1145/3342195.3387533.

[2] Panagiota Fatourou and Nikolaos D. Kallimanis. “Revisiting the Combining Synchronization Technique”. In: *ACM SIGPLAN NoticesVolume 47 Issue 8*. 2012. DOI: https://doi.org/10.1145/2370036.2145849.

[3] June-Hyung Kim Jangwoong Kim Hyeongu Kang Chang-Gyu Lee Sungyong Park Youngjae Kim. “pNOVA: Optimizing Shared File I/O Operations of NVM File System on Manycore Servers”. In: *APSys ’19: Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*. 2019. DOI: https://doi.org/10.1145/3343737.3343748.

[4] Jakob Eriksson Sepideh Roghanchi and Nilanjana Basu. “ffwd: delegation is (much) faster than you think”. In: *SOSP ’17: Proceedings of the 26th Symposium on Operating Systems Principles*. 2017. DOI: https://doi.org/10.1145/3132747.3132771.