Scalable Range Locks for Scalable Address Spaces and Beyond

Alex Kogan
Oracle Labs
Burlington, MA, USA
alex.kogan@oracle.com

Dave Dice
Oracle Labs
Burlington, MA, USA
dave.dice@oracle.com

Shady Issa*
U. Lisboa & INESC-ID
Lisbon, Portugal
shadi.issa@tecnico.ulisboa.pt

Abstract

Range locks are a synchronization construct designed to provide concurrent access to multiple threads (or processes) to disjoint parts of a shared resource. Originally conceived in the file system context, range locks are gaining increasing interest in the Linux kernel community seeking to alleviate bottlenecks in the virtual memory management subsystem. The existing implementation of range locks in the kernel, however, uses an internal spin lock to protect the underlying tree structure that keeps track of acquired and requested ranges. This spin lock becomes a point of contention on its own when the range lock is frequently acquired. Furthermore, where and exactly how specific (refined) ranges can be locked remains an open question.

In this paper, we make two independent, but related contributions. First, we propose an alternative approach for building range locks based on linked lists. The lists are easy to maintain in a lock-less fashion, and in fact, our range locks do not use any internal locks in the common case. Second, we show how the range of the lock can be refined in the `mprotect` operation through a speculative mechanism. This refinement, in turn, allows concurrent execution of `mprotect` operations on non-overlapping memory regions. We implement our new algorithms and demonstrate their effectiveness in user-space and kernel-space, achieving up to 9× speedup compared to the stock version of the Linux kernel. Beyond the virtual memory management subsystem, we discuss other applications of range locks in parallel software. As a concrete example, we show how range locks can be used to facilitate the design of scalable concurrent data structures, such as skip lists.

*Work was done while the author was an intern at Oracle Labs.

EuroSys '20, April 27–30, 2020, Heraklion, Greece

ACM ISBN 978-1-4503-6882-7/20/04…$15.00
https://doi.org/10.1145/3342195.3387533

CCS Concepts • **Theory of computation → Concurrency**; • **Computer systems organization → Multicore architectures**; • **Software and its engineering → Mutual exclusion**; **Concurrency control**; **Virtual memory**.

Keywords reader-writer locks, semaphores, scalable synchronization, lock-less, Linux kernel, parallel file systems

ACM Reference Format:
Alex Kogan, Dave Dice, and Shady Issa. 2020. Scalable Range Locks for Scalable Address Spaces and Beyond. In *Fifteenth European Conference on Computer Systems (EuroSys '20), April 27–30, 2020, Heraklion, Greece.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3342195.3387533

1 Introduction

Range locks are a synchronization construct designed to provide concurrent access to multiple threads (or processes) to disjoint parts of a shared resource. Originally, range locks were conceived in the context of file systems [2], to address scenarios in which multiple writers would want to write into different parts of the same file. A conventional approach of using a single file lock to mediate the access among those writers creates a synchronization bottleneck. Range locks, however, allow each writer to specify (i.e., lock) the part of the file it is going to update, thus allowing serialization between writers accessing the same part of the file, but parallel access for writers working on different parts.

In recent years, there has been a surge of interest in range locks in a different context. Specifically, the Linux kernel community considers using range locks to address contention on `mmap_sem` [13], which is "one of the most intractable contention points in the memory-management subsystem" [9]. `mmap_sem` is a reader-writer semaphore protecting the access to the virtual memory area (VMA) structures. VMA represents a distinct and contiguous region in the virtual address space of an application; all VMA structures are organized as a red-black tree (`mm_rb`) [6]. The `mmap_sem` semaphore is acquired by any virtual memory-related operation, such as mapping, unmapping and mprotecting memory regions, and handling page fault interrupts. As a result, for data intensive applications that operate on chunks of dynamically allocated memory, the contention on the semaphore becomes a significant bottleneck [6, 9, 11].

The existing implementation of range locks in the Linux kernel is relatively straightforward. It uses a range tree

(based on red-black trees) protected by a spin lock [20]. Given that every acquisition and release of the range lock, for any range, results in the acquisition and release of that spin lock, the latter can easily become a bottleneck on its own under heavy use regardless of the contention on actual ranges. Note that even non-overlapping ranges and/or ranges acquired for read have to synchronize using that same spin lock. We expand on the implementation of existing range locks in the kernel and its shortcomings in Section 3.

Even when putting the issues in the existing range lock implementation aside, exploiting the potential parallelism when using range locks to protect the access to VMA structures in the Linux kernel is far from trivial. The key challenge is that addresses presented to virtual memory (VM) operations (singular addresses arising from page fault handling or ranges associated with APIs such as `mprotect`) do not necessarily fall on VMA boundaries. Thus, the enclosing range of the VM space that needs to be protected is not known in advance of walking the `mm_rb` tree. Therefore, simply applying a VM operation under the lock acquired for the range of that operation does not work. As an intuitive example, consider two `mprotect` operations on different (non-overlapping) memory ranges. If those operations acquire the range lock only on those (non-overlapping) ranges, they may race with each other on updates to the VMA metadata if they end up operating on the same VMA. Furthermore, regardless of whether two `mprotect`s operate on the same VMA, if one of them rotates the `mm_rb` tree, the other one may read an inconsistent state while traversing the tree in parallel. All these issues might be the reason that in the kernel patch that replaces `mmap_sem` with a range lock, the latter is always acquired for the full range[1] [5], exploiting no potential parallelism that range locks can provide[2].

This paper makes two related, but independent contributions. First, we propose an alternative design for efficient scalable range locks that addresses the shortcomings of the existing algorithm. Our idea is to organize ranges in a linked list instead of a range tree. Each node in the list represents an acquired range. Therefore, conceptually, once a thread manages to insert its node into the list, it holds the range lock for that particular range. While traversing a list to find the insertion point is less efficient than traversing a tree, the number of nodes in the list is expected to be relatively low, as it corresponds to the number of threads in the system accessing ranges. At the same time, lists are known to be more amenable for non-blocking updates, since unlike a (balanced) tree, one needs to modify atomically just one pointer to update the list. As a result, our list-based design does not require any lock in the common case.

Our second contribution is the discussion of applications for range locks in parallel software. Our prime focus is on scaling the virtual memory management in the Linux kernel by introducing a speculative mechanism into the `mprotect` operations. As we observe, in certain cases handling `mprotect` calls results in modifying the metadata of the underlying VMA without changing the structure of `mm_rb`. For those cases, our mechanism acquires the range lock only for a relatively small (refined) range, thus enabling parallel execution of `mprotect` operations on non-overlapping regions of virtual memory. As it turns out, those are the common cases for applications that use the GLIBC memory allocator, which is the default user-mode malloc-free allocator. The latter employs per-thread memory arenas, which are initialized by mmaping a large chunk of memory and mprotecting the pages that are actually in use. Those `mprotect` calls expand or shrink the size of the VMA corresponding to the set of pages with currently allocated objects, which are exactly the cases that our speculative mechanism supports.

We note that the applicability of range locks extends beyond the virtual memory management subsystem. As Kim et al. demonstrated recently [22], range locks can be used to optimize shared file I/O operations in a file system; we believe that the range locks we present in this paper can be used as a drop-in replacement for the implementation used in [22]. More generally, drawing from the original motivation behind the concept of range locks, the ideas presented in this paper appear to be a natural fit for parallel file systems; we plan to experiment with such systems in the future work. In addition, we argue that range locks can be highly useful in facilitating the design of scalable concurrent data structures. As a concrete example, we discuss the design of a new skip list in which a range lock is used for scalable synchronization between threads applying concurrent operations on the skip list. The new skip list is based on a well-known optimistic skip list by Helrihy et al. [19]. Instead of acquiring multiple locks during an update operation (potentially, as many as the number of levels in the skip list) [19], our design acquires one range only. Beyond the potential performance benefits of reducing lock contention and the number of required atomic operations, our design eliminates the need for associating a (spin) lock with every node in the list, thus reducing the memory footprint of the skip list.

We have evaluated our ideas both in the user-space and kernel-space. For the former, we implemented our list-based range locks and compared them to the tree-based range lock implementation that we ported from the Linux kernel into the user-space. Our experiments confirm that the new range locks scale better and outperform existing range locks in virtually all evaluated settings. Moreover, we show that the range lock-based skip lists perform significantly better when using our implementation of range locks underneath, compared to the tree-based range lock implementation. We

[1] The range lock API includes calls to acquire the lock for a specific range (e.g., [10..25]) as well as a special call to acquire the lock for the entire (full) range (i.e., $[0..2^{64}-1]$).

[2] The author of the patch notes that "while there is no improvement of concurrency perse, these changes aim at adding the machinery to permit this in the future." We are not aware of any follow-up work that does that.

also implemented the new range locks in the kernel, and evaluated them with Metis, a suite of map-reduce benchmarks [23] used extensively for the scalability research of the Linux kernel [3, 6, 11, 21]. When coupled with the speculative mechanism in `mprotect`, some Metis benchmarks run up to 9× faster on the modified kernel compared to stock and up to 69× faster compared to the kernel that uses tree-based range locks.

2 Related Work

Range locks (or byte-range locks) were conceived in the context of file systems to support concurrent access to the same file [2]. Since files are a continuous range of bytes, different processes can access disjoint regions within the same file if they acquire a (range) lock for the desired region, e.g., through the `fcntl` operation in Unix [2]. More recently, range locks gained attention as an important piece in the design of parallel and distributed file systems. Aarestad et al. [1], for instance, proposed using a red-black tree to store the ranges acquired by different processes. The same approach is taken by recent efforts within the Linux kernel development community to replace the read-write semaphore within the virtual memory sub-system with a red-black tree-based range lock implementation [4, 20]. However, as explained earlier, relying on a red-black tree protected by a spin lock can be a serious scalability bottleneck, as we will confirm later in Section 7. At the same time, our approach does not use locks in the common case.

In a recent and highly relevant work [22], Kim et al. consider using range locks in the context of parallel file systems, and make a similar observation regarding the lack of scalability of the existing kernel range locks. They propose an alternative design for range locks in which the entire range is divided into (a preset number of) segments, each associated with a reader-writer lock. To acquire a certain part of the range for read or write, one needs to acquire the reader-writer locks of the corresponding segments in the respective mode. In their proposal, the full range acquisition is particularly expensive, as it requires acquiring all underlying reader-writer locks. Moreover, choosing the right granularity, i.e., the number of segments, is critical – too few segments would create contention on the underlying reader-writer locks, while too many segments would make range acquisition more expensive – yet, Kim et al. do not discuss how the granularity should be tuned. Therefore, we believe the applicability of Kim et al.'s scenarios is limited to the cases where the size of the entire range and the granularity of the access are known and static, which is precisely the case considered in [22]. Nevertheless, we include Kim et al.'s range locks in our performance study in Section 7.

As mentioned earlier, one of the main motivations behind the renewed interest in range locks is to design a scalable locking mechanism for the kernel address space operations. Song et al. attempted to address this problem in the context of parallelizing live VM migration [27]. To that end, they proposed a range lock implementation based on a skip list protected by a spin lock. Conceptually, their design is very similar to the one found in the Linux kernel [20]. In particular, both cases have the same bottleneck in the form of a spin lock protecting their corresponding underlying data structures for tracking acquired ranges.

Several works pursued the same goal of scaling kernel address space operations via a different route: replacing the red-black-tree `mm_rb` with alternative data-structures. Clements et al. [6] proposed using a RCU-balanced tree to allow concurrency between a single writer and multiple readers. In addition to not allowing parallel update operations, the proposed tree trades fewer rotations for tree imbalance, which can increase tree traversal times. In another work by the same authors, they proposed using a radix tree, where each mapped page will be inserted in a separate node within the tree [7]. Such design supports concurrent read and update accesses to non-overlapping nodes. However, this comes at two significant costs: *(i)* a large memory footprint for using per-page nodes, and *(ii)* high locking overhead, since locking a range of pages entails locking several nodes within the tree. Unlike both proposals by Clements et al., our work does not require changing `mm_rb` and thus requires less intrusive changes to the kernel.

3 Existing Range Locks in the Kernel

The existing implementation of range locks in the Linux kernel uses a range tree (based on red-black trees) protected by a spin lock [20]. To acquire a range, a thread first acquires the spin lock and then traverses the tree to find a count of all the ranges that overlap with (and thus, block) the given range. For a reader-writer range lock, this count does *not* include overlapping ranges belonging to other readers (if the given acquisition is also for read) [4]. Next, the thread inserts a node describing its range into the tree, and releases the spin lock. If at that point the count of blocking ranges is zero, the thread has the range lock and can start the critical section that the lock protects. Otherwise, it waits until the count drops to zero, which would happen when threads that have acquired blocking (i.e., overlapping) ranges exit their respective critical sections. Specifically, when a thread is done with its range, it acquires the spin lock, removes its node from the tree and then traverses the tree, decrementing the count of blocking ranges for all relevant ranges, and finally releases the spin lock.

This range lock implementation has several shortcomings. The most severe one is the use of a spin lock to protect the range tree. This lock can easily become a bottleneck on its own even without the logical contention on ranges. Note that every acquisition and release of the range lock results in the acquisition and release of that spin lock. Therefore, even non-overlapping ranges and/or ranges acquired for read have to synchronize using that same spin lock.

Furthermore, while placing all ranges in the range tree preserves the FIFO order, it limits concurrency. Assume that we have three exclusive acquisition requests for ranges coming in this order: A=[1..3], B=[2..7], C=[4..5]. While A holds the lock, B is blocked (it overlaps with A), and C is blocked behind B, but in practice, it could proceed as it does not overlap with A. Finally, the existing range locks have no fast path, that is, even when there is a single thread acquiring a range, it still would go through the same path of acquiring the spin lock, updating the range tree and so on.

The list-based range locks presented in this paper address all the aforementioned issues. First, they only use a lock when fairness is concerned, i.e., to avoid starvation of threads trying to acquire a range, but repeatedly failing to do so due to other threads that manage to acquire overlapping ranges. In our experiments, this is an unlikely scenario, meaning that our range locks do not use any locks in the common case. Second, list-based range locks can achieve a higher level of parallelism by allowing concurrent threads to acquire more (non-overlapping) ranges. Considering the example above, for instance, while A is in the list, B waits until A finishes, but C can go ahead and insert its node into the list after A. Finally, our design allows the introduction of a fast path, in which the range lock can be acquired in a small constant number of steps. This path is particularly efficient for single-thread applications or multi-thread applications in which a range lock is acquired by one thread at a time.

We opted to use a linked list as an underlying data structure for the relative simplicity and amenability to concurrent updates of the former. We note that, in general, a linear-time search provided by a linked list is less efficient than the logarithmic-time search provided by a balanced search tree or a skip list. In practice, however, this should not present an issue, as in all applications that we consider the number of stored elements (ranges) in the list is relatively small since it is proportional to the number of threads accessing concurrently the resource(s) protected by the range lock. For the setting in which this assumption does not hold, we plan to investigate extending our design to employ a skip list for more efficient search operations in the future.

4 Scalable Range Lock Design

4.1 Exclusive Access Variant

We start with a simpler version of our linked list-based range locks algorithm intended for mutual exclusion, i.e., it supports concurrent acquisition of disjoint ranges, but no overlapping ranges are allowed. In the next section, we describe an extension of the algorithm to support reader-writer exclusion, where readers can acquire overlapping ranges, but a writer cannot overlap with another (reader or writer) thread.

The idea at the basis of the algorithm is to insert acquired ranges in a linked list sorted by ranges' starting points. Accordingly, any overlapping ranges will compete to be inserted at the same position in the list. Therefore, by relying on an atomic compare-and-swap (CAS) primitive, it is possible to ensure that only one range from a group of overlapping ranges will succeed in entering the list while others will fail.

The pseudo-code for the exclusive access list-based range locks algorithm is shown in Listing 1. It presents the lock structures and the implementation of the `MutexRangeAcquire` and `MutexRangeRelease` functions as well as the auxiliary functions called by those two. For the clarity of exposition, we assume sequential consistency. Our actual implementation uses `volatile` keywords and memory fences where necessarily. CAS and FAA indicate opcodes for the compare-and-swap and fetch-and-add atomic instructions, respectively[3]; `Pause()` is a no-op operation used for polite busy-waiting.

For each shared resource protected by a range lock, a `ListRL` list must be defined. Each node, `LNode`, within the list contains the range it defines and a pointer to the next node in the list (cf. Listing 1). At the beginning, the head of the list points to `null`, indicating that the list is empty.

When a thread requests an exclusive access over the given region within a resource, it first creates an instance of the `RangeLock` structure (cf. Line 11), which contains a pointer to the `LNode` structure. Note that for simplicity, we allocate a new `RangeLock` instance each time the `MutexRangeAcquire` is called. It is possible, however, to maintain and reuse a pool of `RangeLock` instances; we discuss memory management of those instances in detail in Section 4.4. Next, the thread initializes the `RangeLock` structure (cf. Lines 12–13). Finally, in order to acquire a range, the thread must successfully insert the corresponding node into the given range lock list structure (cf. Line 14). To release the acquired range (in `MutexRangeRelease`), a node corresponding to the range is deleted from the list (cf. Line 18).

Correctness Argument: We argue that the pseudo-code in Listing 1 is a correct and a deadlock-free implementation of exclusive access range locks. For correctness, we argue that the implementation never allows two threads to acquire range locks with overlapping ranges. This claim is based on the following invariant:

Invariant 1. *For any two consecutive ranges R1 and R2 in the list ListRL, R1.end ≤ R2.start.*

To prove the progress property, we note that a thread T would remain infinitely long in the `InsertNode` function only if (a) it finds infinitely often its `prev` variable pointing to a deleted node (cf. Line 33), or (b) it traverses infinitely many logically deleted nodes (cf. Lines 35–38), or (c) it traverses infinitely many ranges that end before the thread's range starts (cf. Lines 41–43), or (d) it waits infinitely long to a thread

[3] It is easy to simulate FAA with CAS on architectures that do not have a native support for the former.

```
1  class LNode:                                  ##defines a node a within the list
2    __u64 start; __u64 end
3    LNode* next

4  ##defines a list for range locks protecting the same resource
5  class ListRL:
6    LNode* head                                 ## pointer to the head of the list

7  ##defines a range lock to protect a region within a shared resource
8  class RangeLock:
9    LNode* node

10 def MutexRangeAcquire(ListRL* listrl, __u64 start, __u64 end):
11   RangeLock* rl = new RangeLock()
12   rl->node = new LNode()
13   rl->node->start = start; rl->node->end = end; rl->node->next = NULL
14   InsertNode(listtl, rl->node)
15   return rl
16
17 def MutexRangeRelease(RangeLock* rl)
18   DeleteNode(rl->node)

19 def compare(LNode* lock1, LNode* lock2):
20   if !lock1: return 1                  ## lock1 is end of the list, no overlap
21   ## check if lock1 comes after lock2, no overlap
22   if lock1->start >= lock2->end: return 1
23   ## check if lock1 is before lock2, no overlap
24   if lock2->start >= lock1->end: return -1
25   return 0                                    ## lock1 and lock2 overlap

26 def marked(LNode *node): return is_odd((__u64)node)
27 def unmark(LNode *node): return (__u64)node - 1

28 def InsertNode(ListRL * listrl, LNode *lock):
29   while true:
30     LNode** prev = &listrl->head
31     LNode* cur = *prev
32     while true:
33       if marked(cur):                         ## prev is logically deleted?
34         break  ## traversal must restart as pointer to previous is lost.
35       elif cur and marked(cur->next):         ## cur is logically deleted?
36         LNode *next = unmark(cur->next)
37         CAS(prev, cur, next)                  ## try to remove it from list
38         cur = next                            ## and continue traversing the list
39       else:                                   ## cur is currently protecting a range
40         auto ret = compare(cur, lock)
41         if ret == -1:                         ## lock succeeds cur:
42           prev = &cur->next                   ## continue traversing ...
43           cur = *prev                         ## the list.
44         elif ret == 0:                        ## lock overlaps with cur:
45           while(!marked(cur->next)):          ## wait until ...
46             Pause()                           ## cur marks itself as deleted
47         elif ret == 1:                        ## lock precedes cur or reached end of list:
48           lock->next = cur                    ## then try to ...
49           if CAS(prev, cur, lock): ## insert lock into the list
50             return 0                          ## success - the range is acquired now.
51           cur = *prev                         ## o/w continue traversing the list.

52 def DeleteNode(LNode *lock):
53   FAA(&lock->next, 1)                         ## logically mark lock as deleted.
```

Listing 1. Pseudo-code for the exclusive access range locks implementation.

with an overlapping range (cf. Line 46). Given that the list contains a finite number of nodes when T calls `InsertNode`, cases (a), (b), and (c) are possible only if some other thread (or threads) insert (and delete) infinitely many nodes, which in turn means that those threads acquire and release infinitely many ranges while T is executing `InsertNode`. Assuming that no thread fails while holding the range lock, then either case (d) is impossible as the thread would mark its node as logically deleted in a finite number of steps (if the hardware supports wait-free FAA), or case (d) is possible only if infinitely many threads would acquire and release a range lock (for the CAS-based implementation of FAA). Thus, T would either return from `InsertNode` (and thus acquire the range lock), or infinitely many threads would acquire and release the range lock while T is executing `InsertNode`.

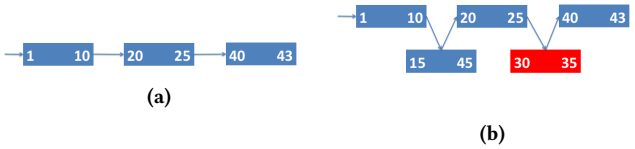


Figure 1. An example for a race condition between readers and writers solved by validation. (a): Three reader ranges are in the list. (b): A new reader with the range `[15..45]` arrives, and since it starts before the reader with the range `[20..25]`, it inserts itself into the list after a reader with the range `[1..10]`. At the same time, a writer with the range `[30..35]` arrives, finds that it does not overlap with any reader and inserts itself into the list after the reader with the range `[20..25]`.

We note that the described implementation of the list-based range lock is not starvation-free, e.g., a thread trying to insert a node into the list may continuously fail to apply CAS (cf. Line 49) and/or be forced to restart the traversal if its `prev` pointer gets marked (cf. Lines 33–34). In Section 4.3 we describe a simple mechanism to introduce fairness and avoid starvation.

4.2 Reader-Writer Variant

In the previous section, we have presented a range lock algorithm that supports acquiring exclusive access on defined ranges. Now, we extend the algorithm to handle reader-writer synchronization. For the sake of brevity, in this section threads acquiring a range lock in shared mode will be referred to as readers while threads acquiring a range lock in exclusive mode will be referred to as writers.

A natural way to extend the range locks algorithm from the previous section is to consider the access mode (read or write) in the `compare` function, and allow an overlap when both compared ranges belong to readers. In other words, we would traverse the list (in `InsertNode`) and insert the given node into the list even if that node (i.e., its range) overlaps with an existing node, and both nodes belong to readers.

Unfortunately, this approach enables a race condition between readers and writers, exemplified in Figure 1. A reader may "miss" a writer with an overlapping range located down the list. At the same time, a writer may "miss" a reader with an overlapping range that entered the list at the point that

the writer has already traversed. This race condition is possible because overlapping readers and writers may insert themselves into the list at different points (i.e., after different nodes), and therefore they do not compete to modify the same (`next`) pointer (see Figure 1).

We solve this problem with an extra validation step performed by readers and writers. Specifically, when a reader inserts its node into the list, it continues to scan the list until it finds a node with a range that does *not* overlap. If during this scan the reader comes across a writer, it waits until the writer's node is (logically) deleted. As for the writer, its validation step is slightly different (since a similar wait by writers for readers would lead to deadlock). Once the writer inserts itself into the list, it re-traverses the list from the head until it finds its own node. If during this re-traversal, a writer finds a reader with an overlapping range, the writer leaves the list (by logically deleting its node) and restarts the acquisition attempt from the beginning. The race conditions can only happen between a reader and a writer that have both inserted themselves into the list, therefore re-traversing the list will guarantee detecting such a race.

Note that this validation approach may cause starvation of writers, as they may be forced to restart repeatedly by incoming readers. We describe a way to avoid this issue in Section 4.3. Furthermore, note that our validation approach gives preference to readers, since in case of a conflict they stay in the list while writers restart. It is straightforward to reverse the scheme and give preference to writers instead, by letting them stay in the list (while waiting for conflicting readers to leave) and making the readers restart in case of a conflict.

Listing 2 shows how to implement shared range locks, where overlapping acquisitions with shared (reader) accesses do not block each the other. The pseudo-code in Listing 2 is presented in the form of diffs from Listing 1. The `LNode` structure includes now a flag (`reader`) indicating whether the corresponding range is acquired for read or for write. (This is a trivial change and thus not shown). The `RWRangeAcquire` function is similar to `MutexRangeAcquire` (in Listing 1), except that the call to `InsertNode` is now wrapped in a `do-while` loop. This loop will be executed more than once by a writer only, and only in the case the writer's validation fails. The `RWRangeRelease` function is identical to `MutexRangeRelease` (in Listing 1) and thus not shown. The `compare` function is adapted in a straightforward way to allow overlapping reader ranges (see Lines 20–27). Finally, the only change in the `InsertNode` function is the call to validation functions according to the access mode for which the range lock is acquired (see Lines 30–32). The details of the validation functions are omitted due to lack of space.

Correctness Argument: We argue that the pseudo-code in Listing 2 is a correct implementation of reader-writer range locks. To that end, we argue that the implementation never allows two threads to acquire conflicting ranges — ranges

```
1  ##function called to protect a given range
2  ##within a resource protected by list
3  def RWRangeAcquire(ListRL* listrl, __u64 start, __u64 end, int reader):
4    do:
5      RangeLock* rl = new RangeLock()
6      rl->node = new LNode()
7      rl->node->start = start
8      rl->node->end = end
9      rl->node->next = NULL
10     rl->node->reader = reader          ## set to 1 if reader, 0 if writer
11   while(InsertNode(listrl, rl->node))
12   return rl
13
14 ## return values:
15 ## -1: if lock1 comes before lock2, or
16 ##       if both are readers and lock1 starts before lock2
17 ## 0:  if they overlap (and at least one of the locks is a writer)
18 ## +1 if lock1 comes after lock2, or
19 ##       if both are readers and lock1 starts after lock2
20 def compare(LNode* lock1, LNode* lock2):
21   if !lock1: return 1
22   int readers = lock->reader + lock2->reader
23   if lock2->start >= lock1->end: return -1
24   if lock2->start >= lock1->start and readers == 2: return -1
25   if lock1->start >= lock2->end: return 1
26   if lock1->start >= lock2->start and readers == 2: return 1
27   return 0
28 def InsertNode(ListRL * listrl, LNode *lock):
29   ...                          ## same as InsertNode in Listing 1 up to Line 49
30     if CAS(prev, cur, lock):          ## try to insert lock into the list
31       if lock->reader:  return r_validate(lock)   ## validate as a reader
32       else: return w_validate(listrl, lock)       ## validate as a writer
33   ...                                             ## same as Listing 1
34
```

Listing 2. Reader-Writer range locks presented as diffs from the corresponding functions in Listing 1.

conflict when they overlap and at least one of them is a writer. Our claim is based on the following invariant:

Invariant 2. *For any two consecutive ranges R1 and R2 in ListRL, R1.start ≤ R2.start. Moreover, if R1 is a writer, then R1.end ≤ R2.start.*

Based on this invariant, if a reader or a writer G in *ListRL* overlaps with a writer W, then $G.start \leq W.start$ (if $W.start \leq G.start$ then, according to Invariant 2, $W.end \leq G.start$, thus they can not overlap). Assume there is a writer G in *ListRL* that overlaps with W and $G.start \leq W.start$. Since G is a writer then $G.end \leq W.start$ (otherwise Invariant 2 breaks), a contradiction. Now, we are left with the case of G being a reader. There are two possibilities: either *(i)* G entered *ListRL* before W or *(ii)* after W. Note that a range enters *ListRL* after a successful CAS operation at Line 30 in Listing 2.

The intuition at the basis of our correctness argument is that if a conflicting range that enters *ListRL* last (among the two conflicting ranges) defers to the other conflicting range, we can guarantee reader-writer exclusion. Accordingly, to handle the first case, `w_validate` is executed after the CAS operation at Line 30, and since it starts traversing ranges in *ListRL* from the head node, then any range

G with $G.start \leq W.start \leq G.end$ that entered *ListRL* before W (and has not left yet) is guaranteed to be visited during the traversal. For the second case, `r_validate` is executed after the CAS operation, and since it starts traversing ranges in *ListRL* from the node succeeding G, any range W with $G.start \leq W.start \leq G.end$ that entered *ListRL* before G is guaranteed to be visited during the traversal. Consequently, by ensuring that both `w_validate` and `r_validate` do not return successfully if a conflicting range lock is visited, reader-writer exclusion is guaranteed.

As for deadlock freedom, the same arguments used for the basic mutual exclusion apply also for the reader-writer pseudo-code. There are two additional cases, though, in which thread T may wait infinitely long in `InsertNode`: (a) when `w_validate` infinitely often returns 1 and (b) when `r_validate` function waits infinitely long for a thread with an overlapping writer range. Case (a) is possible if other thread (or threads) insert (and delete) infinitely many overlapping nodes, which in turn means that those threads acquire and release infinitely many reader ranges while T is executing `InsertNode`. Assuming that no thread fails after executing the CAS operation at Line 30, case (b) is similar to case (d) in the exclusive access variant (see Section 4.1).

Similarly to what we mentioned earlier, we note that while the presented reader-writer range locks are deadlock-free, they are not starvation-free. We discuss next how our design can be augmented with an auxiliary lock to avoid starvation.

4.3 Fairness

The range lock design presented so far does not use any locks. However, it allows starvation of a thread repeatedly failing to insert its node into the list due to other threads concurrently acquiring and releasing locks (and thus modifying the list). A simple way to avoid that is to introduce an auxiliary (fair) RW-lock coupled with an `impatient` counter. A thread acquiring the range lock checks the `impatient` counter, and if it is equal to zero (common case), proceeds with the range acquisition. Otherwise, if the counter is non-zero, it acquires the RW-lock for read. When a thread fails to acquire the range lock in a few attempts, it bumps up the `impatient` counter (atomically) and acquires the RW-lock for write. The counter is decremented (atomically) upon the release of the RW-lock that was acquired for write. Note that any race between a thread reading zero from the counter and a thread incrementing the counter is benign, as the sole purpose of this counter is to introduce fairness rather than ensure the correctness of the underlying range lock.

4.4 Memory Reclamation

In the proposed design of range locks, threads traverse list nodes concurrently with threads modifying the list. While this approach avoids the bottleneck of an auxiliary lock protecting the underlying structure as found in the existing implementation of range locks [4, 20], the lock-less traversal of a list poses a challenge with respect to the memory management of list nodes. This is because a list node may not be immediately reclaimed once it is removed from the list, since other threads traversing the list may have a reference to this node and may try to access its memory after it has been removed from the list. This is a well-known problem in the area of concurrent data structures [15, 26], and multiple solutions are available [18].

For our kernel-space implementation, we employ the read-copy-update (RCU) method [25], which is readily supported in the Linux kernel [24]. RCU is a synchronization mechanism that allows readers (threads that access shared data without modifying it) to execute concurrently with a writer (a thread modifying shared data) without acquiring locks. The idea at the basis of RCU is for readers to announce when they start and finish accessing shared data, while writers apply their changes to a copy of the data that is visible to only new readers (i.e., readers that started after the writer). The old data is then atomically replaced by the (modified) copy when there are no more active old readers. In the context of memory reclamation, threads traversing the list mark themselves as readers throughout the traversal, while a thread trying to reclaim memory, performs that operation as a writer. In the user-space, we opted for a related, but simpler approach of epoch-based reclamation scheme [16].

4.5 Fast Path Optimization

The proposed range lock implementation is amendable to a fast path optimization, which allows the range lock to be acquired and released in a constant number of steps when the lock is not contended. This is particularly important for a single thread execution, but is also useful when the lock is accessed by multiple threads while only one of them accesses the lock at a time.

The fast path is implemented as following. When a thread acquires the range lock, it checks whether the list is empty (i.e., whether `head` points to `null`). If so, it attempts to set (using CAS) the `head` of the list to the marked pointer to the node corresponding to the range lock acquisition request. If successful, the range lock acquisition is complete. In pseudo-code, the fast range lock acquisition path is implemented with the following two lines inserted right before the call to `InsertNode` in the range lock acquisition function (e.g., before Line 14 in Listing 1):

```
if ( listrl ->head == NULL and CAS(&listrl->head, NULL, mark(rl->node))):
  return rl ;
```

The (not shown) `mark` macro simply sets the LSB of the given pointer. Note that the `head` pointer can be marked only if the lock has been acquired on the fast path. We exploit this fact in two places. First, during unlock, if a thread t finds that the `head` is marked and points to t's node, t realizes that it has acquired the range lock through the fast path, and attempts to release it by setting `head` to `null` (using CAS). At the same time, if another thread t' attempts to acquire the

range lock on the regular path and finds head being marked, it first removes the mark (by changing head to point to the same node but without mark using CAS), and then proceeds with the acquisition. This ensures that a range lock l acquired on the fast path would be properly released on the regular path if other threads acquired other ranges in the meantime between l's acquisition and release.

In summary, the main difference between the fast and regular paths is in the way nodes are removed from the list. While on the regular path, the node is marked during lock release, and removed during lock acquisition when (possibly) another thread traverses the list, on the fast path the removal is eager. This reduces the total number of atomic operations required to delete a node from the list, and keeps the number of steps performed during the lock operation constant as there are no marked nodes that are needed to be removed from the list first.

5 Refining Ranges in VM Operations

5.1 Background

Operating systems provide processes with the virtual memory (VM) abstraction. It allows processes to assume they have access to all possible addressable memory, regardless of the actual underlying physical memory. To keep track of how regions within a process's virtual memory map to actual physical memory pages (whether located in the main memory or swapped to disk), the Linux kernel uses the concept of Virtual Memory Area (VMA) structures [14]. In practice, VMA is a data structure that defines a distinct contiguous region within the virtual memory address space using a start address and a variable length (multiple of a page size). The VMA metadata also includes other attributes, such as the mapping to physical memory, access permissions, pointers to neighboring VMA structures, etc. For each process, the Linux kernel stores all its associated VMA structures in a red-black tree (mm_rb). A typical VM operation starts by querying mm_rb with an address (provided as an input from the API caller) to find the enclosing VMA (if it exists). According to the nature of the operation, it may read or change some metadata of a VMA, split a VMA, merge two VMA structures, insert a VMA, delete a VMA, etc. Note that a single VM operation may perform several of these operations on one or more VMA structures, according to the given input address range. Moreover, splitting, merging, inserting and deleting VMA structures incur structural changes to the mm_rb. To that end, operations that might modify VMA structures and/ormm_rb (such as mprotect) acquire mmap_sem for write, while operations that only read VMA's metadata (such as the page fault handler) acquire mmap_sem for read.

While the concept of range locks may appear, at a first glance, as a natural fit for synchronizing the access to regions of the shared virtual memory address space, the task of applying those locks for this purpose in the Linux kernel is not straightforward due to mainly two reasons: *(i)* the APIs of VM operations are oblivious to the underlying VMA structures, and rely on querying mm_rb for this purpose; and *(ii)* a VM operation may end up performing structural changes to mm_rb (and thus interfere with other concurrent VM operations accessing mm_rb), and this is unknown a-priori.

(a) Two adjacent VMA structures, with different protection flags.

(b) Same VMA structures after mprotect(0x1800, 4096, PROT_READ | PROT_WRITE) returns.

Figure 2. Example for an mprotect operation changing VMA metadata without modifying the mm_rb tree.

As a concrete example of the challenge of using range locks in VM operations, consider two calls: mprotect(0x100000, 65536, PROT_NONE) and mprotect(0x180000, 65536, PROT_READ). If we naively protect only the range on which each call operates (i.e., [0x100000 .. 0x110000] and [0x180000 ... 0x190000]), and those two ranges fall within the scope of the same VMA, the two operations may simultaneously acquire range locks for the corresponding ranges, and overwrite each other's updates to the metadata of that same VMA. Moreover, if those calls result in a structural modification to mm_rb, they would perform those modifications without synchronizing one with another.

To overcome these issues, one might always acquire the range lock for the full range whenever this lock is required in write mode. This would, however, preclude any parallelism when a writer acquires the range lock, and in fact, is expected to perform worse than mmap_sem (since the latter has a more efficient acquisition path).

5.2 mprotect

By inspecting the implementation of various VM operations [14], we notice that they do not always end up modifying mm_rb. For instance, consider the case when there are two neighboring VMA structures describing two contiguous memory regions with different protection flags, and mprotect is called on the area at the head of the second VMA (or the tail of the first VMA), with protection flags identical to the flags of the other VMA (see Figure 2). In that scenario, the boundaries (i.e., the metadata) of the involved VMA structures are changed, but the structure of mm_rb remains unchanged. As mentioned in the Introduction, this case is common in the GLIBC memory allocator. Consequently, for the cases where mm_rb does not change, we devise a speculative approach, in which the range lock is optimistically acquired only for the relevant part of the VM address space. We note that when a VM operation needs to modify mm_rb (e.g., when mprotect splits a VMA into two, thus it needs to create a node corresponding to the new VMA and insert it into mm_rb), acquiring the range lock for

the entire range is the only available option to synchronize correctly with other operations traversing the mm_rb tree.

Listing 3 provides the pseudo-code for the mprotect operation with the integrated speculative mechanism. The intuition behind our speculative approach is that if we are able to decide whether the mprotect operation will end up modifying mm_rb before the mprotect applies its changes, then it is safe to lock only the respective range; otherwise, if we discover that mm_rb needs to be modified, we restart the mprotect operation after acquiring the full range (for write). The latter action prevents other concurrent speculative operations from running and potentially reading inconsistent mm_rb while it is being modified. To this end, we augment the major memory management structure in the Linux kernel (mm) with a sequence number. This number is incremented every time a range lock acquired for the full range in write mode is released. We use the sequence number to detect whether the mm_rb has changed during the speculative operation as described below.

The first step of the mprotect operation is to locate the relevant VMA given the input address and size. Therefore, we first acquire the range lock in read mode for the input range. This ensures that the structure of the underlying mm_rb would not change while find_vma() is running, since we make sure that mm_rb only changes under the range lock acquired in write mode for the entire range. (As its name suggests, find_vma() traverses mm_rb searching for the VMA that contains the given address, or more precisely, searching for the first VMA whose end address is larger than the given address). Note that since the range lock is acquired in read mode, this step may run in parallel with other speculating operations (or any other operation that acquires a range lock in read mode). After locating the VMA, we unlock the range lock, and lock it again, this time in write mode and with the range adjusted to span the entire VMA (plus some small extra space, as we explain below). Note that during the time the range lock is not held, mm_rb may change and, in particular, the VMA returned by find_vma() might not be valid anymore. We use a sequence number mentioned above to detect this scenario. Specifically, we read the sequence number right before dropping the read range lock and compare it to the number read right after acquiring the write range lock. If those numbers differ (or the boundaries of the found VMA have changed), the speculation fails, and we restart the mprotect operation from the beginning. We note that it is trivial to limit the number of retries, although we do not do that in our prototype implementation.

In case the speculation can proceed, we continue with the operation by going through the logic of identifying the required changes to the VMA(s) involved in the given mprotect operation. If this logic identifies that the changes require a structural modification to mm_rb, the speculation fails, the write range lock is dropped, and the mprotect operation is restarted by acquiring the write range lock for the full range. Otherwise, the mprotect operation completes while holding the write range lock for the relevant range only, thus allowing parallelism with other mprotect operations and/or operations that acquire the range lock for read (e.g., page faults discussed in the next section).

```
1  mprotect(__u64 addr, size_t size, int prot_flags):
2    __u64 start = addr
3    __u64 end = addr + size
4    bool speculate = true
5
6    while true:
7      if speculate: range_read_lock(range_lock, start, end)
8      else:         range_full_write_lock (range_lock)
9
10     vm_area_struct *vma = find_vma(addr)
11
12     if speculate:
13       __u64 seq_number = mm->seqnumber
14       __u64 aligned_start = vma->start - 4096
15       __u64 aligned_end = vma->end + 4096
16
17       range_read_unlock(range_lock)
18       range_write_lock(range_lock, aligned_start, aligned_end)
19
20       if seq_number != mm->seqnumber or
21          aligned_start != (vma->start-4096) or
22          aligned_end != (vma->end+4096):
23          ## validation failed, retry
24          range_write_unlock(range_lock);
25          continue
26
27     ##apply mprotect logic
28     ...
29     if speculate and will perform structural modification:
30       range_write_unlock(range_lock)
31       speculate = false
32       continue
33     ...
34     release_write_unlock (range_lock)
35     return
```

Listing 3. Simplified pseudo-code for the speculative mprotect implementation.

We are left to describe one subtle detail of determining the size of the range for the write acquisition during speculation. We note that it is not enough to lock only the underlying VMA of the given mprotect operation. This is because as discussed in Section Section 5.1, two mprotect operations on neighboring VMA structures can change the metadata of one another concurrently, thus creating a race condition. To avoid this situation, we set the range of the write range lock acquisition to the underlying VMA plus a page (4096 bytes) from each side of the VMA.

While the speculative mechanism described in this section is presented in the context of mprotect, we note that a similar mechanism can be employed in other operations as well. For instance, mmap, munmap and brk all start from calling find_vma (or a similar function), during which the range lock can be held in the read mode. Those operations, however, typically (but not always) end up modifying mm_rb, and thus would need to drop the read range lock and acquire

the write range lock for the entire range. Thus, the speculative approach would shorten the time during which the write range lock is held at the cost of an extra (read) range lock acquisition. Evaluating the effect of this speculation is left for future work.

5.3 Page Faults

Page fault interrupts access the VM subsystem to identify whether the address that triggered the fault is allowed to be accessed. They do so by locating the appropriate VMA (by calling the same `find_vma()` function) and then handling the fault based on that VMA's metadata (such as protection flags). Since the page fault routine only queries the metadata of VMA structures (but does not change them), it acquires the range lock in read mode. The original patch that introduced range locks into the Linux kernel, however, does all the acquisitions, including the one in the page fault routine, for the full range [5].

We observe that the page fault routine accesses only the metadata of the VMA returned by `find_vma()`. Therefore, it is straightforward to refine the range of the lock acquisition to contain only the given address (in our implementation, we lock the range of a page size). We note that any modification to `mm_rb` is done while holding the write range lock for the full range, while any modification to VMA metadata is done while holding the write range lock (at least, according to Section 5.2) that covers the range being modified. Therefore, the refinement of the range of the lock acquired in page faults is safe. Furthermore, note that this refinement alone is not expected to improve the scalability of the VM subsystem, because the range lock is acquired in read mode, similarly to the original `mm_sem`. However, when coupled with the speculation in `mprotect`, page fault interrupts can now lock and access VMA structures in parallel with some (or at least part) of the `mprotect` operations.

6 Range Lock-based Skip Lists

In this section, we show how range locks can be used to coordinate concurrent accesses to a skip list. We base our design on the optimistic skip list by Herlihy et al. [19]. In the original design, each node is associated with a spin lock. Search operations are wait-free, and in particular do not acquire any locks. Update operations start by searching the list for the given key, locking all relevant nodes (we elaborate on that below) and validating that the list has not changed in a way that precludes completing the operation (e.g., the node we want to delete is still in the list), perform the required update (removing the node from the list, or inserting a new node), and finally unlock all the acquired locks. If the validation above fails, the operation releases all the locks it has acquired, and restarts.

When replacing the per-node spin lock with a single range lock, we maintain the same properties. In particular, the search operations are still wait-free, which is important for read-dominated workloads. The major change is in the locking protocol. The original optimistic skip list acquires node-level locks for all the predecessors of the node returned by search (in case of a remove operation) or of the node with a key larger than the given key (in case of an insert operation). Note that each node has between 1 and N predecessors, where N is the number of levels in the skip list, and thus the locking protocol consists of between 1 and N lock acquisitions. In addition, remove operations acquire the lock of the target node to be deleted, adding one more lock acquisition to the locking protocol. With range locks, we always need to acquire one range only. For inserts, the range is the interval between the key of the predecessor at the highest level (at which the new node will be inserted) and the target key (to be inserted). For removes, the range is defined from the key of the predecessor at the highest level to the target key (to be removed) plus 1; the latter is to avoid races with inserts that may attempt to update pointers in the to-be-deleted node.

We note that beyond the conceptual simplicity and the potential performance benefits stemming from the fact that each operation acquires at most one (range) lock, the range lock-based skip list has a smaller memory footprint than its original lazy counterpart. This is due to elimination of spin locks associated with every node in the skip lists. As the number of nodes in skip lists is typically (much) larger than the number of concurrent threads updating the skip list, this may translate into significant memory savings.

7 Performance Evaluation

7.1 User-space

In this section, we evaluate our linked list-based range locks using two user-space applications.

We start with ArrBench, a microbenchmark that we developed in which threads access a range of slots of a shared array for either read or write. This benchmark allows us to assess the performance of our range locks in different contention scenarios. Array slots are padded to the size of a cache line. In read mode, a thread reads the values stored in each slot in the given range, while for write a thread increments the value stored in each slot by 1. Each operation acquires a range lock for the corresponding range, and in the corresponding access mode (read or write). Between operations on the array, each thread performs some (non-critical) work, emulated by a variable number of no-op operations. The number of no-op operations is chosen uniformly randomly from the given range (2048 in our case). We set the size of the array (i.e., the number of slots) to 256.

To simulate various levels of contention and possible usage scenarios for range locks, we created three variants of the ArrBench: in the first variant, each thread acquires the entire range of the array. In the second variant, each thread acquires a non-overlapping range calculated by dividing the size of the array by the number of threads. Note that in this

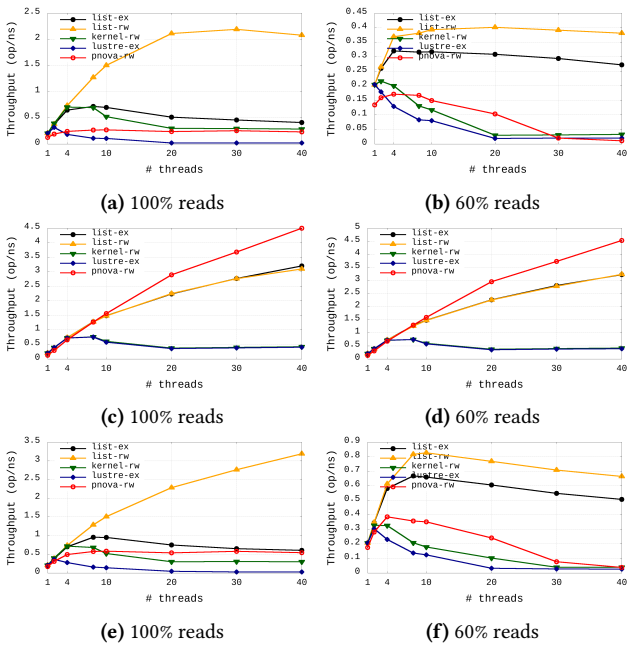
variant, threads do not conflict on the ranges they acquire. Furthermore, in order to keep the amount of work (i.e., the number of slot accesses) performed under the range lock the same independent of the number of threads, in this variant only, threads traverse the corresponding portion of the array the number of times equal to the number of threads. In other words, when this variant is run with one thread, that thread would traverse the entire array once for every acquisition of the range lock; when run with two threads, each of the threads would traverse half of the array twice for every acquisition of the range lock, and so on. Finally, in the third variant, each thread picks random starting and ending points from the range defined by the size of the array[4], acquires the range lock with that range, and performs one traversal of corresponding slots.

We implemented the mutex and reader-writer variants of the range lock described in the paper (without the fast path and fairness optimizations – we leave the evaluation of those for future work). We denote those variants as `list-ex` and `list-rw`, respectively. We ported two implementations of range locks found in the kernel into the user-space, one found in the Lustre file system (denoted as `lustre-ex`) and another recently proposed by Bueso [4] (denoted as `kernel-rw`). As mentioned earlier, the latter is a reader-writer version of the former. In the user-space experiments, we used a simple test-test-and-set lock to implement a spin lock protecting the range tree in `lustre-ex` and `kernel-rw`. We note that the Linux kernel uses a slightly more sophisticated spin lock implementation [8, 12], however, this detail is insignificant in our context[5]. In addition, we implemented the recent proposal for range locks by Kim et.al. [22]. Those locks were proposed in the context of pNOVA, a variant of a non-volatile memory file system, hence we denote this version of range locks as `pnova-rw`. As described in Section 2, `pnova-rw` operates with a present number of segments, each of a preset size [22]; in our experiments we set this number to 256 segment, spanning one array slot each. We also experimented with other number of segments, spanning multiple slots; although the results were quantitatively different, they lead to similar conclusions.

We ran the experiments on a system with two Intel Xeon E5-2630 v4 sockets featuring 10 hyperthreaded cores each (40 logical CPUs in total) and running Fedora 29. We did not pin threads to cores, relying on the OS to make its choices. We also disabled the turbo mode to avoid the effects of that mode (which may vary with the number of threads) on the results. We vary the number of threads between 1 and 40, as well as the mix of operations performed by each thread (100% reads, 80% reads and 20% writes, and 60% reads and 40% writes). The results for the 80% reads workload were

[4]We select starting and ending points randomly modulo the size of the array, and switch if the former is larger than the latter.

[5]To confirm that, we tried a different lock and observed similar relative performance results.



(a) 100% reads **(b)** 60% reads

(c) 100% reads **(d)** 60% reads

(e) 100% reads **(f)** 60% reads

Figure 3. Throughput for the ArrBench microbenchmark, where all threads acquire the entire range (first row), threads acquire non-overlapping ranges (second row) and threads acquiring random ranges (third row).

similar to the 60% reads workload and thus omitted. Each reported experiment has been run 5 times in exactly the same configuration. Presented results are the mean of throughput results reported by each of those 5 runs, where throughput is calculated based on the total number of operations performed by all the threads running for ten seconds. The standard deviation of nearly all results is less than 3% of the mean.

The results for the first variant of ArrBench, in which threads acquire and access the entire range, are shown in Figure 3 (a) and (b). The `lustre-ex` variant does not scale at all, it allows only one thread to traverse the array at a time as it does not support reader-writer semantics. Moreover, all threads contend heavily on the spin lock protecting the range tree structure. This is not the case for the `list-ex`, where the fact the threads perform non-critical work without a range lock helps it to scale for low thread counts. In fact, in most cases `list-ex` performs better than `kernel-rw`, even though the latter allows readers run concurrently. Once again, the spin lock protecting the underlying range tree plays detrimental role in the performance of `kernel-rw`. The `pnova-rw` variant also does not scale due to the high lock acquisition latency (acquiring this lock for the entire range requires acquiring all the underlying segment reader-writer locks). At the same time, `list-rw` does not use locks in the common case, and shows scalability across most thread counts.

The results for the second variant of ArrBench, in which each thread acquires a non-overlapping part of the range, are shown in Figure 3 (c) and (d). Note that the maximum

number of concurrent range accesses is equivalent to the number of threads depicted on the x-axis, which determines the size of the list (or the tree) in the corresponding range lock implementation. In theory, in this case the total throughput should scale with the number of threads for every range lock, as threads never compete for the same range (regardless of the access mode). In practice, however, all range locks scale almost linearly up to a small number of threads (4–8). Beyond that, the contention on the spin lock in `lustre-ex` and `kernel-rw` degrades the performance of those variants. `list-ex` and `list-rw` lack a single point of contention, and manage to scale, albeit less than linearly, across all thread counts. `pnova-ex` tops the charts as in this workload none of its underlying segment reader-writer locks is contended.

When considering the results for the third variant of ArrBench, in which each thread acquires a random part of the range (see Figure 3 (e) and (f)), one can note a mix of behaviors seen in the previous two variants. Overall, `lustre-ex` does not scale, `kernel-rw` scales up to a small number of threads, while `list-ex` either slightly better than (in read-only workload) or significantly outperforms (when workloads include writes) `kernel-rw`, despite providing only exclusive access to each range. `pnova-ex` performs poorly as its underlying reader-writer locks are once again contended. At the same time, `list-rw` provides superior performance across all workloads, scaling better than any other variant.

Next, we used the Synchrobench benchmark [17] to evaluate the performance of new skip lists that employ range locks to synchronize concurrent access, as discussed in Section 6. We compare three variants: the original optimistic skip list [19] (provided in Synchrobench, denoted as `orig`), and two variants of our new skip list that uses a range lock, one built on top of the Lustre range locks (denoted as `range-lustre`) and another on top of the exclusive list-based range lock presented in Section 4.1 (denoted as `range-list`). As it is not clear how one should set the number and the size of segments in pNOVA range locks, we do not include that lock in the evaluation of skip lists.

Figure 4 shows the results for the typical set workload composed of 80% find and 20% update operations (split evenly between inserts and removes); the key range is 8M, and 4M keys are randomly selected and inserted into the skip list before each experiment. We report the mean throughput after repeating each experiment 5 times (here as well the standard deviation is less than 3% of the mean for nearly all data points). The results show that `range-list` performs similarly to `orig`, even though the former is simpler and consumes less memory as it does not use a lock per skip list node. `range-lustre` tracks both versions at lower thread counts. Once thread counts grow, however, the contention on its internal spin lock increases, and as expected, its performance drops to less than half of the other two variants. This workload demonstrates that the increased concurrency allowed by `range-list` outweighs the linear complexity of the linked list, in contrast with the logarithmic complexity of `range-lustre`'s range tree.

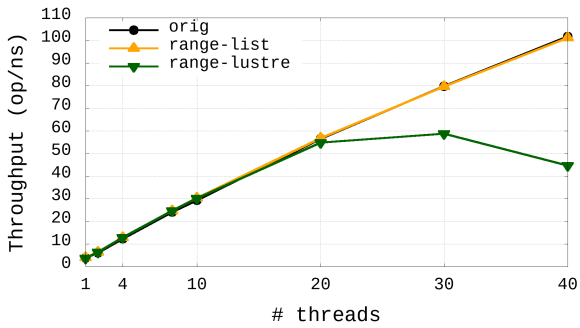


Figure 4. Throughput for the skip list benchmark.

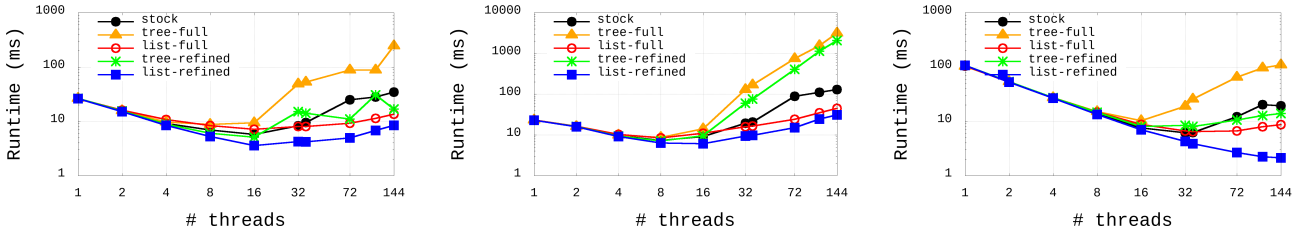
7.2 Kernel-space

For the kernel-level experiments, we compared the stock version (4.16.0-rc2) with the one that has `mm_sem` replaced with a range lock. For the latter, we used the patch by Bueso [5]; we call this variant `tree-full` as it always acquires the range lock for the full range. Based on this patch, we replaced the range lock implementation with the reader-writer linked list-based one described in this paper; we call this variant `list-full`. Furthermore, we refined the ranges of the acquired range locks as described in Section 5. We refer to the variants with refined ranges as `tree-refined` and `list-refined`, respective of the range lock implementation used by each. All the variants were compiled in the default configuration.

We ran the experiments on a system with four Intel Xeon E7-8895 v3 sockets featuring 18 hyperthreaded cores each (144 logical CPUs in total). Like for user-space experiments, we do not pin threads to cores and disable the turbo mode. For our evaluation, we used Metis, an open source MapReduce library [23], known for stress-testing the VM subsystem through the mix of VM-related operations (such as page-faults, `mmap` and `mprotect`) [21]. Each experiment was repeated 5 times, and we report the mean of the results. The standard deviation of the majority of the results was below 5% of the mean.

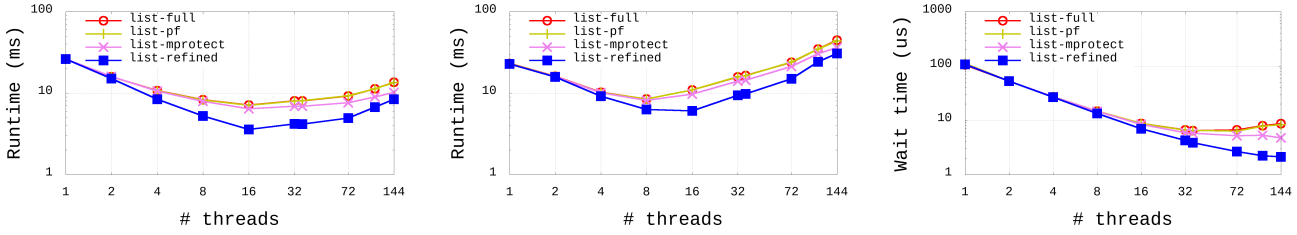
Through the tracing facility in the kernel (`ftrace`), we identified that three benchmarks in the Metis suite use `mprotect` extensively. Those applications are wc (word count), wr (inverted index calculation) and wrmem, which is a variant of wr that allocates a chunk of memory and fills it with random "words" instead of reading its input from a file. We used default input files for wc and wr, and 2GB input size for wrmem. The tracing also revealed that the majority of the calls to `mprotect` (over 99%) succeed in the speculative path. We note that in all other Metis benchmarks, which did not call `mprotect` as extensively as the other three benchmarks mentioned above, the impact of range locks was negligible.

Figure 5 shows the runtime results for wc, wr and wrmem



(a) wr **(b)** wc **(c)** wrmem

Figure 5. Runtime for Metis benchmarks.



(a) wr **(b)** wc **(c)** wrmem

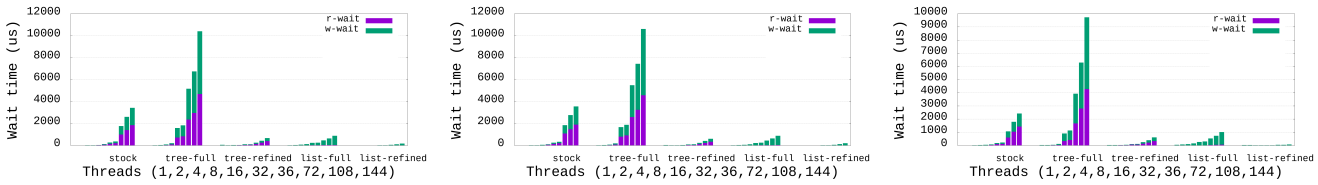
Figure 6. Breakdown of the impact of refining the range in list-based range lock variants.

(lower is better). Up to 8–16 threads, all variants perform similarly and scale linearly with the number for threads. However, once the thread counts increase, and with them the contention on the VM subsystem, the variants produce different results. Notably, the performance of the stock version worsens with the increased contention, while the list-based range lock variants remain mostly flat, or continue to scale, as in the case of wrmem and `list-refined`. In general, the tree-based range locks perform worse than the list-based ones, and mostly worse even when compared with the stock version. We believe this is at least in part because of the contention created on the spin lock protecting the access to the range tree. Refining the ranges of the range lock acquisitions helps both tree-based and list-based variants, i.e., `tree-refined` outperforms `tree-full`, while `list-refined` outperforms `list-full`. In fact, at 144 threads, `list-refined` has 9× speedup over `stock` in wrmem. Again, similar to our observation in the user-space skip list experiment, the higher parallelism achieved by list-based range locks outweighs the linear complexity of list traversal, even with large number of concurrent ranges (144 in this case).

It is interesting to note that `list-full` outperforms `stock` under high contention despite always acquiring the range lock for the full range. We conjecture that this is due to the different waiting policies employed by those two variants. Specifically, `stock` uses a read-write semaphore (`mm_sem`), in which threads block (after spinning for a while if optimistic spinning is enabled) when the semaphore is unavailable until they are waken up by another thread. In `list-full` (and `list-refined`), threads block for a small period of time if the range is unavailable and recheck the range, which turns to be more efficient under contention. Exploring different waiting policies and their impact on lock performance is an active area of research [10, 21].

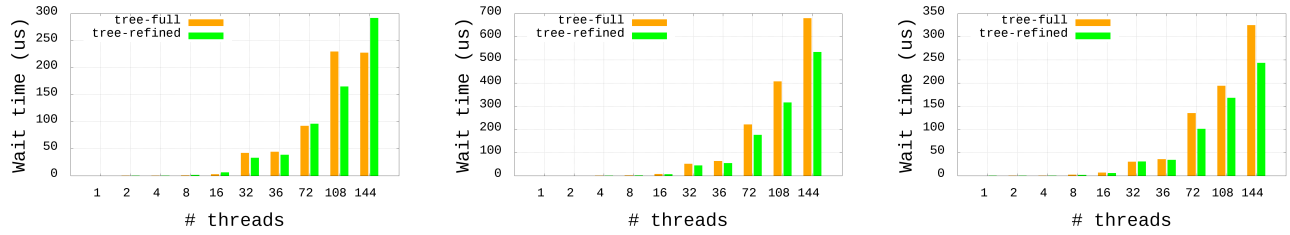
Figure 6 drills down into the effect of refining ranges on the performance of the list-based range locks. Here `list-pf` (`list-mprotect`) denotes the variant where only the range in the page fault routine (`mprotect` operation, respectively) is refined. As expected, the refinement in the page fault routine does not have much effect, since the range lock is acquired there for read while in all other places it is acquired for the full range. At the same time, refining the range in `mprotect` has a small, but positive effect as now `mprotect` operations on non-overlapping ranges can be applied concurrently. As Figure 6 shows, however, it is the combination of the two optimizations that makes a difference – `list-refined`, which refines the range in both page faults and `mprotect` and thus allows their concurrent execution, substantially outperforms all other variants.

Through the `lock_stat` mechanism built into kernel,we collected statistics on the time threads spent waiting for various locks in the kernel. (The `lock_stat` mechanism is known to introduce a probe effect [12], therefore it was enabled only for runs in which we collected statistics on lock wait times.) In Figure 7 we plot the average wait times for `mm_sem` (in the stock variant) as well as for the range lock in all other variants, breaking down between read and write acquisitions. Not surprisingly, those results show a (rough) correlation between high wait times and poor scalability. They also reveal that with range refinement, the average



(a) wr **(b)** wc **(c)** wrmem

Figure 7. Average wait time for `mm_sem` (in stock) and range lock (in all other variants).



(a) wr **(b)** wc **(c)** wrmem

Figure 8. Average wait time for the spin lock protecting the range tree in `tree-full` and `tree-refined`.

wait times decrease.

Figure 8 shows the average wait time on the spin-lock protecting the range tree in the `tree-full` and `tree-refined` variants. Notice that the waiting time grows with the number of threads, supporting our hypothesis that this lock represents a point of contention. The range refinement does not change much the wait time for the spin lock. This is not surprising, as this lock is acquired for every acquisition of the range lock, regardless of whether or not the range is available. However, while in `tree-full` the wait time for the spin lock is relatively small compared to the wait time for the range lock itself (which includes waiting for a range to become available), in `tree-refined` takes the lion share of the range lock wait time (cf. Figure 7 and Figure 8). This underscores the effectiveness of range refinement in allowing parallel processing of the VM operations. That is, when the ranges are refined, most wait time for a range lock can be attributed to the wait time on the auxiliary spin lock rather than waiting for the range availability. Unlike tree-based range locks, list-based range locks do not have a central point of contention and thus can take better advantage of this parallelism, as demonstrated by the results in Figure 5.

8 Conclusion

In this paper, we presented the design and implementation of new scalable range locks. Those locks employ a simple underlying structure (a concurrent linked list) to keep track of acquired ranges. This structure allows simple lock-less modifications with just one atomic instruction. Therefore, our design avoids the pitfall of existing range locks, and does not require an auxiliary lock in the common case.

Furthermore, we show how range locks can be employed effectively to mitigate the contention on the access to the VM subsystem and its data structures, in particular, the red-black tree holding VMA structures. We achieve that through a speculative mechanism introduced into the `mprotect` operation; this mechanism allows to refine the range of the lock acquired in `mprotect`. We also refine the range of lock acquisitions in page fault routines. Together, those refinements allow parallel processing of page faults and `mprotects` operating on non-overlapping regions of VM space, which is particularly beneficial, e.g., for the standard GLIBC memory allocator. In addition, we demonstrate the utility of range locks for the design of concurrent, scalable data structures through the example of a range-lock based skip list.

We evaluate the scalability of the new range locks in user-space through several microbenchmarks and kernel-space through several applications from the Metis suite. The results show that the new range locks provide superior performance compared to the existing range locks (in the user-space and kernel), as well as to the current method of VM subsystem synchronization in the kernel (that uses a read-write semaphore). Future work includes evaluating range locks with additional benchmarks, and exploring the usage of range locks in other contexts, such as parallel file systems [22] and as building blocks for other concurrent data structures, such as hash tables and binary search trees.

Acknowledgments

We thank the anonymous reviewers as well as our shepherd Dilma Da Silva for valuable comments and suggestions to improve the quality of our manuscript.

References

[1] P. M. Aarestad, A. Ching, G. K. Thiruvathukal, and A. N. Choudhary. 2006. Scalable Approaches for Supporting MPI-IO Atomicity. In *Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*, Vol. 1. 35–42.

[2] AT&T. 1986. UNIX System V User's Manual Volume 1. http://bitsavers.trailing-edge.com/pdf/att/3b1/999-801-312IS_ATT_UNIX_PC_System_V_Users_Manual_Volume_1.pdf Accessed: 2019-04-15.

[3] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2010. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 1–16.

[4] Davidlohr Bueso. 2017. locking: Introduce range reader/writer lock. https://lwn.net/Articles/722741/, May 15, 2017. Accessed: 2018-10-29.

[5] Davidlohr Bueso. 2018. mm: towards parallel address space operations. https://lwn.net/Articles/746537/, Feb 5, 2018. Accessed: 2019-04-15.

[6] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. 2012. Scalable Address Spaces Using RCU Balanced Trees. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 199–210.

[7] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. 2013. RadixVM: Scalable Address Spaces for Multithreaded Applications. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*. 211–224.

[8] Jonathan Corbet. 2014. MCS locks and qspinlocks. https://lwn.net/Articles/590243, March 11, 2014. Accessed: 2018-10-29.

[9] Jonathan Corbet. 2017. Range reader/writer locks for the kernel. https://lwn.net/Articles/724502, June 5, 2017. Accessed: 2018-09-28.

[10] Dave Dice. 2017. Malthusian Locks. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*. 314–327.

[11] Dave Dice and Alex Kogan. 2019. BRAVO: Biased Locking for Reader-writer Locks. In *Proceedings of the Usenix Annual Technical Conference (USENIX ATC)*. 315–328.

[12] Dave Dice and Alex Kogan. 2019. Compact NUMA-aware Locks. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*. 12:1–12:15.

[13] Laurent Dufour. 2017. Replace mmap_sem by a range lock. https://lwn.net/Articles/723648/, May 24, 2017. Accessed: 2018-10-29.

[14] L. Torvalds et al. 2020. Linux source code. http://www.kernel.org/. Accessed: 2020-03-10.

[15] Jose M. Faleiro and Daniel J. Abadi. 2017. Latch-free Synchronization in Database Systems: Silver Bullet or Fool's Gold?. In *Proceedings of Conference on Innovative Data Systems Research (CIDR)*.

[16] K. Fraser. 2004. *Practical lock-freedom.* Ph.D. Dissertation. University of Cambridge.

[17] Vincent Gramoli. 2015. More Than You Ever Wanted to Know About Synchronization: Synchrobench, Measuring the Impact of the Synchronization on Concurrent Algorithms. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*.

[18] Thomas E. Hart, Paul E. McKenney, Angela Demke Brown, and Jonathan Walpole. 2007. Performance of Memory Reclamation for Lockless Synchronization. *J. Parallel Distrib. Comput.* 67, 12 (2007), 1270–1285.

[19] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. 2007. A Simple Optimistic Skiplist Algorithm. In *Proceedings of the 14th International Conference on Structural Information and Communication Complexity (SIROCCO)*.

[20] Jan Kara. 2013. lib: Implement range locks. https://lkml.org/lkml/2013/1/31/483, January 31, 2013. Accessed: 2018-09-28.

[21] Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. 2017. Scalable NUMA-aware Blocking Synchronization Primitives. In *Proceedings of the Usenix Annual Technical Conference (USENIX ATC)*.

[22] June-Hyung Kim, Jangwoong Kim, Hyeongu Kang, Chang-Gyu Lee, Sungyong Park, and Youngjae Kim. 2019. pNOVA: Optimizing Shared File I/O Operations of NVM File System on Manycore Servers. In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys)*. 1–7.

[23] Yandong Mao, Robert Morris, and Frans Kaashoek. 2010. *Optimizing MapReduce for Multicore Architectures.* Technical Report. MIT.

[24] Paul E. Mckenney, Silas Boyd-wickizer, and Jonathan Walpole. 2012. *RCU usage in the Linux kernel: One decade later.* Technical Report.

[25] Paul E. McKenney and Jack Slingwine. 1998. Read-copy-update: Using Execution History to Solve Concurrency Problems. In *Parallel and Distributed Computing and Systems*. 509–518.

[26] Maged M. Michael. 2004. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Trans. Parallel Distrib. Syst.* 15, 6 (2004), 491–504.

[27] Xiang Song, Jicheng Shi, Ran Liu, Jian Yang, and Haibo Chen. 2013. Parallelizing Live Migration of Virtual Machines. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*. 85–96.