



CHAIR OF DECENTRALIZED
INFORMATION SYSTEMS & DATA
MANAGEMENT

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Concurrent Range Locking

Thua-Duc Nguyen



CHAIR OF DECENTRALIZED INFORMATION SYSTEMS & DATA MANAGEMENT

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Concurrent Range Locking

| | |
|-----------------|-----------------------|
| Author: | Thua-Duc Nguyen |
| Supervisor: | Prof. Dr. Viktor Leis |
| Advisor: | Lam-Duy Nguyen |
| SubmissionDate: | 15.09.2024 |



I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

München, 15.09.2024

Thua-Duc Nguyen

Acknowledgments

I would like to express my sincere gratitude to my dedicated supervisor Prof. Dr. Viktor Leis and advisor Lam-Duy Nguyen, for their unwavering support and guidance. They have consistently provided support and motivation, even amid their busy schedule.

My heartfelt appreciation to my girlfriend, Thuy-Trang Nguyen, for standing by me through the challenges of my thesis and undergraduate journey. She has consistently provided support and motivation.

Eventually, I would like to send a special acknowledgment to my family and friends whose encouragement has been a source of strength and an integral part during the course of my studies.

I would not have been able to accomplish this work without the support of each of these people.

Abstract

Large-scale distributed systems and applications often face significant challenges in managing concurrent access to shared resources. For instance, high-performance computing environments frequently require efficient handling of numerous simultaneous access requests to various parts of data sets. Similarly, distributed memory systems must support multiple clients accessing the same memory space concurrently, often with diverse access patterns. In such contexts, it becomes crucial to coordinate and manage these concurrent accesses effectively to ensure correctness and performance.

Concurrent range locks address these challenges by providing mechanisms for efficient and accurate management of overlapping and non-overlapping ranges of resources, thereby enabling scalable and reliable access control in complex, high-demand environments.

Previous studies have examined various range lock techniques, with the Linux kernel currently employing a range tree accompanied by a spin lock for managing access to ranges. This single spinlock, however, introduces bottlenecks. Song et al. proposed an enhancement by integrating a skip list with the spinlock, offering a more efficient and less heavyweight solution than traditional interval trees, though challenges with contention persist. Alternatively, Kogan et al. developed a lock-free range lock utilizing a concurrent linked list, each node representing an acquired range. This method addresses limitations found in existing range locks but at the cost of reduced efficiency in insertion and search operations compared to tree-based structures.

In the scope of this research, we propose a new lock-free concurrent range-locking design. We address previous bottleneck issues by leveraging a probabilistic concurrent skip list. By employing the atomic `compareAndSet` function, we eliminate the need for locks, thereby ensuring robust concurrency performance. The proposed mechanism will be developed and evaluated under heavy concurrent access, ensuring correctness in overlapping ranges and concurrent operations. Performance comparisons with existing state-of-the-art approaches will comprehensively assess its effectiveness.

Contents

| | |
|------------------------------------------------------------------|------------|
| Acknowledgments | iii |
| Abstract | iv |
| List of Figures | 1 |
| List of Tables | 2 |
| 1 Introduction | 3 |
| 2 Related Work | 5 |
| 2.1 Coarse-Grained Range Lock | 5 |
| 2.1.1 Tree-Based Range Lock | 5 |
| 2.1.2 List-Based Range Lock | 6 |
| 2.2 Fine-Grained Range Lock | 6 |
| 2.2.1 List-Based Range Lock | 6 |
| 2.2.2 Bitmap Range Lock | 7 |
| 3 Data structure | 8 |
| 3.1 Skiplist is suitable for range lock | 8 |
| 3.2 Modification of skiplist for concurrent range lock | 9 |
| 3.2.1 Node | 10 |
| 3.2.2 AtomicMarkableReference | 11 |
| 4 Approach | 13 |
| 4.1 Concurrent Range Lock | 13 |
| 4.1.1 Concurrent Range Lock API | 15 |
| 4.1.2 Find | 15 |
| 4.1.3 Try Lock | 19 |
| 4.1.4 Release Lock | 22 |
| 5 Evaluation | 24 |
| 5.1 Competitor | 24 |

Contents

| | | |
|----------|--------------------------------------------|-----------|
| 5.2 | Benchmark enviroment | 24 |
| 5.3 | Microbenchmark | 24 |
| 5.3.1 | Workload | 24 |
| 5.3.2 | Optimal Height for Range Locking | 25 |
| 5.3.3 | Result | 27 |
| 5.4 | Leanstore | 29 |
| 5.4.1 | Competitors | 29 |
| 5.4.2 | Workload | 29 |
| 5.4.3 | Result | 30 |
| 6 | Conclusion | 33 |
| 7 | Future work | 34 |
| | Bibliography | 35 |

List of Figures

| | | |
|-----|----------------------------------------------------------------------------------------------------------|----|
| 3.1 | An example of a skip list. It has five levels of sorted linked lists. | 9 |
| 4.1 | ConcurrentRangeLock. Each Node has a unique range. | 13 |
| 5.1 | Runtime for different heights of range lock | 26 |
| 5.2 | Microbenchmark for workload W1 | 28 |
| 5.3 | Microbenchmark for workload W2 | 28 |
| 5.4 | Comparison of transactions by increasing the number of thread workers. | 30 |
| 5.5 | Comparison of cycles, instructions, and L1-misses by increasing the number of thread workers. | 32 |

List of Tables

| | | |
|-----|--------------------------------------------------------------------------------------------------------------|---|
| 3.1 | Time complexities of Insert/Delete operations in Linked List, Skip List, Interval Tree, and Bitmap | 9 |
|-----|--------------------------------------------------------------------------------------------------------------|---|

1 Introduction

In modern computing environments, concurrent threads must often synchronize access to overlapping shared resources, such as database tables, file regions, or memory segments. Systems often rely on coarser synchronization methods, like using a single lock to manage the access among threads. This leads to significant inefficiencies and performance bottlenecks under high concurrency. Range lock [1, 2, 3] provides a more refined approach to this challenge by partitioning a shared resource into multiple arbitrary-sized segments, so that different processes can exclusively acquire and access each of these segments concurrently. This strategy effectively addresses the drawbacks and bottlenecks associated with single-lock methods, allows for more granular control and improves parallelism.

Range lock is needed across various systems. Range lock is an under-studied topic, while it is important across various systems, including DBMS, file systems, and operating systems. In DBMS, range lock ensures data consistency and prevents anomalies like "phantoms," especially under the serializable isolation level. They secure not only individual records but also the gaps between them, preventing other transactions from inserting or modifying records within the range until the transaction is complete. This is particularly challenging in systems where transaction control (TC) and data control (DC) are separated. TC must lock the range before safely interacting with DC, despite not knowing the specific keys involved. Similarly, in high-performance file systems, range lock enables multiple processes to access different file segments simultaneously, reducing contention and enhancing performance, especially in large-scale, distributed environments. For operating systems, particularly within the Linux kernel, range lock is increasingly vital for alleviating contention issues associated with `mmap_sem`, a semaphore that manages access to virtual memory areas (VMAs). As applications with large, dynamically allocated memory grow, contention on this semaphore can significantly hinder performance, making range lock a critical tool for maintaining efficiency and scalability across these diverse computing environments.

Existing range lock is not efficient. Previous implementations of range-locking mechanisms often need to improve their performance. These implementations often suffer from contention points due to the reliance on a single lock [4, 3], as spinlock effectively

serializes all incoming lock and unlock requests. Additionally, some methods may be complex and tightly coupled with lock-based concurrency control protocols, which are not applicable for general DBMS operations [5, 6]. These limitations highlight the necessity for more refined and scalable solutions to better accommodate the demands of modern, large-scale systems. This drives us to develop an enhanced range lock.

Contribution. In the scope of this thesis, we introduce a new lock-free concurrent range lock. Our method addresses the inefficiencies of previous range locks and ensures high performance in highly concurrent environments. Result shows that our method is at least three times faster than existing solutions. This thesis provides an in-depth exploration of the development and evaluation of the proposed range-locking mechanism, offering a thorough understanding of its performance under heavy concurrent access. Furthermore, we compare the proposed solution with state-of-the-art methods, providing a clear assessment of its effectiveness.

2 Related Work

2.1 Coarse-Grained Range Lock

2.1.1 Tree-Based Range Lock

Several works have explored coarse-grained range-locking mechanisms. Jan Kara introduced a range-locking mechanism for the Linux kernel [4], which utilizes a range tree (specifically a red-black tree) to manage range locks and employs a spinlock for synchronization. Each lock is represented as a node in the tree. Similarly, Kim et al. adopted a comparable range-locking mechanism in their work on pNOVA [7], a variant of the NOVA file system that uses range-based reader-writer locks to enable parallel I/O within a single shared file.

When a thread requests a range lock, it first acquires a spinlock, then traverses the tree to determine the number of locks intersecting with the requested range. Afterward, the thread inserts a node describing its range into the tree and releases the spinlock. If no intersecting locks are found, the thread can proceed with accessing the critical section. If intersecting locks are detected, the thread waits until those locks are released and the number of intersecting locks drops to zero. Upon completing its operation, the thread re-acquires the spinlock, removes its node from the tree, updates the count of overlapping locks, and releases the spinlock. This method ensures that each range is locked only after all previous conflicting range locks have been released, thereby achieving fairness and avoiding livelocks.

Drawbacks

One significant observation is that the coarse-grained spinlock of an interval tree can severely hinder parallelism, as the spinlock effectively serializes all incoming lock and unlock requests. Under heavy concurrent access, this serialization easily becomes a contention point, limiting the system's performance.

Consider three exclusive lock requests for the ranges $A = [1..3]$, $B = [2..7]$, and $C = [4..5]$, arriving in that order. While A holds the lock, B is blocked because it overlaps

with A, and C is blocked behind B. However, in practice, C does not overlap with A and could proceed without waiting. This unnecessary blocking reduces the overall efficiency and concurrency of the system.

2.1.2 List-Based Range Lock

Song et al. [3] introduced a dynamic range-locking design to enhance the implementation of the Linux kernel. Their range lock uses a skip list [8] to dynamically manage the address ranges that are currently locked.

When a thread requests a specific range $[start, start+len)$, the range lock searches the skip list. If an existing or overlapping range is found, it indicates that another thread is currently modifying that range, requiring the requesting thread to wait and then retry. If no overlapping range is found, the requested range is added to the skip list, signifying that the lock has been acquired. Releasing a range involves deleting the corresponding range from the skip list.

Compared to the interval tree, the skip list is more lightweight and efficient, allowing for quicker searches of overlapping ranges.

Drawbacks

Similar to the tree-based range lock, contention remains an issue with this approach. Additionally, it unnecessarily blocks non-conflicting requests, further reducing system efficiency and limiting concurrency.

2.2 Fine-Grained Range Lock

2.2.1 List-Based Range Lock

Kogan et al. [2] introduced a range lock based on a concurrent linked list, where each node represents an acquired range. This design aims to provide a lock-free mechanism, addressing critical shortcomings of previous range-locking implementations. In a lock-free system, processes can proceed without being blocked by locks held by other processes, thereby improving performance and scalability.

The proposed method involves inserting acquired ranges into a linked list sorted by their starting points, ensuring that only one range from a group of overlapping ranges can be inserted using an atomic compare-and-swap (CAS) operation. A significant

difference in this method compared to previous ones is that each node has two statuses: marked (logically deleted) or unmarked (present).

When a thread wants to acquire a range, it iterates through the skip list. If it encounters a marked node, it removes it using CAS and continues to iterate. If the current node protects a range that overlaps, the thread waits until that node is deleted. Otherwise, a node is inserted into the list, signaling that the range is acquired. To release a range, the thread marks the node as deleted.

Drawbacks

Linked List Inefficiency: While this design implements a lock-free mechanism that effectively addresses the limitations of existing range locks, it comes with its own set of trade-offs. In general, insertion and lookup operations in a linked list are less efficient than in tree-like structures. The average time complexity for searching in a linked list is $O(n)$, whereas it is only $O(\log n)$ for skip lists or tree-like structures [9]. Our evaluation will demonstrate that this inefficiency becomes particularly pronounced when handling multiple overlapping ranges within the list.

2.2.2 Bitmap Range Lock

In addition to the tree-based range locking method discussed in 2.1.1, Kim et al. proposed a lock-free range locking mechanism, which they claim offers enhanced efficiency compared to interval tree-based locks. This approach involves dividing a file into segments, each managed by a 32-bit variable that functions as a reader-writer lock. The most significant bit represents the writer lock status (1 for locked, 0 for unlocked), while the remaining 31 bits count the number of active readers. The mechanism utilizes hardware-supported atomic operations to ensure that writer locks can only be set when no other locks are active and that reader locks are granted as long as no writer lock is present. Unlocking is achieved by clearing the writer lock bit and decrementing the reader counter.

Although this method provides finer-grained locking with reduced overhead compared to interval tree-based locks, it is tailored to handle both reader and writer modes and depends on specific memory size constraints. Since our research focuses exclusively on the exclusive mode and does not address the reader-writer combination, we have chosen not to consider this approach as a competitor in our project.

3 Data structure

3.1 Skiplist is suitable for range lock

The effectiveness of a range lock is strongly influenced by the underlying data structure used to manage these ranges. Existing range locks have used many different comparison-based data structures, such as interval tree [4, 7], linked list [3], skip list [2] and bitmap [7, 10], each of which has advantages and disadvantages.

Interval trees are considered heavyweight compared to linked lists and skip lists because they require complex balancing mechanisms to maintain efficient search times, which adds significant computational and memory overhead. Interval trees also require the maintenance of additional data, such as subtree intervals, making them more resource-intensive. This makes interval trees unsuitable for range locking.

Linked lists have minimal structure and are easy to implement and maintain. However, it is very expensive in terms of Insert/Delete time. The complexity $O(n)$ for the average case of search becomes a significant disadvantage when the number of ranges increases,

Bitmaps are another alternative, especially in scenarios where the size of the shared resource is known in advance. However, bitmaps are expensive for writing areas into the map. For example, in range locking based on the bitmap version from the work of Nguyen et al. [10], they write range directly on the map and reverse in case they reach a set bit. This results in an average case for lock operation of $O(n)$, where n is the size of the range, making it inefficient for range locking. In addition, bitmap range locks need to know the size of the shared object in advance. They need one bit for each block. So, even if we only have one or two areas in the lock, we would still need the total size of the bitmap, which is unnecessary.

Skiplists strike a balance between the heaviness of interval trees and the simplicity of linked lists. Despite their theoretically poor worst-case performance, skip lists rarely exhibit worst-case behavior, making them efficient in most scenarios. For example, in a dictionary with more than 250 entries, the probability that a search will take more than three times the expected time is less than one in a million [11]. Skip lists are, therefore,

ideal for implementing range lock.

| Data Structure | Best Case | Average Case | Worst Case |
|----------------------|-------------|--------------|-------------|
| Linked List | $O(1)$ | $O(n)$ | $O(n)$ |
| Skip List | $O(1)$ | $O(\log n)$ | $O(n)$ |
| Interval Tree | $O(\log n)$ | $O(\log n)$ | $O(\log n)$ |
| Bitmap | $O(1)$ | $O(n)$ | $O(n)$ |

Table 3.1: Time complexities of Insert/Delete operations in Linked List, Skip List, Interval Tree, and Bitmap

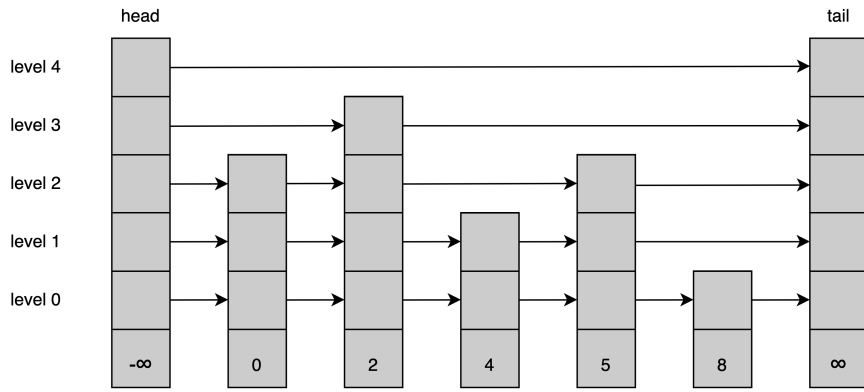


Figure 3.1: An example of a skip list. It has five levels of sorted linked lists.

3.2 Modification of skiplist for concurrent range lock

We developed our concurrent range lock based on the LockFreeSkipList proposed by Herlihy et al. [12]. In summary, LockFreeSkipList uses atomic operations (`compareAndSet()`) to manage Node references without locks, which enhances performance in multithreaded environments. When adding a Node, the process starts at the lowest level and moves upward to ensure immediate visibility. Removing a Node involves marking nodes from the top down before unlinking them. Furthermore, it relaxes strict structural maintenance of higher levels, focusing on the bottom-level list for set representation, which offers improved scalability and efficiency.

However, conventional skiplists does not work out of the box for range locking. Firstly, each Node of the skiplist stores key and value, not range. Secondly, the comparison condition needs to be modified because we now must compare both range start and

range end while inserting or deleting instead of just using the key. We present this in two critical changes in the ?? and 4.1.2 sections, respectively.

3.2.1 Node

Node is the base of our ConcurrentRangeLock structure. Each Node now contains start and end, which represents range. Node uses an array of AtomicMarkableReference (more details in section 3.2.2) to maintain forward links at each level, which allows for efficient traversal and updates. Node provides the following methods:

- `initialize`: sets up a Node with specific range and level values.
- `initializeHead`: configures the head Node with forward pointers directed to a provided tail Node, establishing the initial structure.
- `getTopLevel()`, `getStart()`, `getEnd()`: accessor methods to retrieve the Node's properties.

```
1 class Node {
2     private:
3         uint64_t start, end;
4         int topLevel;
5         AtomicMarkableReference** next = nullptr;
6
7     public:
8         void initialize(uint64_t start, uint64_t end, int topLevel) {
9             this->start = start; this->end = end;
10            this->topLevel = topLevel;
11
12            next = new AtomicMarkableReference*[topLevel + 1];
13            for (int i = 0; i <= topLevel; ++i) {
14                next[i] = new AtomicMarkableReference();
15            }
16        }
17
18        void initializeHead(uint64_t start, uint64_t end, int topLevel, Node* tail) {
19            initialize(start, end, topLevel);
20            for (int i = 0; i <= topLevel; ++i){
21                next[i]->store(tail, false);
22            }
23        }
24
25        int getTopLevel() const { return topLevel; }
26        uint64_t getStart() const { return start; }
27        uint64_t getEnd() const { return end; }
28    };
```

Listing 3.1: Pseudocode for Node structure

3.2.2 AtomicMarkableReference

The AtomicMarkableReference class uses a single atomic variable, `atomicRefMark`, to store a packed representation of both the reference (specifically, a Node) and a mark. If the mark is 1, it indicates that the Node it references is softly deleted. These values are packed and unpacked using bitwise operations, where the least significant bit represents the mark.

Listing 3.2 provides the pseudo code for AtomicMarkableReference.

The `pack` method combines a Node pointer and a boolean mark into a single `uintptr_t` value by encoding the pointer into the lower bits and the mark into the highest bit. Conversely, the `unpack` method decodes this packed value to retrieve the original Node pointer and boolean mark.

To atomically set a new Node pointer and mark value, the `store` method uses relaxed memory ordering.

The `compareAndSet` method performs an atomic update of both the reference and mark if they match the expected values, employing acquire-release memory ordering for proper synchronization.

The `attemptMark` method focuses on updating the mark alone, provided that the current reference matches the expected one and the mark differs. If the update succeeds, it returns `true`; otherwise, it returns `false`.

The `get` method retrieves the current reference and mark, which stores the mark in a provided boolean pointer. In contrast, the `getReference` method returns the current reference without accessing the mark.

```
1 class AtomicMarkableReference {
2     private:
3         std::atomic<uintptr_t> atomicRefMark;
4
5         uintptr_t pack(Node* ref, bool mark) const {
6             return reinterpret_cast<uintptr_t>(ref) | (mark ? 1 : 0);
7         }
8
9         std::pair<Node*, bool> unpack(uintptr_t packed) const {
10             return {reinterpret_cast<Node*>(packed & ~1), packed & 1};
11         }
12
13     public:
14         AtomicMarkableReference() {
15             atomicRefMark.store(pack(nullptr, false), std::memory_order_relaxed);
16         }
17
18         void store(Node* ref, bool mark) {
19             atomicRefMark.store(pack(ref, mark), std::memory_order_relaxed);
20         }
21
22         bool compareAndSet(Node* expectedRef, Node* newRef, bool expectedMark, bool newMark) {
23             return atomicRefMark.compare_exchange_strong(
24                 pack(expectedRef, expectedMark), pack(newRef, newMark), std::memory_order_acq_rel);
25         }
26
27         bool attemptMark(Node* expectedRef, bool newMark) {
28             auto [currentRef, currentMark] = unpack(atomicRefMark.load(std::memory_order_acquire));
29             if (currentRef == expectedRef && currentMark != newMark) {
30                 return atomicRefMark.compare_exchange_strong(
31                     current, pack(expectedRef, newMark), std::memory_order_acq_rel);
32             }
33             return false;
34         }
35
36         Node* get(bool* mark) const {
37             auto [ref, currentMark] = unpack(atomicRefMark.load(std::memory_order_acquire));
38             mark[0] = currentMark;
39             return ref;
40         }
41
42         Node* getReference() const {
43             auto [ref, _] = unpack(atomicRefMark.load(std::memory_order_acquire));
44             return ref;
45         }
46 };
```

Listing 3.2: AtomicMarkableReference

4 Approach

In this section, we will demonstrate how our concurrent range lock operates. In the first section ??, we will discuss the skip list and its suitability as a data structure for range locking. Then, we will delve into the details of our proposed mechanism in the second section 4.1.

4.1 Concurrent Range Lock

We developed our concurrent range lock based on the LockFreeSkipList proposed by Herlihy et al. [12]. In summary, our concurrent range lock uses atomic operation `compareAndSet` to manage Node references without locks, which enhances performance in multithreaded environments. When adding a Node, the process starts at the lowest level and moves upward to ensure immediate visibility. Removing a Node involves marking nodes from the top down before unlinking them. Furthermore, it relaxes strict structural maintenance of higher levels, focusing on the bottom-level list for set representation, which offers improved scalability and efficiency.

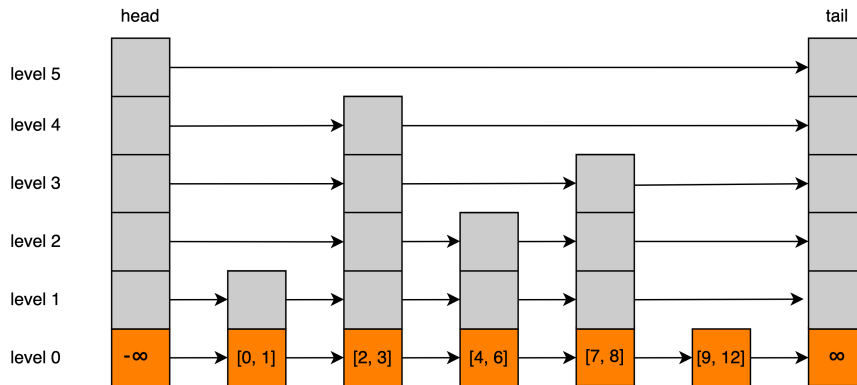


Figure 4.1: ConcurrentRangeLock. Each Node has a unique range.

A crucial modification is required to utilize a skip list for range locking. Instead of storing key-value pairs as in a conventional skip list, each Node will store a range. The

search, insert, and delete operations logic will be adapted and redefined as `tryLock` and `releaseLock` methods to accommodate this change.

In this section, we will provide a detailed explanation of the algorithm behind the concurrent range lock. For the sake of simplicity, we use `uint64_t` for our pseudocode provided in this section. We use the template feature in our open-source C++ implementation to enable generic programming. For further detail, please refer to our open-source code.

4.1.1 Concurrent Range Lock API

The ConcurrentRangeLock class consists of two main functions: tryLock and releaseLock.

The tryLock method attempts to acquire a lock for the specified range [start, end], returning true on success and false otherwise. The releaseLock method releases the lock for the range [start, end], with true indicating success and false if the range was not found or an error occurred. We will discuss these methods in subsection 4.1.3 and 4.1.4.

The two primary methods rely heavily on private searching methods such as findInsert, findExact, and findDelete, which handle insertion finding, exact range finding, and physical deletion of ranges, respectively. We will discuss these methods in section 4.1.2.

```
1 class ConcurrentRangeLock {
2     public:
3         ConcurrentRangeLock();
4
5         bool tryLock(uint64_t start, uint64_t end);
6         bool releaseLock(uint64_t start, uint64_t end);
7
8     private:
9         Node *head, *tail;
10
11         int randomLevel();
12         bool findInsert(uint64_t start, uint64_t end, Node **preds, Node **succs);
13         bool findExact(uint64_t start, uint64_t end, Node **preds, Node **succs);
14         void findDelete(uint64_t start, uint64_t end);
15 };
```

Listing 4.1: Pseudocode for ConcurrentRangeLock API

4.1.2 Find

Both tryLock and releaseLock methods rely heavily on find methods. There are several find methods in our implementation that serve different purposes:

- `bool findInsert(uint64_t start, uint64_t end, Node** preds, Node** succs)`: checks if the target range [start, end] is free to be inserted.
- `bool findExact(uint64_t start, uint64_t end, Node** preds, Node** succs)`: checks if the target range [start, end] is already present in the skip list.
- `void findDelete(uint64_t start, uint64_t end)`: finds the target range [start,

end] from the skip list to physically delete the Node which contains the corresponding range.

These `findInsert` and `findExact` methods also fill in the `preds[]` and `succs[]` arrays with the target node's predecessors and successors at each level. Because the goal of `findDelete` is only to snip out all the deleted Node, there is no need to fill any array.

Nevertheless, these methods have to maintain the following two properties:

- During traversal, they need to skip over marked nodes. They use `compareAndSet()` to ensure that they remove all softly deleted Node on the way.
- Every `preds[]` reference is to a node with a key strictly less than the target.

Algorithm in details

The `find()` method starts by traversing the `LockFreeSkipList` from the `topLevel` of the head sentinel, which has the maximal allowed node level. It proceeds down the list level by level, filling in the `preds` and `succs` nodes. These nodes are repeatedly advanced until `pred` refers to a node with the end value on that level that is strictly less than the target range `end` (lines 13–29).

While traversing, it repeatedly snips out marked nodes from the current level as they are encountered (lines 15–22) using a `compareAndSet()`. `compareAndSet()` function also validates that the next field of the predecessor still references the current Node.

Once an unmarked `curr` node is found (line 23), it is tested to see if its `start` is greater than or equal to the target `start`. If so, `pred` is advanced to `curr`, `curr` is advanced to `succ`, and the traverse continues. Otherwise, the current range of `pred` is the immediate predecessor of the target node. The `find()` method then breaks out of the current level search loop, saving the current values of `pred` and `curr` (lines 26–32).

The `find()` method continues this process until it reaches the bottom level. An important point is that each level's traversal maintains the previously described properties. Specifically, if a node with the target key is in the list, it will be found at the bottom level even if nodes are removed at higher levels. When traversal stops, `pred` refers to a predecessor of the target node. The method descends to each next lower level without skipping over the target node. If the Node is in the list, it will be found at the bottom level. Additionally, if the Node is found, it cannot be marked because if it were marked, it would have been snipped out in lines 15–22. Thus, the condition test on line 35 only

```

1 bool ConcurrentRangeLock::find(uint64_t start, uint64_t end,
2   Node **preds, Node **succs) {
3   bool marked[1] = {false};
4   bool snip;
5   Node *pred, *succ, *curr;
6
7   retry:
8   while (true) {
9     pred = head;
10    for (int level = maxLevel; level >= 0; level--) {
11      curr = pred->next[level]->getReference();
12
13      while (start > curr->getStart()) {
14        succ = curr->next[level]->get(marked);
15        while (marked[0]) {
16          snip = pred->next[level]->compareAndSet(curr, succ, false, false);
17
18          if (!snip) goto retry;
19
20          curr = pred->next[level]->getReference();
21          succ = curr->next[level]->get(marked);
22        }
23        if (start >= curr->getStart()) {
24          pred = curr;
25          curr = succ;
26        } else {
27          break;
28        }
29      }
30
31      preds[level] = pred;
32      succs[level] = curr;
33    }
34
35    return **condition**;
36  }
37 }

```

Listing 4.2: General pseudocode for find methods

needs to check if there are overlap ranges (findInsert) or if the start and end of the Node match the target start and end (findExact).

1. findInsert:

```
return !(start > pred->getEnd() && end < curr->getStart());
```

2. findExact:

```
return (start == curr->getStart() && end == curr->getEnd());
```


The linearization points for both successful and unsuccessful calls to the `find()` method occur when the `curr` reference at the bottom-level list is set, either at line 11 or line 20, for the last time before the success or failure of the `find()` call is determined at line 35.

4.1.3 Try Lock

The tryLock method, shown in Listing 4.3, utilizes findInsert() to check if a node with the range [start, end] already exists. If found, tryLock returns false. Otherwise, it creates a new node and attempts to insert it into the list. The Node is inserted starting from the bottom level, with compareAndSet() ensuring the integrity of the insertion. If any insertion fails due to concurrent changes, findInsert() is called again to update the predecessors and successors, and the process repeats until successful.

```
1  bool ConcurrentRangeLock::tryLock(uint64_t start, uint64_t end) {
2      int topLevel = randomLevel();
3      Node *preds[maxLevel + 1];
4      Node *succs[maxLevel + 1];
5
6      while (true) {
7          if (findInsert(start, end, preds, succs)) {
8              return false;
9          } else {
10             auto newNode = new Node();
11             newNode->initialize(start, end, topLevel);
12
13             for (int level = 0; level <= topLevel; ++level) {
14                 newNode->next[level]->store(succs[level], false);
15             }
16
17             if (!preds[0]->next[0]->compareAndSet(succs[0], newNode, false, false)) {
18                 continue;
19             }
20
21             for (int level = 1; level <= topLevel; ++level) {
22                 while (!preds[level]->next[level]->compareAndSet(
23                     succs[level], newNode, false, false)) {
24                     findInsert(start, end, preds, succs);
25                 }
26             }
27
28             return true;
29         }
30     }
31 }
```

Listing 4.3: Pseudocode for tryLock method

Algorithm in details

The tryLock method, shown in listing 4.3, uses findInsert(), show in listing 4.2, to determine whether a node with range [start, end] is already in the list (line 7). tryLock also calls findInsert() to initialize the preds[] and succs[] arrays to hold

the new Node's ostensible predecessors and successors. If an unmarked node with the target key is found in the bottom-level list, `findInsert()` returns accurate and the `tryLock` method returns false, indicating that the key is already in the set. The unsuccessful `tryLock`'s linearization point is the same as the successful `findInsert()`'s (line 8). If no node is found, the next step is to add a new node with the key into the structure.

A new node is created with a randomly chosen top-level (lines 10–11). The Node's next references are unmarked and set to the successors returned by the `findInsert()` method (lines 13–15). The next step is to try to add the new Node by linking it into the bottom-level list between the `preds[0]` and `succs[0]` nodes returned by `findInsert()`. We use the `compareAndSet()` method to set the reference while validating that these nodes still refer one to the other and have not been removed from the list (line 17). If the `compareAndSet()` fails, something has changed and the call restarts. If the `compareAndSet()` succeeds, the item is added, and line 17 is the call's linearization point. The `findInsert()` then links the Node in at higher levels (lines 21–25). For each level, it attempts to splice the Node by setting the predecessor, if it refers to the valid successor, to the new Node (lines 22–23). If successful, it breaks and moves on to the next level. If unsuccessful, then the Node referenced by the predecessor must have changed, and `findInsert()` is called again to find a new valid set of predecessors and successors (line 24). We discard the result of calling `findInsert()` because we care only about recomputing the ostensible predecessors and successors on the remaining unlinked levels. The method returns true once all levels are linked (line 27). The `releaseLock` method, shown in listing 4.4, calls `findExact()` to determine whether an unmarked node with a matching range `[start, end]` is in the bottom-level list (line 7). If no node is found in the bottom-level list, or the Node with a matching range `[start, end]` is marked, the method returns false. The linearization point of the unsuccessful `releaseLock` is that of the `findExact()` method called in line 7.

If an unmarked node is found, the method logically removes the associated key from the abstract set and prepares it for physical removal. This step uses the set of ostensible predecessors (stored by `findExact()` in `preds[]`) and the victim (returned from `findExact()` in `succs[]`). First, starting from the top-level, all links up to and **not including** the bottom-level link are marked (lines 12–20) by repeatedly reading `next` and its mark and applying `attemptMark()`. If the link is found to be marked (either because it was already marked or because the attempt succeeded), the method moves on to the next-level link. Otherwise, the current level's link is reread since another concurrent thread must have changed it, so the marking attempt must be repeated.

Once all levels but the bottom one have been marked, the method marks the bottom

level's next reference. If successful, this marking (line 27) is the linearization point of a successful `releaseLock`. The `releaseLock` method tries to mark the next field using `compareAndSet()`. If successful, it can determine that it was the thread that changed the mark from false to true. Before returning true, the `findDelete()` method is called. This call is an optimization: `findDelete()` physically removes all links to the Node it is searching for.

On the other hand, if the `compareAndSet()` call fails, but the next reference is marked, then another thread must have concurrently removed it, so `releaseLock` returns false. The linearization point of this unsuccessful `releaseLock` is the linearization point of the `releaseLock` method by the thread that successfully marked the next field. Notice that this linearization point must occur during the `releaseLock` call because the `findExact()` call found the Node unmarked before it found it marked.

Finally, if the `compareAndSet()` fails and the Node is unmarked, the next Node must have changed concurrently. Since the victim is known, there is no need to call `find()` again, and `releaseLock` simply uses the new value read from `next` to retry the marking.

4.1.4 Release Lock

The `releaseLock` method first calls `findExact()` to locate an unmarked node with the specified range `[start, end]`. If found, it marks all node levels except the bottom one, preparing the Node for removal. The method then attempts to mark the bottom-level link using `compareAndSet()`, which serves as the linearization point for a successful `releaseLock`. If marking fails due to concurrent modifications, the method retries or returns false, depending on the Node's state.

```

1  bool ConcurrentRangeLock::releaseLock(uint64_t start, uint64_t end) {
2      Node *preds[maxLevel + 1];
3      Node *succs[maxLevel + 1];
4      Node *succ;
5
6      while (true) {
7          bool found = findExact(start, end, preds, succs);
8          if (!found) {
9              return false;
10         } else {
11             Node *nodeToRemove = succs[0];
12             for (int level = nodeToRemove->getTopLevel();
13                  level >= 0 + 1; level--) {
14                 bool marked[1] = {false};
15                 succ = nodeToRemove->next[level]->get(marked);
16                 while (!marked[0]) {
17                     nodeToRemove->next[level]->attemptMark(succ, true);
18                     succ = nodeToRemove->next[level]->get(marked);
19                 }
20             }
21
22             bool marked[1] = {false};
23             succ = nodeToRemove->next[0]->get(marked);
24             while (true) {
25                 bool iMarkedIt = nodeToRemove->next[0]->compareAndSet(
26                     succ, succ, false, true);
27                 succ = succs[0]->next[0]->get(marked);
28                 if (iMarkedIt) {
29                     findDelete(start, end);
30
31                     return true;
32                 } else if (marked[0]) {
33                     return false;
34                 }
35             }
36         }
37     }
38 }

```

Listing 4.4: Pseudocode for `releaseLock` method

Algorithm in details

The `releaseLock` method, shown in listing 4.4, calls `findExact()` to determine whether an unmarked node with a matching range `[start, end]` is in the bottom-level list (line 7). If no node is found in the bottom-level list, or the Node with a matching range `[start, end]` is marked, the method returns false. The linearization point of the unsuccessful `releaseLock` is that of the `findExact()` method called in line 7.

If an unmarked node is found, the method logically removes the associated key from the abstract set and prepares it for physical removal. This step uses the set of ostensible predecessors (stored by `findExact()` in `preds[]`) and the victim (returned from `findExact()` in `succs[]`). First, starting from the top-level, all links up to and **not including** the bottom-level link are marked (lines 12–20) by repeatedly reading `next` and its mark and applying `attemptMark()`. If the link is found to be marked (either because it was already marked or because the attempt succeeded), the method moves on to the next-level link. Otherwise, the current level's link is reread since another concurrent thread must have changed it, so the marking attempt must be repeated.

Once all levels but the bottom one have been marked, the method marks the bottom level's next reference. If successful, this marking (line 27) is the linearization point of a successful `releaseLock`. The `releaseLock` method tries to mark the next field using `compareAndSet()`. If successful, it can determine that it was the thread that changed the mark from false to true. Before returning true, the `findDelete()` method is called. This call is an optimization: `findDelete()` physically removes all links to the Node it is searching for.

On the other hand, if the `compareAndSet()` call fails, but the next reference is marked, then another thread must have concurrently removed it, so `releaseLock` returns false. The linearization point of this unsuccessful `releaseLock` is the linearization point of the `releaseLock` method by the thread that successfully marked the next field. Notice that this linearization point must occur during the `releaseLock` call because the `findExact()` call found the Node unmarked before it found it marked.

Finally, if the `compareAndSet()` fails and the Node is unmarked, the next Node must have changed concurrently. Since the victim is known, there is no need to call `find()` again, and `releaseLock` simply uses the new value read from `next` to retry the marking.

5 Evaluation

This section will evaluate our proposed concurrent range lock under different scenarios. The goal is to see the scalability and throughput of our mechanism compared to state-of-the-art range locks.

5.1 Competitor

We denote our primary implementation as *Our*. In addition to *Our*, we implement two different variants for comparison. The first variant is a scalable range lock proposed by Kogan et al. [2], denoted as *Scalable RL*. We specifically implemented the Exclusive Access Variant presented in their paper, as it aligns with the focus of our research. The second variant is a skip list range lock proposed by Song et al. [3], which we denote as *Song RL*.

5.2 Benchmark environment

For benchmark purposes, we utilized a server with an AMD Ryzen 9 7950X processor, which features 16 cores and 32 threads, providing substantial computational power for the experiments. The server configuration included a virtual memory cache with a capacity of 128 GB, backed by 32 GB of physical memory. The cache management aimed to minimize eviction sizes to enhance performance and reliability during the benchmarks.

5.3 Microbenchmark

5.3.1 Workload

The primary objective of a concurrent range lock mechanism is to enable multiple threads to access disjoint parts of the same shared object efficiently. To simulate and evaluate the effectiveness of different range locking strategies, we utilize the `mmap()`

system call to create a shared object in memory and the `memset()` function to simulate write operations to this shared object. For each range, we will write 1KB. The use of `memset()` serves as a placeholder for actual modifications, enabling us to focus on the performance characteristics of the locking mechanism itself.

To explore various levels of contention and potential usage scenarios for range locks, we have devised two distinct workloads, each designed to stress the locking mechanism under different conditions:

- **W1:** In this workload, each thread operates with fine granularity. A thread locks a single memory range, performs a modification (simulated by `memset()`), and immediately releases the lock before proceeding to the next range in its queue. This approach simulates a scenario with minimal contention, focusing on the efficiency of the lock mechanism in handling rapid lock acquisition and release cycles. The primary objective is to assess the overhead introduced by the locking mechanism under low contention and to evaluate its performance in scenarios demanding high throughput with minimal waiting times.
- **W2:** This workload introduces a more complex and realistic scenario. Here, threads perform batched memory operations, where a series of memory ranges (typically 16) are locked, modified (simulated by `memset()`) and then unlocked in a single batch. The goal is to test all variants under heavier threaded conditions. By locking and unlocking in batches, the number of ranges within the data structure increases, providing insight into how efficiently each variant can search, lock, and unlock ranges as the load increases. This workload emphasizes the performance of the locking mechanism in handling more substantial, real-world use cases where contention may be higher.

Through these workloads, we aim to comprehensively evaluate the performance of the concurrent range lock mechanism under different workloads and contention scenarios. Each workload provides insights into the lock's scalability, efficiency, and overall robustness in handling varying degrees of parallelism and contention.

5.3.2 Optimal Height for Range Locking

The skip list height in our range-locking mechanism plays a critical role in balancing performance and resource utilization. A skip list with insufficient height may need to optimize lookup times effectively, leading to slower performance. In contrast, an excessive height increases memory consumption and management overhead without proportionate gains in efficiency. This test aims to determine the optimal height for our concurrent range-locking mechanism before comparing it to alternative approaches.

We conducted experiments using the two workloads described in 5.3.1 to identify the best skip list height.

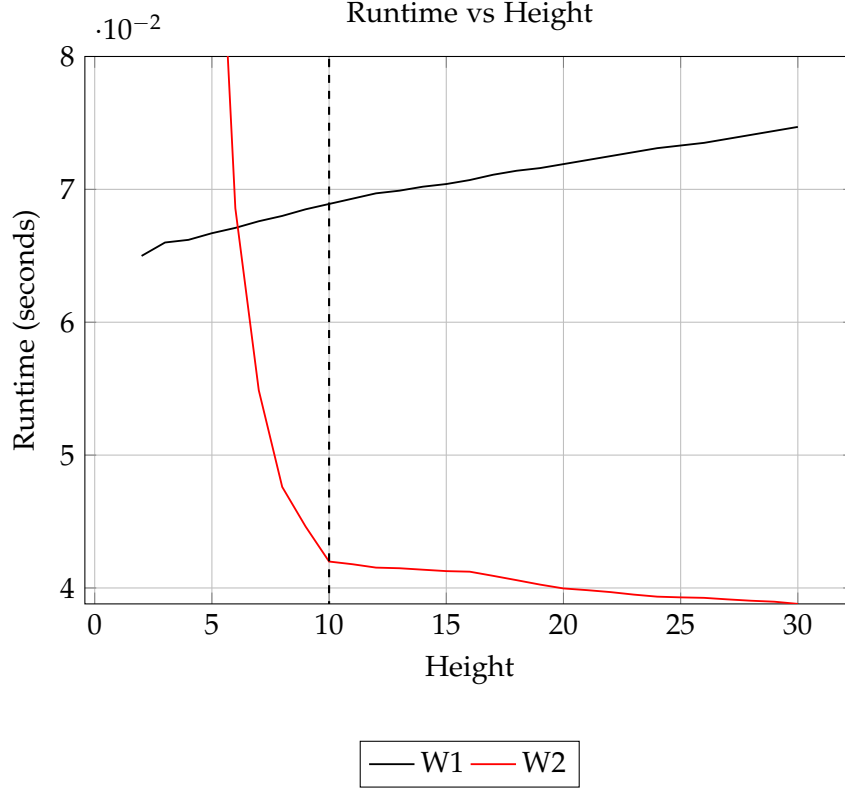


Figure 5.1: Runtime for different heights of range lock

For **W1**, where contention is minimal, and each thread locks and releases ranges immediately, the height of the skip list has negligible impact on performance. In this scenario, since the number of ranges locked at any given time is limited to the number of active worker threads, the benefits of a taller skip list are not realized. The overhead of managing additional levels outweighs any potential gains, leading to similar performance across different heights.

For **W2**, the total runtime decreases as the skip list height increases, aligning with the expected behavior. The multi-level structure of skip lists allows for faster lookups, reducing runtime as height increases. However, beyond a height of 10, further increases yield diminishing returns. This is due to the probabilistic level assignment in skip lists, where the chance of a node reaching higher levels decreases exponentially (e.g., level 12 has a probability of about $\frac{1}{2^{12}} \approx 0.00024\%$). The sparsely populated upper levels

contribute little to performance improvement but increase memory usage.

Based on these observations, we identify a sweet spot at a height of 10, where the trade-off between performance and resource utilization is optimized. This height balances the need for efficient lookups with manageable memory overhead, making it the ideal choice for our range-locking mechanism in subsequent benchmarks.

5.3.3 Result

Figure 5.2 shows the number of successful locks and unlocks per second with increasing working threads under **W1**. We can see that Our variant has good scalability and outperforms the other two. We archive four times more operations than Scalable RL and twelve times more than Song RL. The poor performance of Song RL is due to the immediate locking and releasing mechanism, combined with the low memory overhead of `memset` (1KB per range). The spinlock in Song RL becomes a significant point of contention, leading to its poor performance. Additionally, since the locks are released immediately, Song RL cannot effectively leverage its skip list data structure, as the maximum number of ranges in the list is limited to the number of worker threads. However, in more realistic scenarios, such as those discussed in 5.4 or **W2**, we will see Song RL perform much better.

Figure 5.3 illustrates the same result for **W2**, Our variant continued to outperform the others, achieving two to nine times more operations than Scalable RL and two to six times more than Song RL. In this workload, the number of ranges locked simultaneously increases to the number of threads multiplied by the batch size (16). This allowed Song RL to perform significantly better compared to Scalable RL. However, Scalable RL demonstrated better scalability as the number of worker threads increased. Beyond sixteen threads, Scalable RL began to surpass Song RL in performance.

In summary, Our variant consistently outperforms both Scalable RL and Song RL across different workloads. While Song RL struggles with contention in **W1**, it performs better in more complex scenarios like **W2**. However, Scalable RL demonstrates better scalability with increasing thread counts, eventually surpassing Song RL as threads increase.

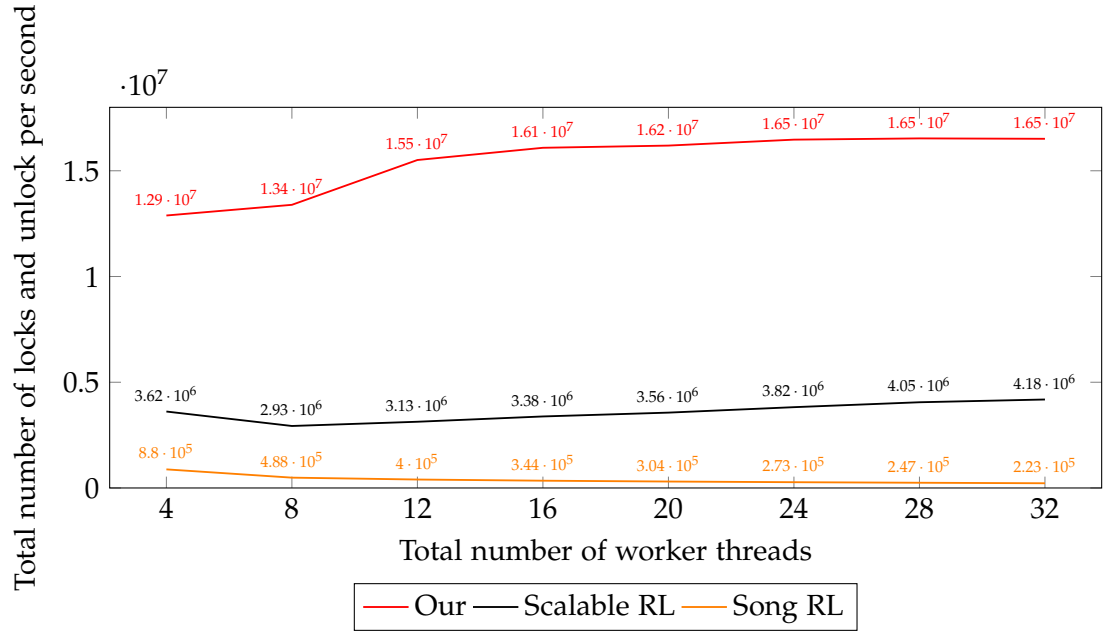


Figure 5.2: Microbenchmark for workload W1

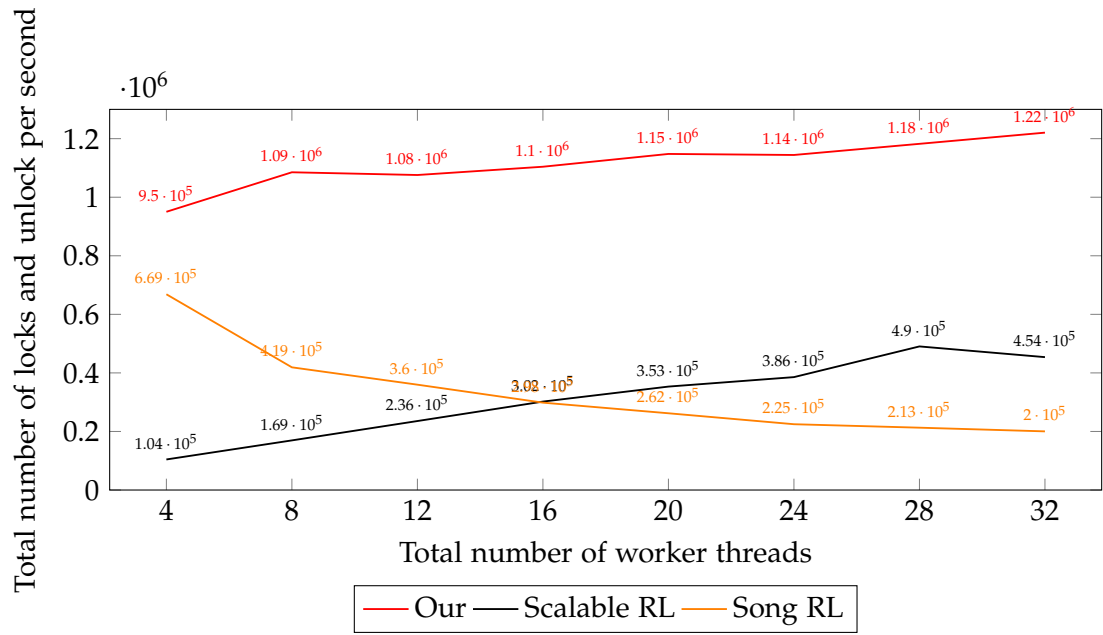


Figure 5.3: Microbenchmark for workload W2

5.4 Leanstore

Leanstore [13] is a high-performance storage engine designed to support various database management systems. To further enhance its capabilities, Nguyen et al. [10] introduced a comprehensive design for allocating and logging large objects, which has been integrated into Leanstore. Their performance study demonstrates that this approach not only outperforms many popular file systems but also ensures transactional consistency and durability for large objects. Given the crucial role that range locks play in their design, we integrated our concurrent range lock mechanism to enable realistic and rigorous benchmarking.

In the proposed design, range locks synchronize access to shared aliasing areas, which are contiguous ranges of virtual memory addresses used to present disjointed extents as contiguous memory. When a worker needs to allocate virtual memory for large BLOBs, particularly when these BLOBs exceed the size of the worker-local aliasing area, it must reserve free virtual memory from the shared aliasing area. A range lock mechanism prevents concurrent workers from accessing overlapping memory regions. The range lock operates by locking specific ranges within the shared aliasing area, ensuring that only one worker can modify or access a particular memory range. This prevents race conditions and ensures data consistency while simultaneously allowing multiple workers to operate on different memory ranges.

5.4.1 Competitors

We integrated all three versions as described in section 5.3. Additionally, we included the original, specifically optimized range lock version by the author as a fourth competitor. This range lock employs a bitmap and compare-and-swap, making it lightweight and efficient, especially for managing small to moderate numbers of logical blocks in memory.

5.4.2 Workload

The experiment, utilizing synthetic YCSB [14] workloads, was designed to evaluate the performance of LeanStore under a read-intensive, multithreaded environment. The workload comprises exclusively read operations, executed over a 10-second duration, with each read operation employing a straightforward `memcpy()` function. Each record in the workload has a payload size of 1 MB, with 1000 records being processed. The experiment also incorporates a buffer management configuration that allocates 128 GB

of virtual and 32 GB of physical memory, providing a robust test of LeanStore’s ability to manage large data sets under constrained physical memory conditions.

5.4.3 Result

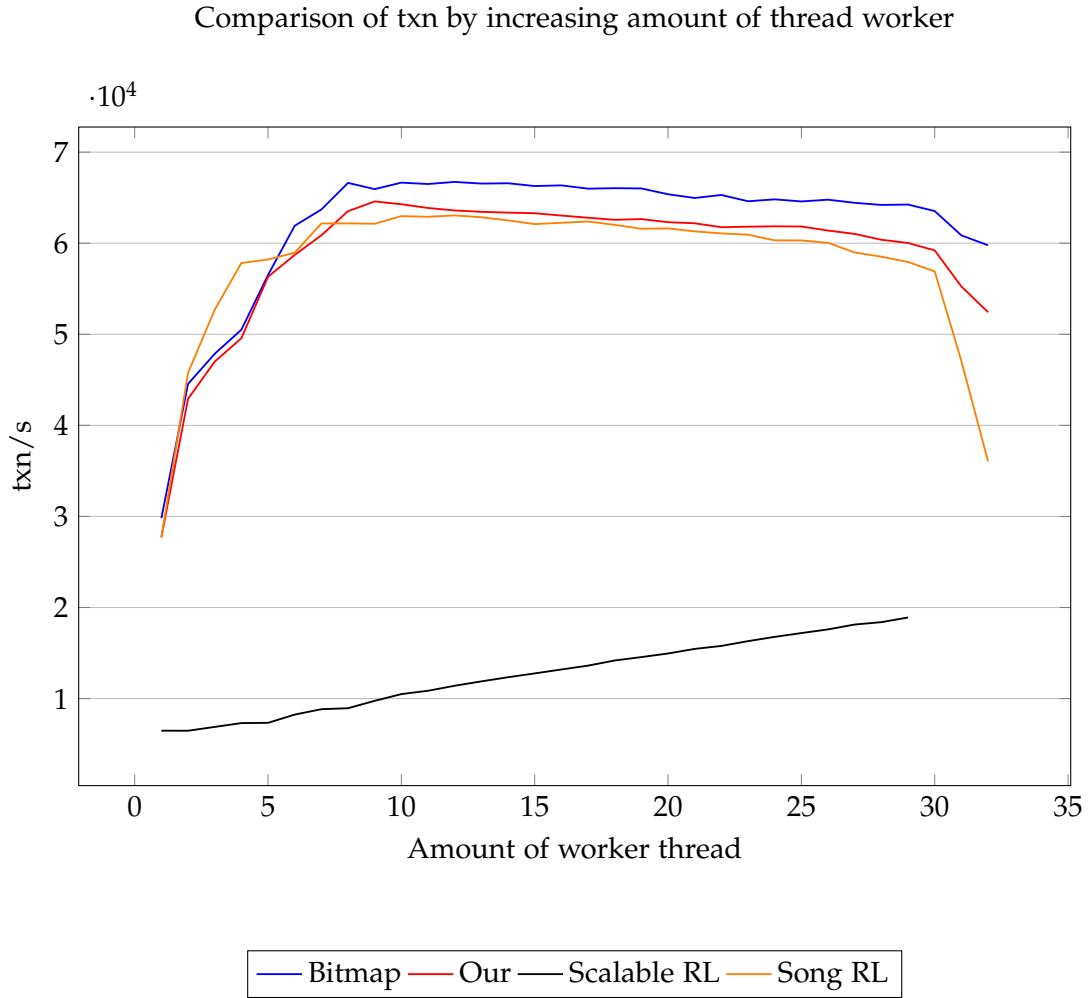


Figure 5.4: Comparison of transactions by increasing the number of thread workers.

We evaluated the throughput of four distinct variants as the number of worker threads increased, as depicted in Figure 5.4. As anticipated, the `Bitmap` implementation consistently outperformed the other three variants. This superiority can be attributed mainly to its optimized use of the aliasing arena, making it the most efficient workload

handling. Our implementation (Our) demonstrated competitive performance, closely trailing Bitmap in terms of transactions per second. Both Bitmap and Our showed effective scalability with an increasing number of worker threads, though Bitmap maintained a consistent, albeit slight, performance advantage.

Conversely, Song RL initially exhibited substantial throughput but encountered a sharp decline beyond 24 worker threads. This decline underscores the limitations of its coarse-grained range lock mechanism, as discussed in 2.1.2. The single spinlock in Song RL becomes a significant bottleneck in high-contention scenarios, leading to marked performance degradation. On the other hand, the Scalable RL variant showed steady, linear throughput growth. However, its overall performance remained considerably lower than the other variants, highlighting its limitations.

An additional key observation is the difficulty faced by the Bitmap, Our, and Song RL variants in scaling effectively beyond 16 worker threads. This limitation is due mainly to the BLOB size in our workload. With a BLOB size of 1MB and 16 worker threads, the combined size of the client-side buffer and the internal DBMS memory block for the BLOB exceeds the L3 cache capacity (32MB in our machine), resulting in significant contention at the L3 cache. Furthermore, LeanStore’s use of `memcpy` for read operations exacerbates this issue. Each 1MB BLOB consumes 2MB of memory due to `memcpy` during read and write operations. An average throughput of approximately 60,000 operations per second equals over 110 GB of memory consumed by `memcpy`, which saturates the memory bandwidth and hinders the application’s ability to scale.

We also analyzed other performance metrics, such as cycles, instructions, and L1 cache misses. As illustrated in Figure 5.5, both the Bitmap and Our implementations maintained stable levels of cycles, instructions, and L1 cache misses as the number of threads increased. In contrast, Song RL exhibited a significant rise in the number of instructions with additional threads, indicating inefficiencies under higher loads. Meanwhile, Scalable RL demonstrated better scalability, with a decreasing number of L1 misses and instructions as the workload intensified.

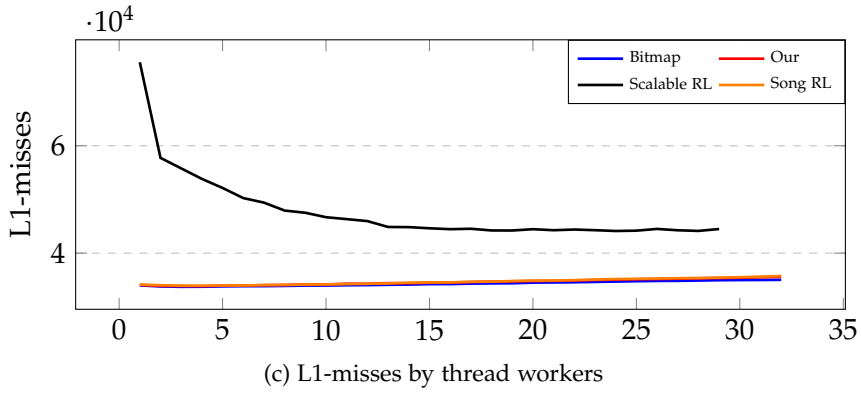
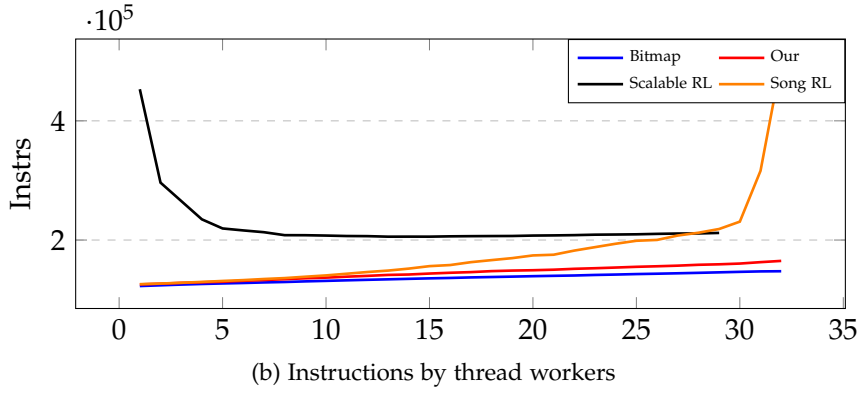
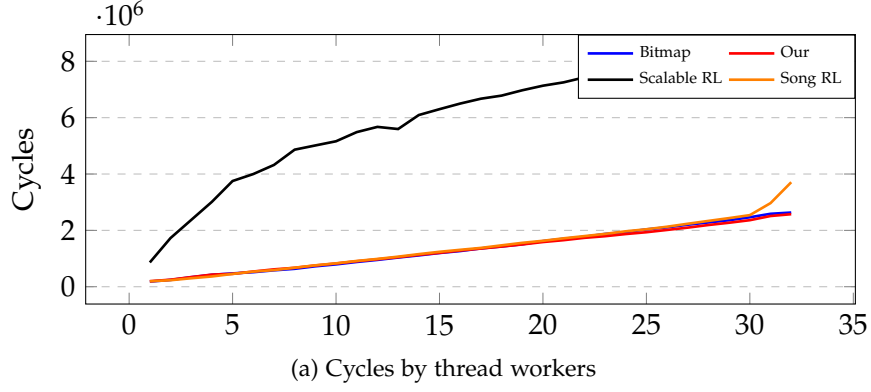


Figure 5.5: Comparison of cycles, instructions, and L1-misses by increasing the number of thread workers.

6 Conclusion

In this thesis, we presented a novel lock-free concurrent range-locking mechanism designed to address the bottlenecks and inefficiencies observed in existing range-locking approaches. Through developing and rigorously evaluating our proposed method, we demonstrated its superiority in various high-concurrency environments, particularly in large-scale distributed systems where efficient access control is paramount.

Our research began by identifying the limitations of traditional range-locking techniques, such as the bottlenecks introduced by coarse-grained locking and the inefficiencies in tree and list-based range locks. While effective in some scenarios, these methods often fall short in high-performance computing environments due to their susceptibility to contention and their reliance on locking mechanisms that do not scale well with increasing concurrency. To overcome these challenges, we introduced a concurrent range lock based on a probabilistic skip list structure, which leverages atomic operations such as `compareAndSet` to manage node references without the need for locks. This design eliminates the bottlenecks associated with traditional locking mechanisms and maintains robust concurrency performance even under heavy load. Our extensive evaluation, including microbenchmarks and integration into a high-performance storage engine like Leanstore, highlighted the efficiency and scalability of our approach. The results showed that our concurrent range lock outperformed existing methods, including scalable range locks and skip-list-based locks, across various workloads and thread counts. Notably, our mechanism achieved significant improvements in throughput and reduced contention, making it well-suited for modern distributed systems requiring high concurrency levels.

This thesis presents an improvement in managing multiple users or processes accessing data simultaneously, providing a scalable and effective method for dealing with environments where many tasks are happening simultaneously. Our research sets the stage for further studies on systems and structures that do not require locking mechanisms for access control. We believe our developed approaches will be widely helpful in scholarly studies and real-world applications.

7 Future work

While our proposed mechanism shows significant improvements, there are still areas that need further exploration and optimization. One key observation from our benchmarking is that our implementation, referred to as `Our`, does not scale as well as `Scalable RL` when it comes to handling workloads that extend beyond L3 cache lines. This scalability issue highlights an area where we can enhance performance, especially in scenarios where cache efficiency is crucial. Addressing this will be essential to ensure that our approach can scale effectively in high-performance environments.

Another important area that requires attention is memory deallocation. Our current design lacks an efficient strategy for managing memory deallocation, which could lead to memory leaks or unnecessary memory usage over time. Developing a robust solution for this issue is crucial to improving the overall efficiency and reliability of our range-locking mechanism.

We also plan to conduct a detailed analysis of memory overhead. Since our implementation uses additional layers of linked lists to maintain the skip list and includes abstractions like `AtomicMarkableReference`, there is a memory overhead associated with these structures. By analyzing this overhead and comparing it with other variants, we can gain insights into the trade-offs between memory usage and performance, helping us refine our approach further.

Additionally, we are interested in exploring how our concurrent range lock can be integrated with other data structures or applied in different contexts. This could provide further insights and improvements. For example, we have identified a version of range lock used in `PerconaFT` [15], where a lock tree is implemented. Investigating how our design could benefit from or be adapted to include such techniques could lead to new opportunities for improving the flexibility and applicability of our approach.

In summary, while our current design offers significant advancements, addressing these areas of optimization, memory management, and integration will be key to advancing the state of concurrent range-locking mechanisms. These future efforts will not only enhance the performance and efficiency of our mechanism but also expand its usefulness across various high-concurrency environments.

Bibliography

- [1] J. Gao, Y. Lu, M. Xie, Q. Wang, and J. Shu. “Citron: Distributed Range Lock Management with One-sided {RDMA}”. In: *21st USENIX Conference on File and Storage Technologies (FAST 23)*. 2023, pp. 297–314.
- [2] A. Kogan, D. Dice, and S. Issa. “Scalable range locks for scalable address spaces and beyond”. In: *Proceedings of the Fifteenth European Conference on Computer Systems*. 2020, pp. 1–15.
- [3] X. Song, J. Shi, R. Liu, J. Yang, and H. Chen. “Parallelizing live migration of virtual machines”. In: *Proceedings of the 9th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. 2013, pp. 85–96.
- [4] J. Kara. “Implement range locks”. In: *lkml.org* (2013). Accessed: 2024-04-21. URL: <https://lkml.org/lkml/2013/1/31/483>.
- [5] G. Graefe. “Hierarchical locking in B-tree indexes”. In: *On Transactional Concurrency Control*. Springer, 2007, pp. 45–73.
- [6] A. Pavlo. “Two-Phase Locking”. In: *15445.courses.cs.cmu.edu* (2022). Accessed: 2024-04-21. URL: <https://15445.courses.cs.cmu.edu/fall2022/slides/16-twophaselocking.pdf>.
- [7] J.-H. Kim, J. Kim, H. Kang, C.-G. Lee, S. Park, and Y. Kim. “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*. 2019, pp. 1–7.
- [8] W. Pugh. *A skip list cookbook*. Citeseer, 1990.
- [9] M. Fomitchev and E. Ruppert. “Lock-free linked lists and skip lists”. In: *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*. 2004, pp. 50–59.
- [10] L.-D. Nguyen and V. Leis. “Why Files If You Have a DBMS”. In: *ICDE*. 2024.
- [11] W. Pugh. “Skip lists: a probabilistic alternative to balanced trees”. In: *Communications of the ACM* 33.6 (1990), pp. 668–676.
- [12] H. Maurice, S. Nir, L. Victor, and S. Michael. *The art of multiprocessor programming*. Newnes, 2020.

- [13] V. Leis, M. Haubenschild, A. Kemper, and T. Neumann. “LeanStore: In-memory data management beyond main memory”. In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE. 2018, pp. 185–196.
- [14] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. “Benchmarking cloud serving systems with YCSB”. In: *Proceedings of the 1st ACM symposium on Cloud computing*. 2010, pp. 143–154.
- [15] *PerconaFT*. <https://github.com/percona/PerconaFT>. Accessed: 2024-08-27. 2024.