CHAIR OF DECENTRALIZED
INFORMATION SYSTEMS & DATA
MANAGEMENT

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Concurrent Range Locking

Thua-Duc Nguyen

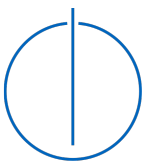
CHAIR OF DECENTRALIZED INFORMATION SYSTEMS & DATA MANAGEMENT

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, disaggregated 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 and removing the lock while maintaining high 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

List of Figures

List of Tables

1 Introduction

Locking mechanism is important. In modern computing environments, the efficiency of locking mechanisms plays a crucial role in the performance of various systems, including databases [1, 2], file systems [3, 4, 5], and operating systems [6, 7]. As these systems grow in scale and complexity, the demand for more sophisticated and efficient locking mechanisms becomes crucial. One of the fundamental challenges in this context is managing concurrent access to shared resources. Traditional locking techniques, such as single-lock mechanisms, often lead to significant performance bottlenecks, particularly in high-concurrency scenarios.

Range locks boot performance through resource segmentation. Range lock [4, 8] provides a more refined approach to this issue by partitioning a shared resource into multiple arbitrary-sized segments. Different processes can exclusively acquire each of these segments. This strategy effectively addresses the drawbacks and bottlenecks associated with single-lock methods, significantly improving the performance.

DBMS needs range locks. As database sizes increase exponentially, locking the entire database becomes impractical. This approach will prevent concurrent transactions from progressing, resulting in poor throughput and high latency. The previous key-range locking in DBMS is complex and tightly coupled with lock-based concurrency control protocols [2, 9]. Consequently, this technique does not apply to general DBMS operations like variable-sized page allocation. Therefore, a new technique, such as range locks, is desirable.

Filesystem needs range locks. In high-performance file systems, particularly those used in large-scale and distributed computing environments, efficiently managing concurrent access to shared files is crucial. As file systems scale to accommodate massive data sets and numerous parallel I/O operations, traditional locking mechanisms often become bottlenecks, reducing throughput and increasing latency. Range locks offer a solution by allowing multiple processes to access different file segments simultaneously without interfering with each other. This segmentation minimizes contention and improves performance by enabling finer-grained locking at the segment level. For file systems handling concurrent access to large files, especially in high-performance computing

(HPC) environments, adopting range locks can significantly enhance efficiency and scalability [4, 3].

Operating system needs range locks. There has been growing interest in using range locks within the Linux kernel community. The focus is on using range locks to alleviate contention issues associated with `mmap_sem`, a semaphore that manages access to virtual memory areas (VMAs). VMAs represent distinct sections of an application's virtual address space and are organized in a red-black tree. The `mmap_sem` semaphore controls access for various operations such as memory mapping, unmapping, protection, and handling page faults. This becomes problematic for data-intensive applications with large, dynamically allocated memory, as contention on this semaphore can become a significant performance bottleneck.

Existing range lock needs improvement. 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 [10, 11]. Additionally, some methods may be complex and tightly coupled with lock-based concurrency control protocols, which are not applicable for general DBMS operations [2, 9]. These limitations underscore the need for more refined and scalable solutions that can better handle the demands of modern, large-scale systems.

New concurrent range-locking design. In this research's scope, we propose a new lock-free concurrent range-locking design. We address previous bottleneck issues by leveraging a probabilistic concurrent skip list [12, 13] and replace traditional locking by `compare-and-swap` methods. By doing so, we address the previous range lock bottleneck issues and maintain the lock's high level of performance.

Outline of the research. The scope of this research includes developing and evaluating the proposed range-locking mechanism. We will evaluate focusing on performance under heavy concurrent accesses, ensuring the correctness of data access in overlapping ranges and concurrent operations. Additionally, we will compare the performance of the proposed solution with existing state-of-the-art approaches to provide a comprehensive assessment of its effectiveness.

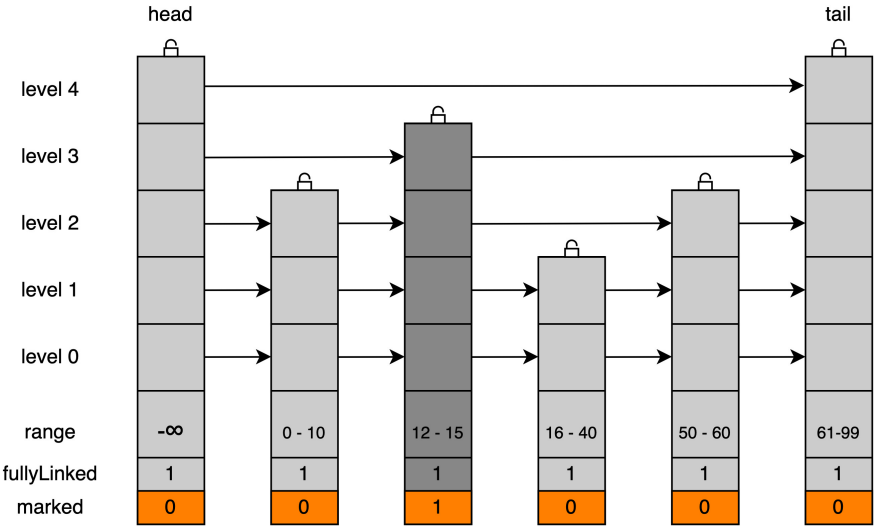


Figure 1.1: Concurrent range lock

2 Related work

This chapter reviews the existing research on range locks, focusing on different aspects of the problem and the solutions researchers have proposed. The following sections provide a detailed examination of different implementations, highlighting their limitations. This analysis aims to show the evolution of range-locking mechanisms and the ongoing efforts to improve their performance and scalability.

2.1 Range Lock in the Linux Kernel

Jan Kara introduced a range-locking mechanism for the Linux kernel [10]. The implementation leverages an interval tree to manage range locks and employs a spin lock for synchronization. Each lock is represented as a node in the tree.

How it works

When a thread wants to acquire a range, it first acquires a spinlock. Then, it traverses the tree to determine the number of locks intersecting with the given range and find the. Next, the thread inserts a node describing its range into the tree and releases the spinlock. If the number of intersecting locks is zero, the thread can access and write to the critical section it acquires. Otherwise, it waits until those locks are released when the number of intersecting locks drops to zero. When a thread completes its range, it acquires the spin lock, removes its node from the tree, updates the number of overlapping locks, and frees the spinlock. It guarantees that each range is locked only after all previous conflicting range locks requested have been unlocked, thus achieving fairness and avoiding livelocks.

Drawbacks

Single Point of Contention. On this range lock, all operations relies on a single spinlock to protect the entire tree. Under heavy concurrent access, this easily becomes a contention point, limiting the system's performance.

Limitation of FIFO Order. 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 the FIFO restriction. This unnecessary blocking reduces the overall efficiency and concurrency of the system.

2.2 Skip List-Based Range Lock

Song et al. [11] introduced a dynamic and fine-grained range-locking design to enhance the implementation of the Linux kernel. Their range lock utilizes a skip list [14] to dynamically manage the address ranges that are currently locked.

How it works

When a thread requests a specific range `[start, start+len)`, the range lock searches for it in the skip list. If an existing or overlapping range is found in the skip list, it means that another thread is currently modifying the specific range, and the requesting thread must wait and then retry. If no overlapping range is found, the range is added to the skip list, indicating that the range 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, and it can efficiently perform searches for overlapping ranges.

Drawbacks

Single Contention Point: Similar to the one found in the Linux kernel, this range lock is also protected by a spinlock. Hence, the same bottleneck issue still need to be addressed

2.3 Concurrent Linked List-Base Range Lock

Kogan et al. [8] introduced a novel 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 some 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.

How it works

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` operation. A significant difference in this method compared to the previous one is that a 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 reaches a marked node, it simply removes it using CAS and continues to iterate. If the current node is protecting a range that overlaps, it simply 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.

Drawbacks

Linked List Inefficiency. This design implements a lock-free mechanism that effectively addresses the limitations of existing range locks. However, this approach comes with its own set of trade-offs. In general, insertion and lookup operations in a linked list are less efficient than 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 [15].

2.4 Comparative Analysis and Trade-offs

The analysis of range lock implementations reveals distinct trade-offs in terms of performance, scalability, and complexity. The interval tree-based range lock in the Linux kernel, despite its fairness and simplicity, suffers from a significant bottleneck due to its reliance on a single spinlock, which can severely limit performance under high contention. The skip list-based range lock offers improved efficiency and dynamic management of address ranges, leveraging the lightweight and fast nature of skip lists. However, it still retains the single contention point, similar to the interval tree approach, which hinders its scalability under heavy concurrent access. The concurrent linked list-based range lock by Kogan et al. presents a lock-free mechanism that significantly enhances scalability by eliminating blocking behavior. Nonetheless, it introduces

inefficiencies in insertion and lookup operations, as linked lists inherently have a higher average time complexity compared to skip lists and tree-like structures.

2.5 Future Directions

Future research on range-locking mechanisms should focus on hybrid approaches that combine the advantages of different data structures to mitigate their individual weaknesses. For instance, exploring the integration of lock-free principles with more efficient data structures, such as combining skip lists with lock-free operations, could enhance both performance and scalability. Additionally, investigating adaptive range-locking mechanisms that dynamically switch between different data structures based on the current contention level and workload characteristics could provide optimized performance across varying conditions. Finally, further studies on distributed range-locking techniques could extend these mechanisms to multi-node environments, addressing the growing need for scalable synchronization in distributed systems. By advancing these directions, researchers can develop more robust and efficient range-locking solutions that cater to the demands of modern computing environments.

3 Approach

In this research's scope, we propose our new concurrent range-locking design that leverages a probabilistic concurrent skip list [12, 13]. It consists of two main functions:

- **tryLock**: The `tryLock` function searches for the required range `[start,end]` in the skip list. If an overlapping range exists, indicating another thread is modifying that range, the requesting thread must wait and retry. If not, the range is added to the list, signaling that the range is reserved.
- **releaseLock**: The `releaseLock` function releases the lock by finding the address range in the skip list and removing it accordingly.

Our range lock design is also lock-free. It relies on atomic operations (`compareAndSet()`), thus addressing the bottleneck problem of the spinlock-based range lock and maintaining the lock's high level of performance.

3.1 Skip List

A skip list is a probabilistic data structure. It allows fast search, insertion, and deletion. It is an alternative to balanced trees, such as AVL trees or red-black trees [14, 16]. The key idea of a skip list is to use multiple layers of sorted linked lists to maintain elements, where each layer is an ”express lane” for faster traversal.

How it work

Multiple Layers: A skip list consists of multiple layers where the bottom-most layer is a regular sorted linked-list. Each higher layer acts as an ”express lane” that speeds up access by skipping over multiple elements from the layer below. Nodes in higher layers provide shortcuts, allowing faster traversal across the list and effectively reducing the time complexity of search operations.

Probabilistic Balancing: Each `Node` is created with a random top level and belongs to all lists up to that level. Top levels are chosen so that the expected number of nodes in

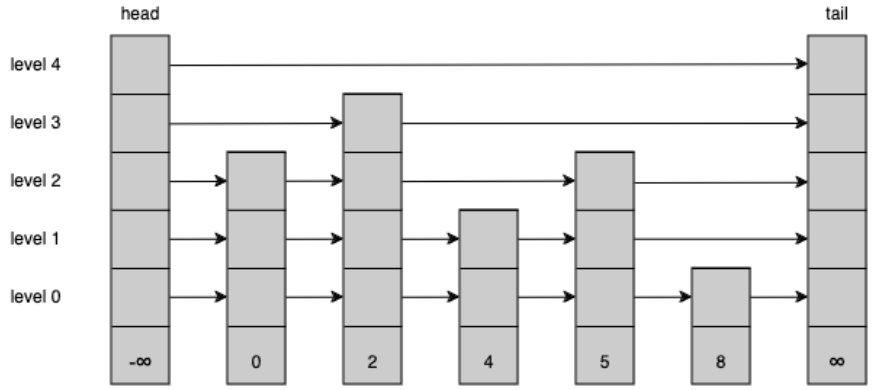


Figure 3.1: Skip List: this example has five levels of sorted linked lists.
Each `Node` has a unique key, and the head and tail have $-\infty$ and $+\infty$ keys.

each level's list decreases exponentially. Let $0 < p < 1$ be the conditional probability that a `Node` at level i also appears at level $i + 1$. All nodes appear at level 0. The probability that a `Node` at level 0 also appears at level $i > 0$ is p^i. For example, with $p = 1/2$, $1/2$ of the nodes are expected to appear at level 1, $1/4$ at level 2, and so on, providing a balancing property like the classical sequential tree-based search structures, except without the need for complex global restructuring. This random generation ensures that the list structure remains balanced. Consequently, skip list insertion and deletion algorithms are much simpler and faster than equivalent algorithms for balanced trees.

Search Operation: To search for an element, the algorithm starts at the topmost layer and moves horizontally through the elements of that layer. When it encounters an element greater than the target, it drops to the next lower layer and continues the search horizontally. This process of horizontal traversal and vertical descent continues until the target element is found or the search reaches the bottom-most layer without finding the target. The hierarchical structure allows for logarithmic search time.

Insertion and Deletion: Inserting an element involves locating the appropriate position in the bottom-most layer, placing the element there, then potentially promoting the element to higher layers. Each promotion step is independent, ensuring the probabilistic balancing of the structure. Deleting an element requires removing it from all layers in which it appears, which is straightforward once the element is located using the search algorithm. The process of updating references in multiple layers ensures that the skip list remains balanced and efficient for subsequent operations.

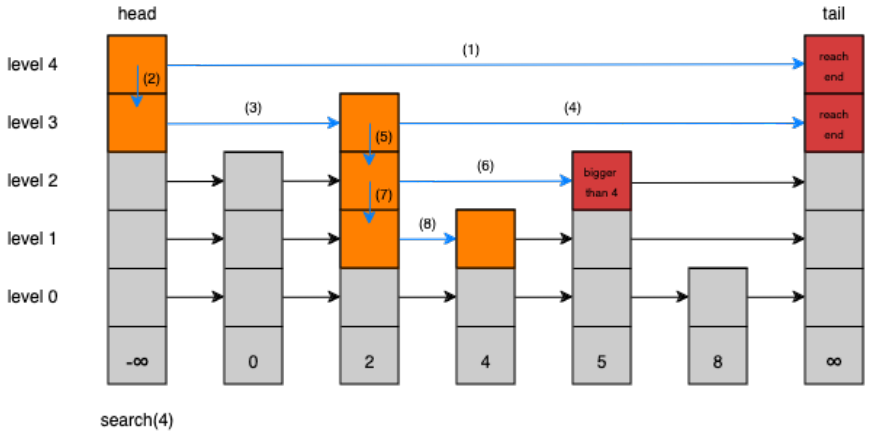


Figure 3.2: Skip List: In this example, the list searches for a `Node` with value 4. It starts on the head `Node` on the highest level, tries to move horizontally until it reaches a greater value than 4, and then goes down a level and repeats. The number noted on the arrows implies the order of the traversal.

Despite their theoretically poor worst-case performance, skip lists rarely exhibit worst-case behavior, making them efficient in most scenarios. For instance, in a dictionary with over 250 elements, the likelihood of a search taking more than three times the expected duration is less than one in a million [16]. Skip lists are ideal for implementing range locks, offering a balanced structure that improves concurrency.

Operation	**Best Case**	**Average Case**	**Worst Case**
Search, Insert, Delete	$O(1)$	$O(\log n)$	$O(n)$

Table 3.1: Time complexities of skip list operations

3.2 Concurrent Range Lock

We developed our concurrent range lock based on the design of `LockFreeSkipList` proposed by Herlihy et al. [13]. In summary, our concurrent range lock 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.

For the sake of simplicity, we use uint64_t for our pseudocode provided in this section. In our open-source C++ implementation, we use the template to enable generic programming.

3.2.1 Concurrent Range Lock API

The `ConcurrentRangeLock` class provides a concurrent mechanism to manage range-based locks. Its primary API includes methods for locking and unlocking ranges. Each range is stored in a single `Node`. 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.

The two primary methods rely heavily on `find` 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 3.2.4.

```
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 3.1: Concurrent Range Lock

3.2.2 Node

Let first understand the class `Node`, the base of our `ConcurrentRangeLock` structure.

Each `Node` contains `start` and `end`, which represents range. `Node` also uses an array of `AtomicMarkableReference` (more details in section 3.2.3) 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<Node>** next;
6
7    public:
8      Node() : next(nullptr) {}
9
10     void initialize(uint64_t start, uint64_t end, int topLevel) {
11         this->start = start; this->end = end;
12         this->topLevel = topLevel;
13
14         next = new AtomicMarkableReference<Node>*[topLevel + 1];
15         for (int i = 0; i <= topLevel; ++i) {
16             next[i] = new AtomicMarkableReference<Node>();
17         }
18     }
19
20     void initializeHead(uint64_t start, uint64_t end, int topLevel, Node* tail) {
21         initialize(start, end, topLevel);
22         for (int i = 0; i <= topLevel; ++i){
23             next[i]->store(tail, false);
24         }
25     }
26
27     int getTopLevel() const { return topLevel; }
28     uint64_t getStart() const { return start; }
29     uint64_t getEnd() const { return end; }
30 };
```

Listing 3.2: Node class implementation

3.2.3 AtomicMarkableReference

`AtomicMarkableReference` is designed to maintains an object reference (in this case `Node`) along with a mark bit, that can be updated atomically [17].

The `AtomicMarkableReference` class uses a single atomic variable, `atomicRefMark`, to store a packed representation of the reference and the mark. These values are packed and unpacked using bitwise operations, where the least significant bit represents the mark. The class offers several atomic methods:

- `store`: directly sets the reference and mark.
- `compareAndSet`: tests if the current reference and mark match the expected values and, if so, updates them to new values atomically.
- `attemptMark`: sets the mark if the reference matches the expected value.
- `get`: returns both the object's reference and sets the mark reference.
- `getReference`: returns just the reference of the object.

```
1  const uintptr_t MARK_MASK = 0x1;
2
3  class AtomicMarkableReference {
4    private:
5      std::atomic<uintptr_t> atomicRefMark;
6
7      uintptr_t pack(Node* ref, bool mark) const {
8          return reinterpret_cast<uintptr_t>(ref) | (mark ? MARK_MASK : 0);
9      }
10
11     std::pair<Node*, bool> unpack(uintptr_t packed) const {
12         return {reinterpret_cast<Node*>(packed & ~MARK_MASK), packed & MARK_MASK};
13     }
14
15   public:
16     AtomicMarkableReference() {
17         atomicRefMark.store(pack(nullptr, false), std::memory_order_relaxed);
18     }
19
20     void store(Node* ref, bool mark) {
21         atomicRefMark.store(pack(ref, mark), std::memory_order_relaxed);
22     }
23
24     bool compareAndSet(Node* expectedRef, Node* newRef, bool expectedMark, bool newMark) {
25         return atomicRefMark.compare_exchange_strong(
26             pack(expectedRef, expectedMark), pack(newRef, newMark), std::memory_order_acq_rel);
27     }
28
29     bool attemptMark(Node* expectedRef, bool newMark) {
30         auto [currentRef, currentMark] = unpack(atomicRefMark.load(std::memory_order_acquire));
```

```
31          if (currentRef == expectedRef && currentMark != newMark) {
32              return atomicRefMark.compare_exchange_strong(
33                  current, pack(expectedRef, newMark), std::memory_order_acq_rel);
34          }
35          return false;
36      }
37
38      Node* get(bool* mark) const {
39          auto [ref, currentMark] = unpack(atomicRefMark.load(std::memory_order_acquire));
40          mark[0] = currentMark;
41          return ref;
42      }
43
44      Node* getReference() const {
45          auto [ref, _] = unpack(atomicRefMark.load(std::memory_order_acquire));
46          return ref;
47      }
48  };
```

Listing 3.3: AtomicMarkableReference class implementation

3.2.4 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 ostensible 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()` (as discussed in 3.2.3) to ensure that remove all softly deleted Node on the way.
- Every `preds[]` reference is to a node with a key strictly less than the target.

```
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 3.4: Find method implementation

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 largest value on that level that is strictly less than the target key (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()`. The `compareAndSet()`

function validates that the following 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` (Line 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 needs to check if there are overlap ranges (for `findInsert`) or if the start and end of the Node match the target start and end.

- condition of findInsert
- condition of findExact

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 Evaluation

The proposed approach will be evaluated under these evaluation criteria:

- **Performance:** We will test the range lock mechanism under increasing load and concurrent accesses to measure its performance.
- **Correctness:** We will ensure the consistency and correctness of data accesses, especially when there are overlapping data ranges and concurrent operations.
- **Comparison:** We will compare the performance of the proposed solution with existing state-of-the-art approaches.

5 Result

We aim to develop a scalable range lock that performs better than the existing range locks. The evaluation results will provide insights into the performance characteristics and potential trade-offs of the proposed mechanism.

6 Conclusion

Conclusion

Bibliography

[1] D. B. Lomet. *Key range locking strategies for improved concurrency*. Digital Equipment Corporation, Cambridge Research Laboratory UK, 1993.

[2] G. Graefe. "Hierarchical locking in B-tree indexes". In: *On Transactional Concurrency Control*. Springer, 2007, pp. 45–73.

[3] C.-G. Lee, S. Noh, H. Kang, S. Hwang, and Y. Kim. "Concurrent file metadata structure using readers-writer lock". In: *Proceedings of the 36th Annual ACM Symposium on Applied Computing*. 2021, pp. 1172–1181.

[4] 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.

[5] L. Chang-Gyu, B. Hyunki, N. Sunghyun, K. Hyeongu, and Y. Kim. "Write optimization of log-structured flash file system for parallel I/O on manycore servers". In: *Proceedings of the 12th ACM International Conference on Systems and Storage*. 2019, pp. 21–32.

[6] J. Corbet. "Range reader/writer locks for the kernel". In: *LWN.net* (2022). Accessed: 2024-04-21. URL: https://lwn.net/Articles/724502/.

[7] L. Dufour. "Replace mmap_sem by a range lock". In: *LWN.net* (2017). Accessed: 2024-04-21. URL: https://lwn.net/Articles/723648/.

[8] 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.

[9] 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.

[10] J. Kara. "Implement range locks". In: *lkml.org* (2013). Accessed: 2024-04-21. URL: https://lkml.org/lkml/2013/1/31/483.

[11] 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.

[12] H. Maurice, L. Yossi, L. Victor, and S. Nir. "A provably correct scalable concurrent skip list". In: *Conference On Principles of Distributed Systems (OPODIS). Citeseer*. Vol. 103. 2006.

[13] H. Maurice, S. Nir, L. Victor, and S. Michael. *The art of multiprocessor programming*. Newnes, 2020.

[14] W. Pugh. *A skip list cookbook*. Citeseer, 1990.

[15] 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.

[16] W. Pugh. "Skip lists: a probabilistic alternative to balanced trees". In: *Communications of the ACM* 33.6 (1990), pp. 668–676.

[17] "Class AtomicMarkableReference<V>". In: (). URL: `https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicMarkableReference.html`.