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 is pivotal to the performance of various systems, including databases [1, 2], file systems [3, 4, 5], and operating systems [6, 7]. The need for more advanced and fine-grained locking mechanisms becomes critical as these systems continue to scale and increase complexity. A key challenge in this context is the management of concurrent access to shared resources. Traditional locking techniques, such as single-lock mechanisms, often introduce significant performance bottlenecks, especially in scenarios with high concurrency, thereby underscoring the necessity for more sophisticated approaches to locking.

Range locks boot performance through resource segmentation. Range lock [4, 8] provides a more refined approach to this challenge 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. Range locking is crucial in database management systems, particularly for ensuring data consistency and preventing anomalies such as "phantoms" in high-concurrency environments. When transactions require strict isolation, as under the serializable isolation level, range locks are used to secure not only the individual records within a specified range but also the gaps between these records. This prevents other transactions from inserting new records or modifying existing ones in the Range until the transaction is completed, thereby maintaining the integrity of operations that depend on the stability of a data set. Implementing range locks becomes particularly challenging in systems where transaction control (TC) and data control (DC) are separated, as the TC must lock the Range before safely interacting with the DC, despite not knowing the specific keys or records involved. Effective range-locking protocols are essential to managing this complexity, ensuring that all relevant resources are locked throughout the transaction to prevent race conditions and maintain consistency across concurrent operations.

Filesystem needs range locks. In high-performance file systems, particularly those used in large-scale and distributed computing environments, managing concurrent access to shared files is important. 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 [9, 10]. Additionally, some methods may be complex and tightly coupled with lock-based concurrency control protocols, which are not applicable for general DBMS operations [2, 11]. 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.

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 [9], 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 [14], 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. [10] introduced a dynamic range-locking design to enhance the implementation of the Linux kernel. Their range lock uses a skip list [15] 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. [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 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 [16]. 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 Approach

In this research's scope, we propose a new concurrent range-locking design. We developed our concurrent range lock based on the `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.

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 [15, 17]. 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.

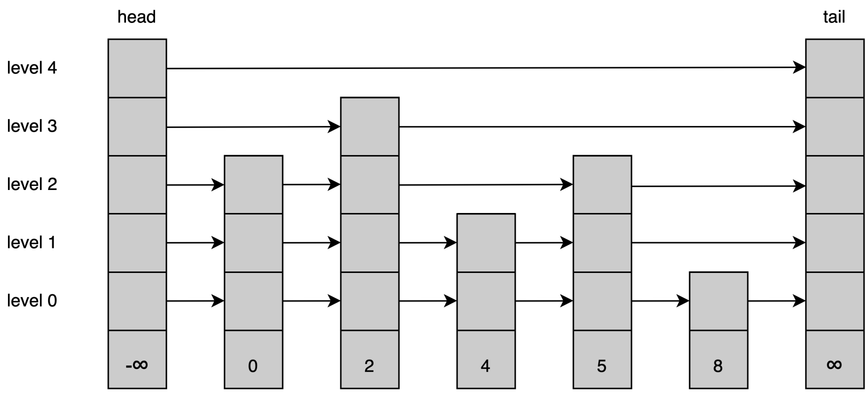


Figure 3.1: Skip List: In this example, it has five levels of sorted linked lists. Each `Node` has an unique key.

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 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.

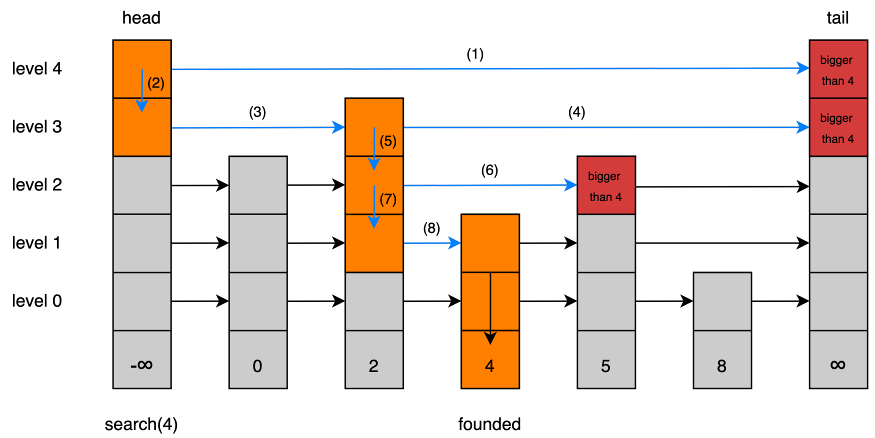


Figure 3.2: Skip List: In this example, the list searches for a `Node` with value 4. It starts on the head `Node` on the highest level, tries to move horizontally until it reaches a greater value than 4, and then goes down a level and repeats. The number noted on the arrows implies the order of the traversal.

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, and 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.

Skiplist are suitable for concurrent range lock

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 [17].

Skip lists are, therefore, 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

In this section, we will focus on the algorithm of concurrent range lock in details. 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. For furthur detail, please checkout our open-source code.

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. We will discuss these methods in subsection 3.2.5 and 3.2.6.

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.

```
class ConcurrentRangeLock {
  public:
    ConcurrentRangeLock();

    bool tryLock(uint64_t start, uint64_t end);
    bool releaseLock(uint64_t start, uint64_t end);

  private:
    Node *head, *tail;

    int randomLevel();
    bool findInsert(uint64_t start, uint64_t end, Node **preds, Node **succs);
    bool findExact(uint64_t start, uint64_t end, Node **preds, Node **succs);
    void findDelete(uint64_t start, uint64_t end);
};
```

Listing 3.1: Pseudocode for ConcurrentRangeLock API

3.2.2 Node

`Node` is 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.

```
class Node {
  private:
    uint64_t start, end;
    int topLevel;
    AtomicMarkableReference** next = nullptr;

  public:
    void initialize(uint64_t start, uint64_t end, int topLevel) {
        this->start = start; this->end = end;
        this->topLevel = topLevel;

        next = new AtomicMarkableReference*[topLevel + 1];
        for (int i = 0; i <= topLevel; ++i) {
            next[i] = new AtomicMarkableReference();
        }
    }

    void initializeHead(uint64_t start, uint64_t end, int topLevel, Node* tail) {
        initialize(start, end, topLevel);
        for (int i = 0; i <= topLevel; ++i){
            next[i]->store(tail, false);
        }
    }

    int getTopLevel() const { return topLevel; }
    uint64_t getStart() const { return start; }
    uint64_t getEnd() const { return end; }
};
```

Listing 3.2: Pseudocode for Node structure

3.2.3 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 3.3 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`.

To retrieve the current reference and mark, the `get` method is used, which stores the mark in a provided boolean pointer. In contrast, the `getReference` method simply returns the current reference without accessing the mark.

```cpp
class AtomicMarkableReference {
  private:
    std::atomic<uintptr_t> atomicRefMark;

    uintptr_t pack(Node* ref, bool mark) const {
        return reinterpret_cast<uintptr_t>(ref) | (mark ? 1 : 0);
    }

    std::pair<Node*, bool> unpack(uintptr_t packed) const {
        return {reinterpret_cast<Node*>(packed & -1), packed & 1};
    }

  public:
    AtomicMarkableReference() {
        atomicRefMark.store(pack(nullptr, false), std::memory_order_relaxed);
    }

    void store(Node* ref, bool mark) {
        atomicRefMark.store(pack(ref, mark), std::memory_order_relaxed);
    }

    bool compareAndSet(Node* expectedRef, Node* newRef, bool expectedMark, bool newMark) {
        return atomicRefMark.compare_exchange_strong(
            pack(expectedRef, expectedMark), pack(newRef, newMark), std::memory_order_acq_rel);
    }

    bool attemptMark(Node* expectedRef, bool newMark) {
        auto [currentRef, currentMark] = unpack(atomicRefMark.load(std::memory_order_acquire));
        if (currentRef == expectedRef && currentMark != newMark) {
            return atomicRefMark.compare_exchange_strong(
                current, pack(expectedRef, newMark), std::memory_order_acq_rel);
        }
        return false;
    }

    Node* get(bool* mark) const {
        auto [ref, currentMark] = unpack(atomicRefMark.load(std::memory_order_acquire));
        mark[0] = currentMark;
        return ref;
    }

    Node* getReference() const {
        auto [ref, _] = unpack(atomicRefMark.load(std::memory_order_acquire));
        return ref;
    }
};
```

Listing 3.3: AtomicMarkableReference

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.

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

```
bool ConcurrentRangeLock::find(uint64_t start, uint64_t end,
    Node **preds, Node **succs) {
    bool marked[1] = {false};
    bool snip;
    Node *pred, *succ, *curr;

  retry:
    while (true) {
    pred = head;
    for (int level = maxLevel; level >= 0; level--) {
        curr = pred->next[level]->getReference();

        while (start > curr->getStart()) {
            succ = curr->next[level]->get(marked);
            while (marked[0]) {
                snip = pred->next[level]->compareAndSet(curr, succ, false, false);

                if (!snip) goto retry;

                curr = pred->next[level]->getReference();
                succ = curr->next[level]->get(marked);
            }
            if (start >= curr->getStart()) {
                pred = curr;
                curr = succ;
            } else {
                break;
            }
        }

        preds[level] = pred;
        succs[level] = curr;
    }

    return **condition**;
    }
}
```

Listing 3.4: General pseudocode for find methods 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 (`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.

3.2.5 Try Lock

The tryLock method, shown in Listing 3.5, 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.

Algorithm in details

The tryLock method, shown in listing 3.5, uses findInsert(), show in listing 3.4, 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. Once all levels are linked, the method returns true (line 27). The releaseLock method, shown in listing 3.6, calls findExact() to determine whether an

```
bool ConcurrentRangeLock::tryLock(uint64_t start, uint64_t end) {
    int topLevel = randomLevel();
    Node *preds[maxLevel + 1];
    Node *succs[maxLevel + 1];

    while (true) {
        if (findInsert(start, end, preds, succs)) {
            return false;
        } else {
            auto newNode = new Node();
            newNode->initialize(start, end, topLevel);

            for (int level = 0; level <= topLevel; ++level) {
                newNode->next[level]->store(succs[level], false);
            }

            if (!preds[0]->next[0]->compareAndSet(succs[0], newNode, false, false)) {
                continue;
            }

            for (int level = 1; level <= topLevel; ++level) {
                while (!preds[level]->next[level]->compareAndSet(
                    succs[level], newNode, false, false)) {
                    findInsert(start, end, preds, succs);
                }
            }

            return true;
        }
    }
}
```

Listing 3.5: Pseudocode for tryLock method 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.

3.2.6 Release Lock

The `releaseLock` method first calls `findExact()` to locate an unmarked node with the specified range `[start, end]`. If found, it marks all levels of the node 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 state of the node.

Algorithm in details

The `releaseLock` method, shown in listing 3.6, 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

```
bool ConcurrentRangeLock::releaseLock(uint64_t start, uint64_t end) {
    Node *preds[maxLevel + 1];
    Node *succs[maxLevel + 1];
    Node *succ;

    while (true) {
        bool found = findExact(start, end, preds, succs);
        if (!found) {
            return false;
        } else {
            Node *nodeToRemove = succs[0];
            for (int level = nodeToRemove->getTopLevel();
                    level >= 0 + 1; level--) {
                bool marked[1] = {false};
                succ = nodeToRemove->next[level]->get(marked);
                while (!marked[0]) {
                    nodeToRemove->next[level]->attemptMark(succ, true);
                    succ = nodeToRemove->next[level]->get(marked);
                }
            }

            bool marked[1] = {false};
            succ = nodeToRemove->next[0]->get(marked);
            while (true) {
                bool iMarkedIt = nodeToRemove->next[0]->compareAndSet(
                        succ, succ, false, true);
                succ = succs[0]->next[0]->get(marked);
                if (iMarkedIt) {
                    findDelete(start, end);

                    return true;
                } else if (marked[0]) {
                    return false;
                }
            }
        }
    }
}
```

Listing 3.6: Pseudocode for releaseLock method 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 Evaluation

In this section, we will evaluate our proposed concurrent range lock under diffrerents senarios. The goal is to see the scalability, throughtput as well as the latency of our mechanism comparing to state-of-the-art range lock.

4.1 Competitor

We denote our main 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. [8], denoted as `ver2`. 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 skiplist range lock proposed by Song et al. [10], which we denote as `ver3`.

4.2 Benchmark enviroment

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.

4.3 Microbenchmark

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

4.3.2 Optimal Height for Range Locking

The height of the skip list in our range locking mechanism plays a critical role in balancing performance and resource utilization. A skip list with insufficient height may fail to optimize lookup times effectively, leading to slower performance, while 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 three workload described in 4.3.1 to identify the best skip list height.

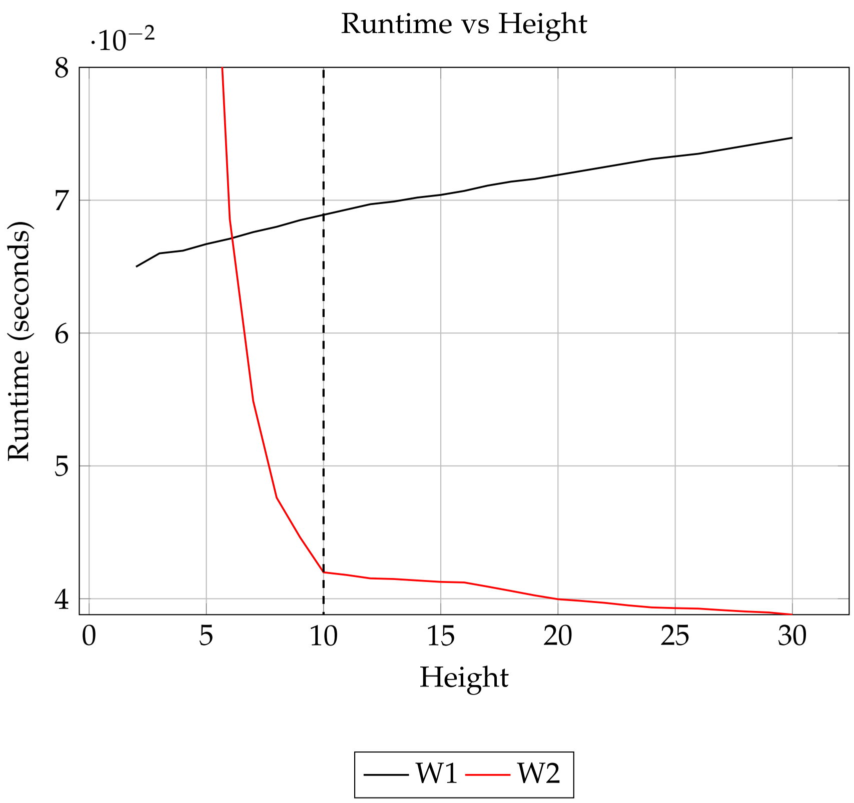


Figure 4.1: Runtime for different heights of range lock

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.

4.3.3 Result

Figure 4.2 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 4.4 or **W2**, we will see `Song RL` perform much better.

Figure 4.3 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.

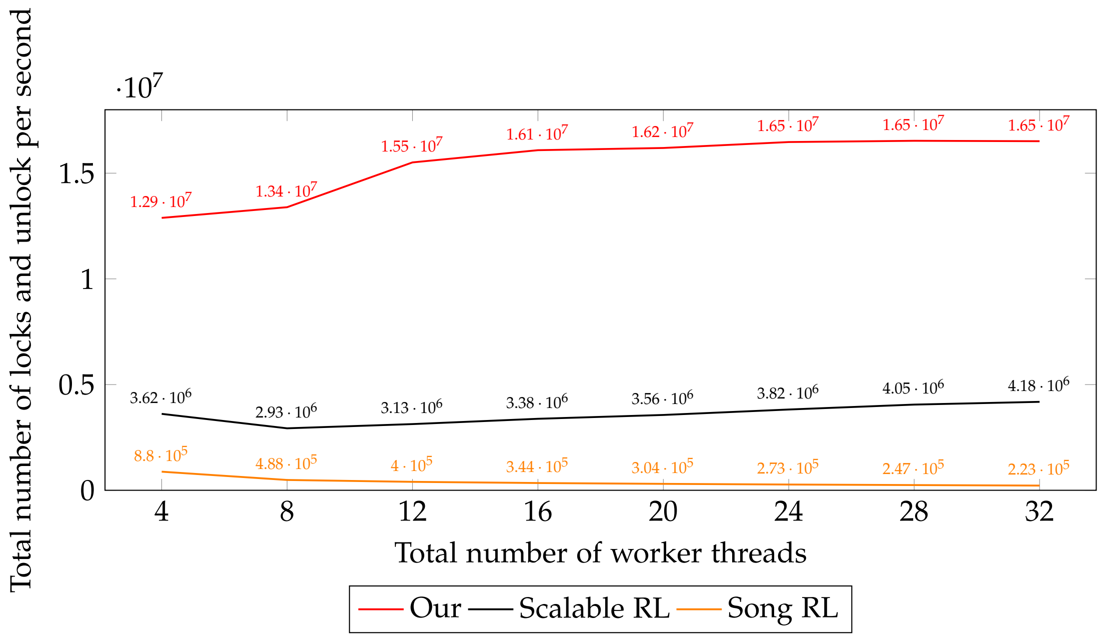


Figure 4.2: Microbenchmark for workload **W1**

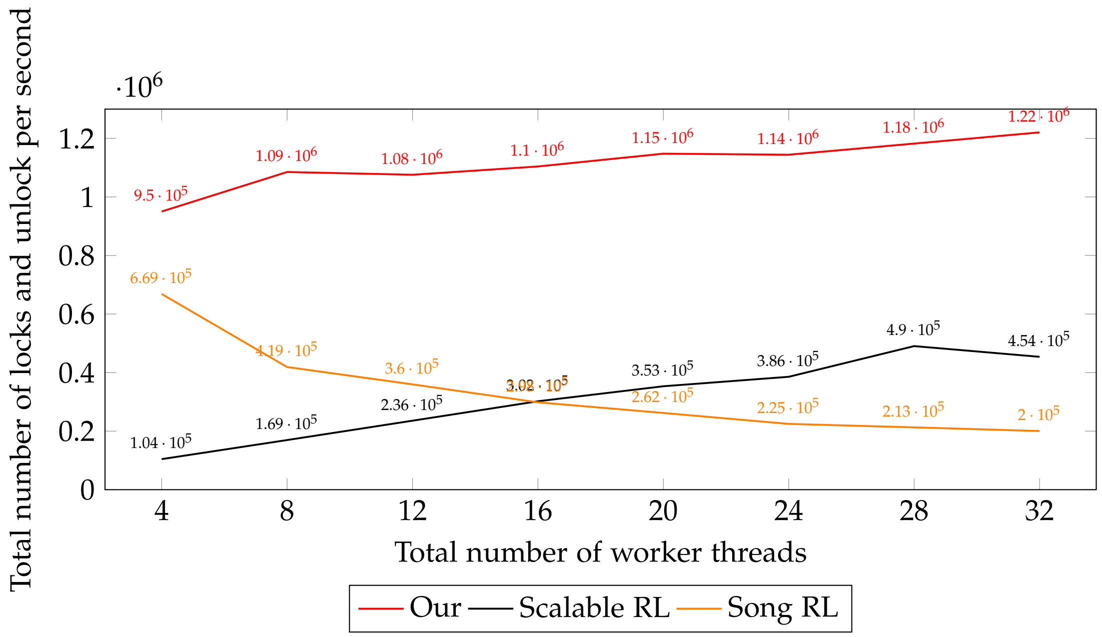


Figure 4.3: Microbenchmark for workload **W2**

4.4 Leanstore

`Leanstore` [18] is a high-performance storage engine designed to support various database management systems. To further enhance its capabilities, Nguyen et al. [19] introduced a new, 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 are used to 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. To prevent concurrent workers from accessing overlapping memory regions, a range lock mechanism is employed. 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 at a time. This prevents race conditions and ensures data consistency while allowing multiple workers to operate on different memory ranges simultaneously.

4.4.1 Competitors

We integrated all three versions as described in Section 4.3. 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.

4.4.2 Workload

The experiment, utilizing synthetic YCSB [20] 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 a total of 1000 records being processed. The experiment also incorporates a buffer management configuration that allocates 128

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

4.4.3 Result

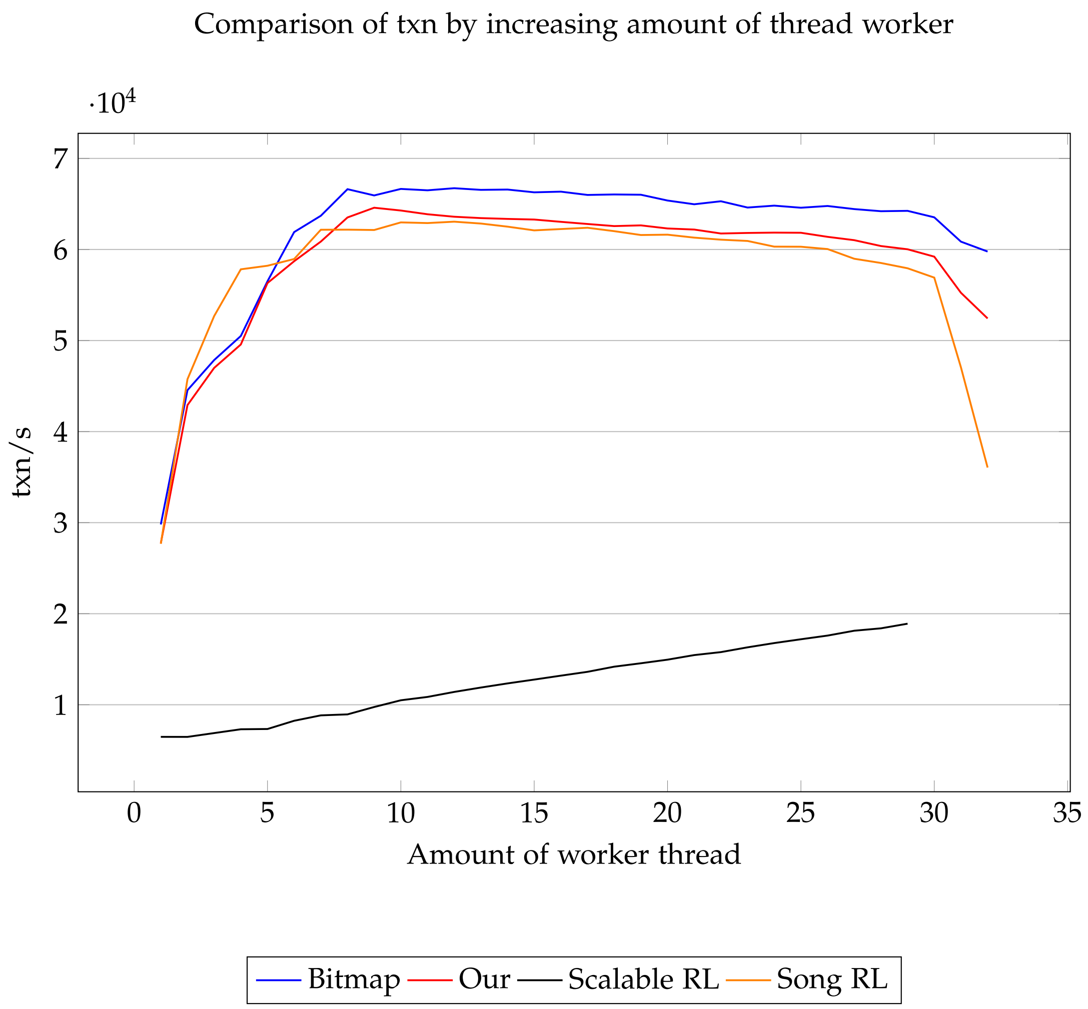


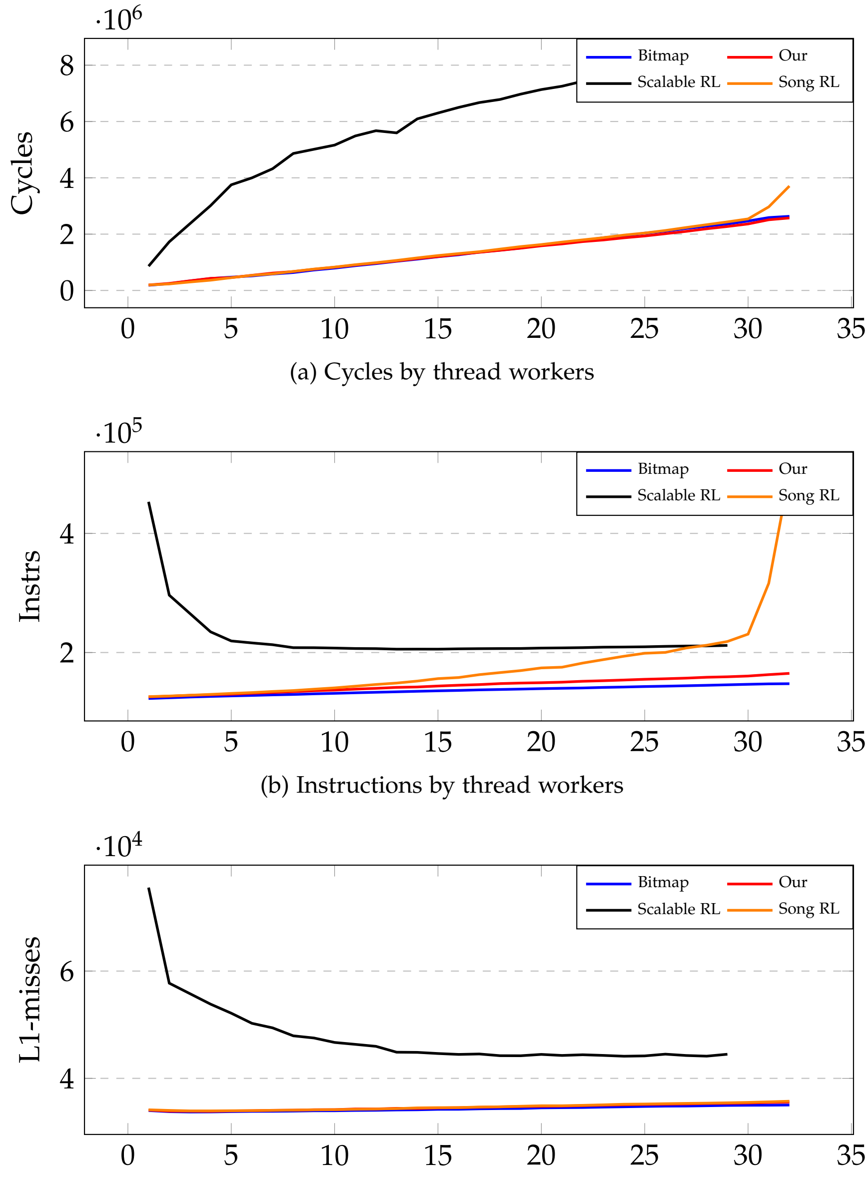
Figure 4.4: Comparison of transactions by increasing the number of thread workers.

We evaluated the throughput of four distinct variants as the number of worker threads increased, as depicted in Figure 4.4. As anticipated, the `Bitmap` implementation consistently outperformed the other three variants. This superiority can be largely attributed to its optimized use of the aliasing arena, which makes it the most efficient in handling the workload. 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 strong 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 largely due 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 L2 cache capacity (16MB in our machine), resulting in significant contention at the L2 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. Given an average throughput of approximately 60,000 operations per second, this equates to 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 4.5, 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.



(a) Cycles by thread workers

(b) Instructions by thread workers

(c) L1-misses by thread workers

Figure 4.5: Comparison of cycles, instrs, and L1-misses by increasing the number of thread workers.

5 Conclusion

Conclusion

6 Future work

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

[10] 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.

[11] 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`.

[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] 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.

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

[16] 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.

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

[18] 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.

[19] L.-D. Nguyen and V. Leis. "Why Files If You Have a DBMS". In: ICDE. 2024.

[20] 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.