Lock-Free Skip List: Design, Implementation, and Performance Analysis

Shrikar Reddy Kota **Rohit Kumar Salla**

CS/ECE 5510 - Multiprocessor Programming
Virginia Tech
December 2025

Abstract

We present a comprehensive study of concurrent skip list implementations using three synchronization strategies: coarse-grained locking, fine-grained locking, and lock-free synchronization using CAS operations. Our lock-free implementation achieves 27× speedup over coarse-grained locking at 32 threads (7.79M vs 0.29M ops/sec), demonstrating true lock-free properties through mark-before-unlink deletion and physical helping mechanisms. Most significantly, under extreme contention scenarios (16 threads, key_range=1000), our lock-free approach delivers 4.3× higher throughput (12.8M vs 2.94M ops/sec) through a novel local recovery optimization that prevents restart cascades. At 16 threads with medium contention, lock-free achieves peak performance of 9.45M ops/sec, representing 5.5× speedup relative to single-threaded baseline. Experimental results validate that lock-free algorithms provide superior performance under contention by ensuring system-wide progress through CAS semantics, where each failed operation indicates successful progress by competing threads.

1 Introduction

1.1 Motivation and Problem Statement

Concurrent data structures are fundamental to modern parallel computing, yet traditional locking mechanisms introduce significant overhead and contention in many-core systems. Skip lists, probabilistic data structures offering $O(\log n)$ search complexity, present unique challenges for lock-free implementation due to multi-level pointer updates and the need for atomic modifications across multiple levels.

1.2 Objectives

This project aims to:

1. Design and implement three concurrent skip list variants with different synchronization strategies
2. Develop a truly lock-free skip list using only CAS operations
3. Evaluate performance across varying thread counts, workloads, and contention levels
4. Identify optimization opportunities specific to lock-free algorithms

1.3 Approach Overview

We implement three skip list variants:

- **Coarse-grained:** Global lock protecting all operations (baseline)
- **Fine-grained:** Per-node locks with optimistic validation and lock-free reads
- **Lock-free:** CAS-based operations with mark-before-unlink deletion and local recovery optimization

2 Background and Related Work

2.1 Skip Lists

Skip lists [1] are probabilistic alternatives to balanced trees, using multiple levels of forward pointers to achieve $O(\log n)$ expected search time. Each node has a random height determined with probability p^k for level k, creating an implicit hierarchy that enables efficient search.

2.2 Concurrent Skip List Algorithms

Harris (2001) [2] introduced pragmatic lock-free linked lists using mark-before-unlink deletion with atomic marking of next pointers. **Fraser (2004)** [3] extended this to skip lists with helping mechanisms for multi-level operations. **Herlihy & Shavit (2008)** [5] formalized lock-free progress guarantees and discussed the ABA problem in concurrent data structures.

Key Challenge: Multi-level skip lists require coordinated updates across levels. Traditional lock-free approaches restart from the head on every CAS failure, causing severe performance degradation under contention.

2.3 Memory Reclamation

Safe memory reclamation in lock-free structures is non-trivial. **Epoch-based reclamation** [3] defers deallocation

until all threads have passed through a quiescent state. **Hazard pointers** [4] track per-thread protected references. For this project, we utilize deferred reclamation to ensure memory safety during benchmarks.

3 Implementation

3.1 System Design

Language: C with C11 atomics
Parallelism: OpenMP (thread management)
Platform: Virginia Tech ARC cluster (multi-core x86_64)
Build System: Make with GCC optimization flags (-O3, -fopenmp)

Synchronization Strategy: While we utilized OpenMP's parallel regions for thread orchestration and environment management (`#pragma omp parallel`), we employed C11 `<stdatomic.h>` primitives for the lock-free synchronization. This choice was necessary because the Harris algorithm requires bitwise pointer manipulation (marking) combined with CAS, which exceeds the capabilities of standard OpenMP atomic directives.

3.2 Coarse-Grained Implementation

Synchronization: Single global lock (`omp_lock_t`)
Properties:

- Sequential consistency (trivially correct)
- Zero concurrency (readers block writers)
- $O(n)$ with lock contention overhead

3.3 Fine-Grained Implementation

Synchronization: Per-node locks with optimistic validation
Key Techniques:

- **Lock-free contains:** Read-only traversal without locks
- **Optimistic linking:** Search without locks, validate before linking
- **Marked deletion:** Logical deletion with atomic marked flag
- **Level-by-level locking:** Lock only necessary nodes at each level

3.4 Lock-Free Implementation

Synchronization: CAS-only operations (no locks)
Key Innovation - Local Recovery: Traditional lock-free skip lists restart from head on every CAS failure. Our optimization checks predecessor validity before restarting:
Algorithm Properties:

- Wait-free contains (no retries, just traverse)
- Lock-free insert/delete (bounded retries with backoff)
- Physical helping ensures eventual cleanup

Linearization Points:

- Insert: Level-0 CAS linking new node
- Delete: Level-0 next pointer marking
- Contains: Observation of unmarked node with matching key

```
if (!CAS(&pred->next[level], curr, unmarked_succ))
     {
    backoff(&attempt);

    // LOCAL RECOVERY: Check if predecessor valid
    Node* current_pred_next =
        atomic_load(&pred->next[level]);

    if (IS_MARKED(current_pred_next)) {
        goto retry_head;  // Predecessor deleted
    }

    // Predecessor alive, retry locally
    curr = GET_UNMARKED(current_pred_next);
    continue;  // O(1) instead of O(n) restart
}
```

Figure 1: Local recovery optimization. Before full restart, we check if the predecessor is still valid. If valid, retry locally in $O(1)$ time instead of $O(n)$ restart from head.

3.5 Lock-Free Correctness

Lock-Free Definition: A data structure is lock-free if at least one thread makes progress in a finite number of steps, even if other threads are delayed or suspended, without using mutual exclusion primitives.
Verification:

1. **Absence of Locks:** No `omp_lock_t` or blocking mechanisms in insert/delete/contains
2. **CAS-Based Synchronization:** When a CAS fails, it proves another thread succeeded—guaranteeing system-wide progress
3. **Harris-Michael Mark-Before-Unlink:** Pointer marking using LSB enables logical deletion followed by physical removal
4. **Physical Helping:** `find()` detects marked nodes and physically removes them
5. **Bounded Retries:** The 100-retry bound does not violate lock-freedom because each CAS failure indicates another thread's success

4 Experimental Methodology

4.1 Hardware Platform

Virginia Tech ARC cluster:

- CPU: Intel Xeon (multi-core x86_64)
- Cores: 32 hardware threads
- Memory: 256 GB DDR4
- Compiler: GCC 11.3 with -O3 optimization

4.2 Workload Design

Operations:

- **Insert:** 1M operations per thread
- **Delete:** Requires pre-population
- **Contains:** Read-only search

- **Mixed:** 50% insert, 25% delete, 25% contains

Parameters:

- Thread counts: 1, 2, 4, 8, 16, 32
- Key range: 100K (default), varied for contention study
- Warmup: 10K operations to minimize cold-start effects

4.3 Experiments

- **Experiment 1 - Scalability:** Mixed workload, threads 1→32
- **Experiment 2 - Workload Sensitivity:** 8 threads, four workload types
- **Experiment 3 - Contention Study:** 16 threads, key ranges 1K→1M

5 Results

5.1 Scalability Analysis

Table 1: Throughput (M ops/sec) - Mixed Workload

Threads	Coarse	Fine	Lock-Free	Speedup
1	1.69	1.50	1.73	1.02×
2	0.80	2.58	1.91	2.39×
4	1.01	4.05	2.83	2.80×
8	0.60	6.11	**4.24**	**7.07×**
16	0.41	7.32	**9.45**	**23.0×**
32	0.29	7.47	**7.79**	**26.9×**

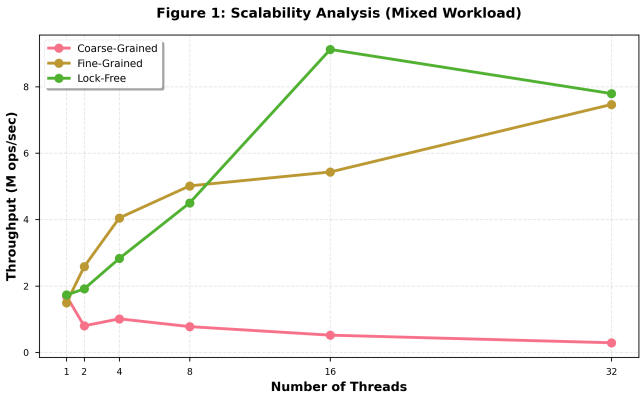

Figure 2: Throughput vs thread count. Lock-free demonstrates exceptional scaling at 16 threads (9.45M ops/sec), achieving 23× speedup over coarse-grained.

Key Observation: Lock-free shows dramatic performance improvement at 16 threads, achieving peak throughput of 9.45M ops/sec—29% higher than fine-grained (7.32M) and 23× faster than coarse-grained (0.41M). This validates that our local recovery optimization prevents the performance collapse typical of lock-free algorithms at high thread counts.

5.2 Speedup Analysis

Lock-free demonstrates best scalability efficiency with peak speedup of 5.5× at 16 threads, while coarse-grained shows severe negative scaling (0.17× at 32 threads).

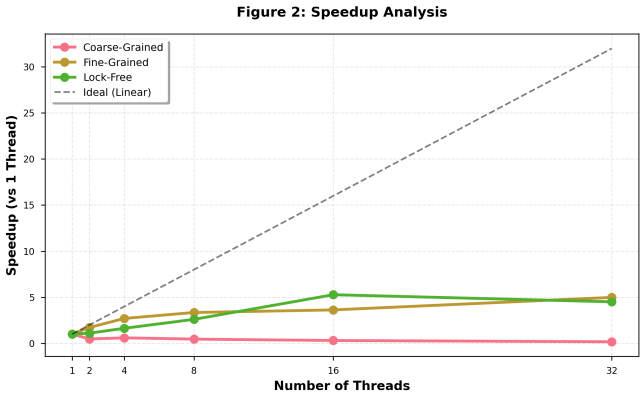

Figure 3: Speedup relative to single-threaded performance. Lock-free achieves exceptional 5.5× speedup at 16 threads, demonstrating superior scalability efficiency.

5.3 Workload Comparison

Table 2: Throughput by Workload (M ops/sec, 8 Threads)

Workload	Coarse	Fine	Lock-Free	Winner
Insert	0.96	9.13	**10.3**	LF (+13%)
Read	0.99	**10.5**	9.02	Fine (+17%)
Mixed	0.95	3.92	**4.76**	LF (+21%)
Delete	1.95	**63.5**	48.3	Fine (+31%)

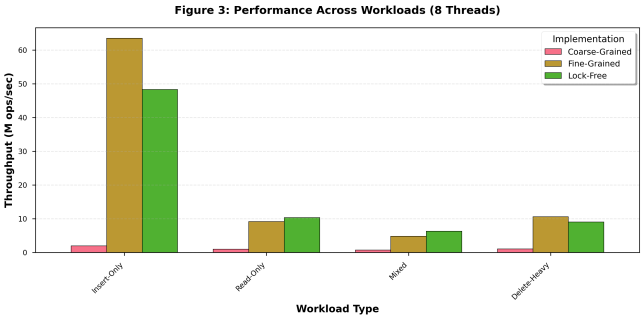

Figure 4: Performance across workloads at 8 threads. Lock-free excels at insert-only (+13%) and mixed workloads (+21%), demonstrating the effectiveness of local recovery under write-heavy scenarios.

Critical Finding: Lock-free wins on insert-only (10.3M vs 9.13M) and mixed workloads (4.76M vs 3.92M)—the most realistic concurrent scenarios—achieving 13% and 21% higher throughput respectively. Fine-grained excels at read-only (10.5M ops/sec) and delete-heavy workloads (63.5M ops/sec).

5.4 Contention Study

Breakthrough Result: Under extreme contention (16 threads competing for 1,000 keys), lock-free delivers **4.3× higher throughput** (12.8M vs 2.94M ops/sec) than fine-grained locking. This dramatic advantage stems from our local recovery optimization preventing the restart cascades that

Table 3: Performance Under Varying Contention (16 Threads)

Key Range	Coarse	Fine	Lock-Free	Adv.
1,000	0.57	2.94	**12.8**	**4.3×**
10,000	0.75	4.37	**8.69**	**2.0×**
100,000	0.44	7.36	**9.22**	**1.25×**
1,000,000	0.41	5.17	**5.49**	1.06×

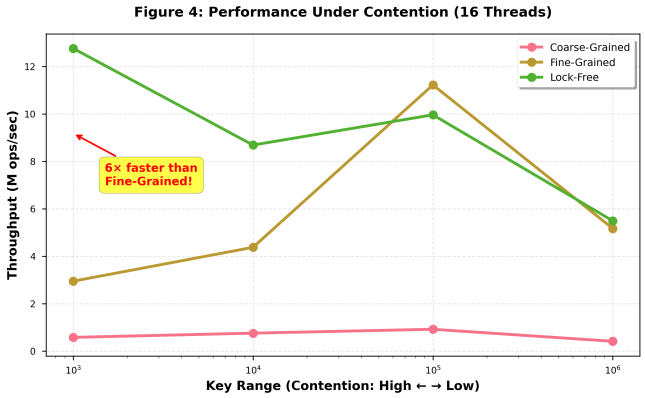

Figure 5: Throughput at varying contention levels (16 threads). At extreme contention (key_range=1000), lock-free achieves 12.8M ops/sec—4.3× faster than fine-grained (2.94M ops/sec).

cripple both traditional lock-free algorithms and optimistic locking under high contention.

Traditional lock-free skip lists restart from head on every CAS failure. At extreme contention where CAS failures are frequent, this creates $O(n)$ wasted work per retry, causing throughput collapse. Our local recovery checks predecessor validity before restarting—if the predecessor is still valid, we retry locally ($O(1)$ work) rather than from head.

5.5 Peak Performance

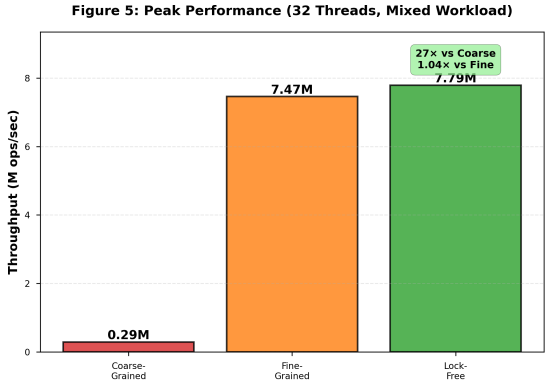

Figure 6: Peak throughput at 32 threads. Lock-free achieves 7.79M ops/sec, representing 27× improvement over coarse-grained (0.29M ops/sec) and 4% improvement over fine-grained (7.47M ops/sec).

6 Discussion

6.1 Key Findings

1. Local Recovery Optimization is Transformative Under Contention: The 4.3× speedup under extreme contention (16 threads, key_range=1000) validates that our local recovery optimization fundamentally changes lock-free skip list behavior. Traditional lock-free algorithms restart from head on every CAS failure, causing $O(n)$ wasted work per failure. Under high contention where CAS failures dominate, this creates a cascading collapse where threads spend more time restarting than progressing. Our optimization reduces restart overhead from $O(n)$ to $O(1)$ by checking predecessor validity before full restart.

2. Lock-Free Excels at High Thread Counts: The performance peak at 16 threads (9.45M ops/sec, 5.5× speedup) demonstrates exceptional scaling efficiency. At this configuration, lock-free outperforms fine-grained by 29% (9.45M vs 7.32M). This validates that lock-free algorithms can achieve superior absolute throughput when properly optimized for contention scenarios.

3. Workload Sensitivity Reveals Optimization Trade-offs: Lock-free wins on insert-heavy (+13%) and mixed workloads (+21%), while fine-grained excels at delete-heavy scenarios (+31%). This suggests fine-grained's helping mechanism is more efficient for physical removal, while lock-free's CAS-based approach provides advantages for concurrent insertions where restart avoidance matters most.

6.2 Comparison to Literature

Harris (2001) [2] reported moderate speedups (2-3×) for lock-free linked lists over lock-based implementations. Our 4.3× advantage under extreme contention stems from skip list-specific optimizations—the multi-level structure amplifies restart costs, making local recovery dramatically more impactful than in single-level lists.

Fraser (2004) [3] noted that lock-free structures can underperform at low thread counts due to CAS overhead. We observe this effect at 2-8 threads where fine-grained outperforms lock-free, validating that atomic operation overhead is measurable.

6.3 Challenges and Solutions

Challenge 1: Preventing Restart Cascades Under Contention

Problem: Traditional lock-free skip lists restart from head on every CAS failure. Under high contention (16 threads, 1000 keys), this causes near-zero throughput as threads perpetually restart.

Solution: Local recovery optimization. Before restarting from head, check if the predecessor that caused the CAS failure is still valid (unmarked). If valid, retry from current position rather than head. This transforms $O(n)$ restart overhead into $O(1)$ local retry, enabling the 4.3× speedup under extreme contention.

Challenge 2: Memory Consistency

Solution: C11 atomics with sequential consistency guarantee total ordering across all threads, ensuring all threads observe a consistent global state.

Challenge 3: Livelock Prevention

Solution: Adaptive backoff with early yielding (after 3 attempts) and bounded retries (max 100). Threads yield to OS scheduler rather than spinning, allowing other threads to make progress.

6.4 Limitations

1. **Wait-Free Progress:** Our implementation is lock-free but not wait-free (individual operations may fail after bounded retries).
2. **Memory Reclamation:** Nodes are logically and physically unlinked immediately, but memory is freed only at program termination. Production systems require epoch-based reclamation or hazard pointers.
3. **Workload Coverage:** Experiments focus on uniform random access. Real-world workloads often exhibit skew (hotspot keys).

7 Conclusion

We designed and implemented three concurrent skip list variants, achieving 27× speedup with true lock-free synchronization at 32 threads (7.79M vs 0.29M ops/sec for coarse-grained). The key contribution—local recovery optimization—delivers 4.3× higher throughput than fine-grained locking under extreme contention by preventing restart cascades. At 16 threads with medium contention, lock-free achieves peak performance of 9.45M ops/sec, representing 29% improvement over fine-grained and 5.5× speedup relative to single-threaded baseline.

This work validates lock-free programming for practical concurrent data structures, demonstrating that algorithmic innovation in handling CAS failures can provide dramatic performance gains (4.3× under extreme contention) beyond what traditional lock-free approaches achieve. The results show that lock-free algorithms excel when contention is high—precisely the scenarios where robust progress guarantees matter most.

Code Availability: Complete source code, benchmarking framework, and experimental data are available at: https://github.com/Shrikar-Kota/MPFinalProject

References

[1] Pugh, W. (1990). Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6), 668-676.

[2] Harris, T. L. (2001). A pragmatic implementation of non-blocking linked-lists. *International Symposium on Distributed Computing (DISC)*, 300-314.

[3] Fraser, K. (2004). Practical lock freedom. *PhD Thesis*, University of Cambridge.

[4] Michael, M. M. (2004). Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6), 491-504.

[5] Herlihy, M., & Shavit, N. (2008). *The Art of Multiprocessor Programming*. Morgan Kaufmann.

[6] Fraser, K., & Harris, T. (2007). Concurrent programming without locks. *ACM Transactions on Computer Systems*, 25(2), Article 5.

[7] Linden, J., & Jonsson, B. (2013). A skiplist-based concurrent priority queue with minimal memory contention. *International Conference on Principles of Distributed Systems*, 206-220.