Parallel Programming
אלכסנדר פיטקין 333879013

איליה וורטינצב 324755735

Final Project

https://github.com/iliavrtn/pp-final-project

Abstract/Introduction

In modern concurrent programming, achieving high performance without locks often relies on the use of lock-free data structures. However, a significant challenge that arises in such systems is memory reclamation: safely freeing memory occupied by nodes that have been removed from the data structure while ensuring that no other thread is still accessing them. This problem is particularly acute in languages like C and C++ where automatic garbage collection is not available. If memory is reclaimed too early, it may lead to dangerous access of freed memory, but delaying reclamation indefinitely risks unbounded memory growth.

Two notable techniques have been proposed to address this challenge. The first, known as the *hazard pointers* technique, requires each thread to publish a small set of pointers indicating which nodes it might access. A reclaiming thread then scans these published pointers and only frees a node once it is certain that no thread holds a reference to it. The second approach, called *ThreadScan*, takes a more automated route by using operating system signals to force threads to scan their own stacks and registers for references to retired nodes. In ThreadScan, when a node is removed, its pointer is added to a shared delete buffer; once this buffer fills, a reclamation process is initiated that marks nodes still in use and frees only those that are unmarked.

Both methods aim to solve the same core problem—ensuring that memory reclamation in lock-free data structures is both safe and efficient—but they differ in their techniques. *Our work* builds on these ideas by proposing a further enhancement to ThreadScan: replacing its delete buffer with an AVL tree to eliminate the need for repeated sorting, potentially reducing reclamation overhead even further.

Summary

Hazard Pointers:

One of the central problems in concurrent programming—safely reclaiming memory in lock-free data structures. Lock-free data structures promise high performance and resilience in multiprocessor systems, but once a node is removed, it remains a potential hazard if another thread still accesses it. The paper introduces the concept of **hazard pointers**, which are essentially a set of per-thread markers that indicate the nodes a thread is about to access. Each thread publishes a small number of these pointers, and before reclaiming a node, the system scans all threads' hazard pointers to ensure that no pointer is referencing the node in question.

The article's central claim is that by using hazard pointers, one can achieve safe memory reclamation in a wait-free manner, using only single-word reads and writes. Researcher supports his claims with both proofs and experimental results, demonstrating that hazard pointers offer performance that is comparable to—and often better than—efficient lock-based implementations, particularly under moderate contention or multiprogramming conditions.

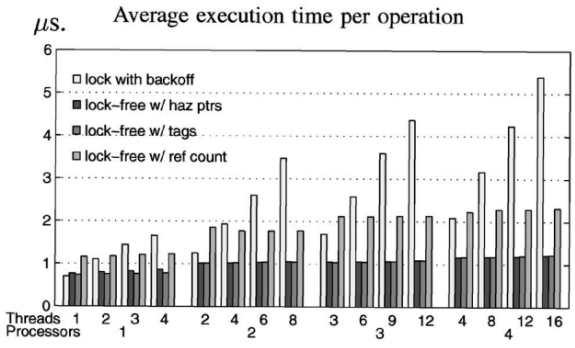


Fig. 11. Performance of FIFO queues.

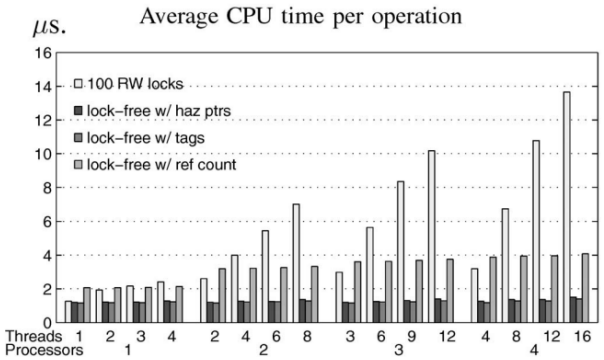


Fig. 13. Performance of hash tables with load factor 1.

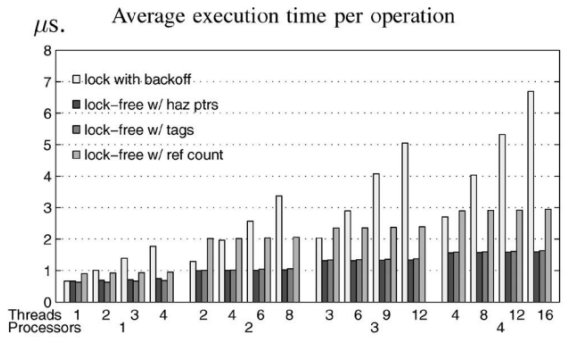


Fig. 12. Performance of LIFO stacks.

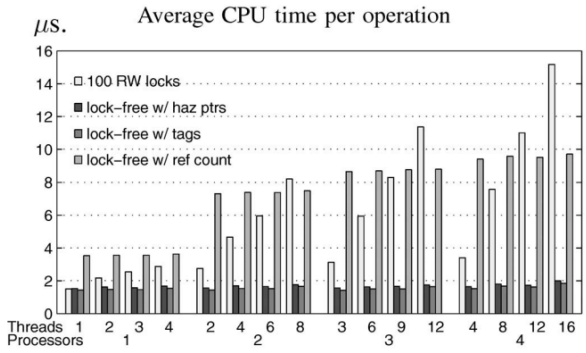


Fig. 14. Performance of hash tables with load factor 5.

ThreadScan:

The second article presents a distinct approach to the same memory reclamation challenge. Instead of requiring every thread to explicitly publish which nodes it is accessing, **ThreadScan** leverages the natural behaviour of unsynchronized traversals. When a thread deletes a node, the pointer to that node is not immediately freed; instead, it is placed into a shared delete buffer. Once the buffer fills up, a reclamation phase is triggered.

During this phase, the reclaiming thread sends signals to all other threads, prompting them to execute a signal handler that scans their own stacks and registers for any references to the nodes in the delete buffer. Each thread then marks the nodes it still references. When the reclaiming thread resumes, it reexamines the buffer and frees only those nodes that were not marked by any thread—ensuring that no active thread holds a reference to the memory being reclaimed.

The researchers provides proofs of correctness and liveness, along with extensive experimental results that indicate ThreadScan can match or even outperform other memory reclamation techniques.

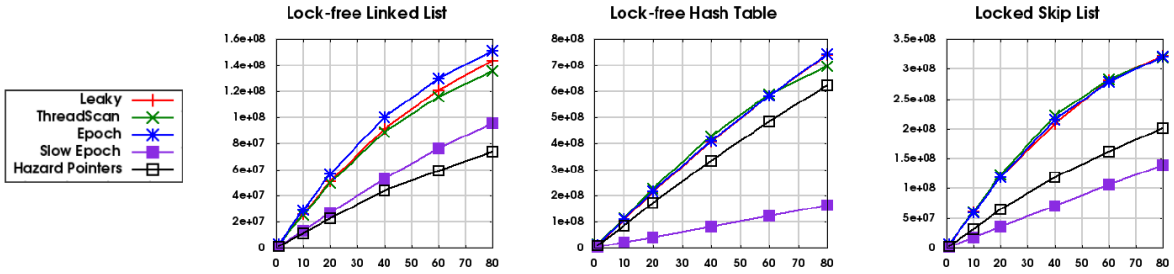


Figure 3: Throughput results for the lock-free linked list, lock-free hash table, and locked skip list: X-axis shows the number of threads, and Y-axis the total number of completed operations.

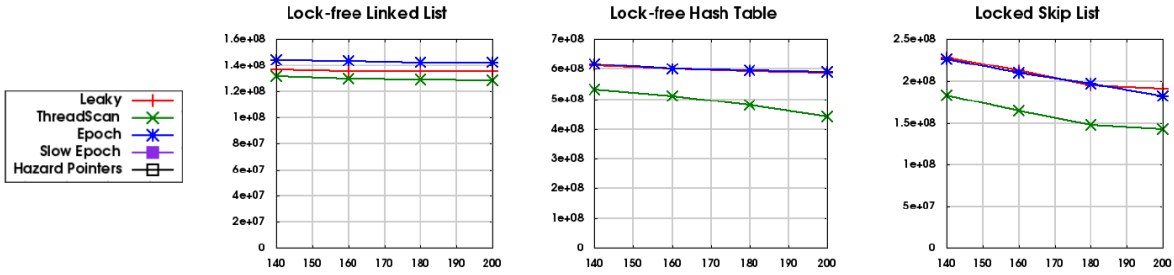


Figure 4: Throughput results for the oversubscribed system.

Common idea between the articles:

Both articles address the same fundamental problem: ensuring that memory reclaimed from removed nodes in concurrent, lock-free data structures is safe—that is, that no other thread will later access freed memory.

Suggestions

Our proposed modification targets a reclamation phase of **ThreadScan**. In the original implementation, when a node is removed from a data structure, its pointer is added to a shared delete buffer. Once the buffer reaches a predefined capacity, the algorithm initiates a reclamation process. During this process, each thread scans its own stack and registers, and the delete buffer is sorted (using a function called `threadscan_util_sort`)—typically employing quick sort or insertion sort based on the hardcoded SORT_THRESHOLD.

Our work began by researching and obtaining the original ThreadScan implementation from GitHub (ThreadScan Repository). After cloning the repository, we invested time in understanding the overall structure of the code, examining how different components interacted, and identifying key functions responsible for memory reclamation.

We proceeded to implement a benchmark inspired by the parameters outlined in the original research. In our tests, the update ratio was set at 20% (meaning roughly 10% of all operations were node removals). For instance, the benchmark was configured to test a data structure with 1024/131,072 nodes and a range of 2048/262,144 and an expected delete buffer size of 1024 nodes.

However, early in our experiments, we encountered persistent segmentation faults during the reclamation process—even when running the algorithm on a very simple data structure with a single thread. Despite various attempts to adjust parameters, modify node sizes, and even alter deletion methods, the problem remained unresolved. We even reached out to the original researcher for guidance but received no response.

Given the instability of the original ThreadScan implementation, we shifted our focus to the more recent ForkScan library (ForkScan Repository). ForkScan implements the same core idea—automated memory reclamation via OS signal-based scanning—but with fewer bugs and more robust support. With ForkScan, our tests proceeded as expected.

We observed that, within the `reclaim_iteration` function (located in `forkscan.c`), the algorithm calls `forkscan_util_sort` to order the pointers in the delete buffer before scanning other threads' stacks and registers. Re-sorting the buffer for every reclamation pass adds significant latency, especially under heavy load or when the delete buffer contains a large number of nodes.

To address this, we replace the current sorting mechanism with a self-balancing AVL tree. Instead of performing a full sort each time, the delete buffer would be maintained as an AVL tree, which dynamically preserves a sorted order.

An AVL tree guarantees a predictable $O(n \log n)$ time complexity, providing more consistent performance compared to sorting methods like quick sort or insertion sort, whose efficiency may vary based on the input ($O(n^2)$ in the worst case).

It is important to note that in our current implementation we build the AVL tree for each reclamation pass and then use it as is—without incrementally updating it between passes. However, for future implementations, we envision a more advanced solution: a persistent, global delete buffer maintained as an AVL tree. Such a structure would be created during the first reclamation process and then updated incrementally, eliminating the need to rebuild the tree from scratch at every iteration. This approach would be especially useful when threads share multiple nodes or pointers, as it would significantly speed up the reclamation process by avoiding repetitive sorting overhead.

Graphs

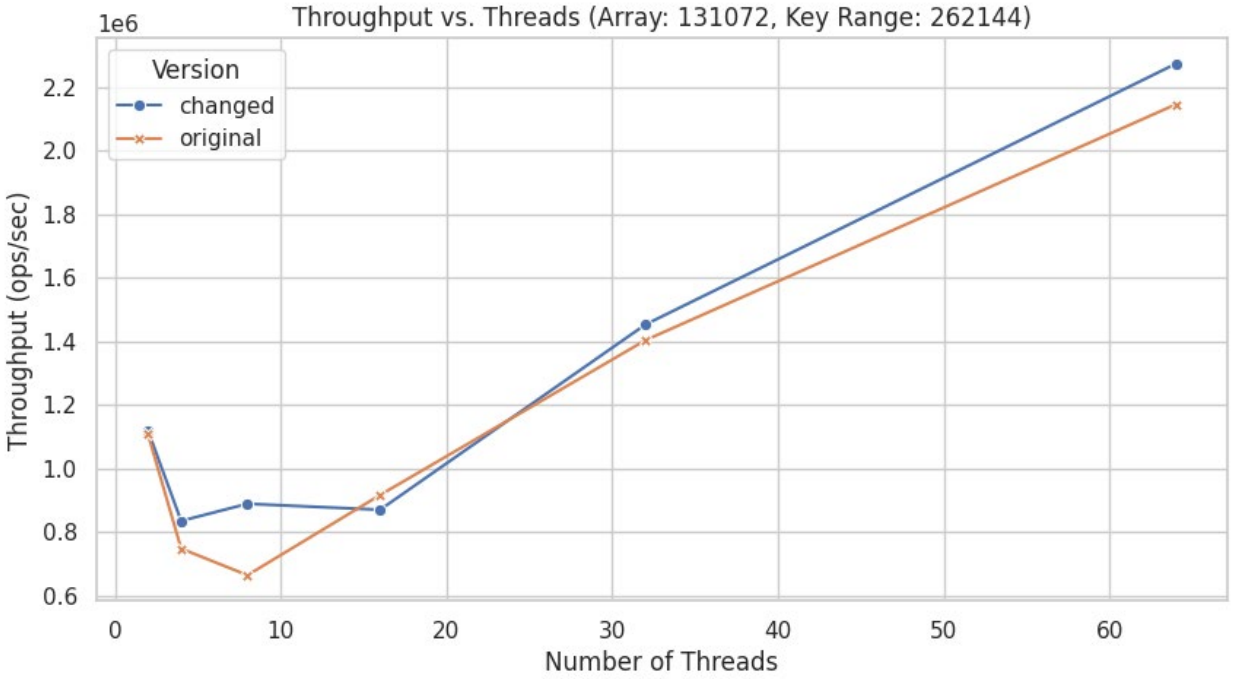
Original vs Changed

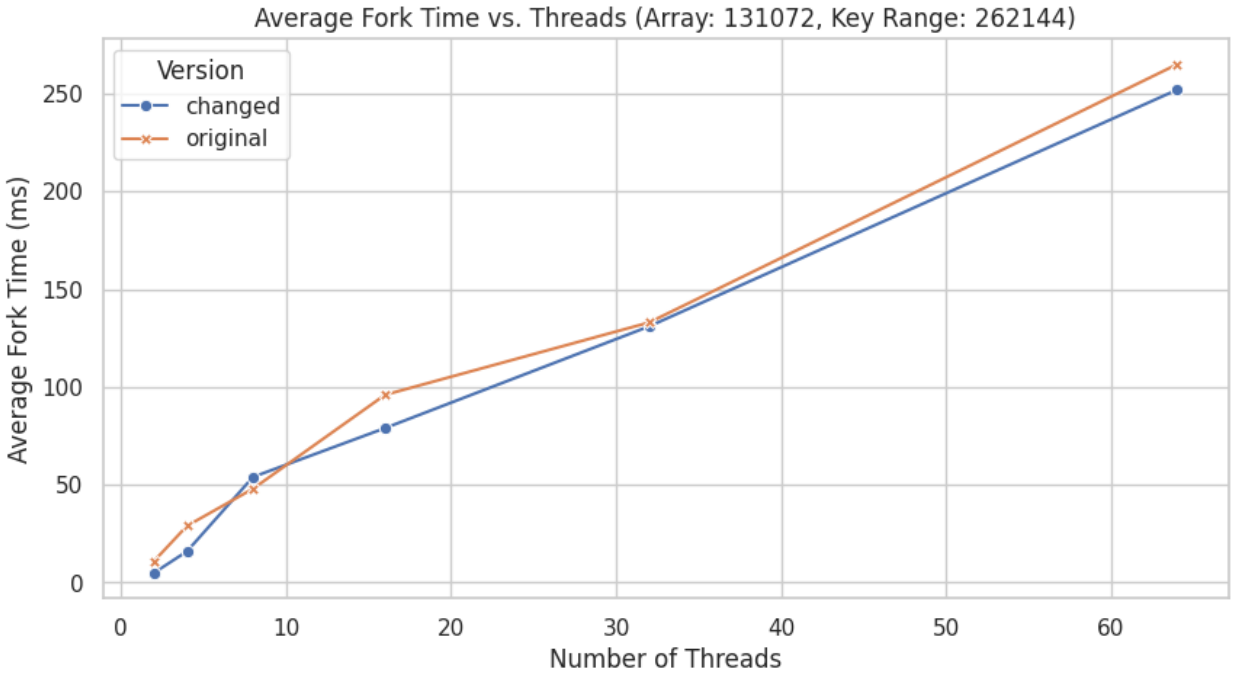
Throughput - The number of operations completed per second.

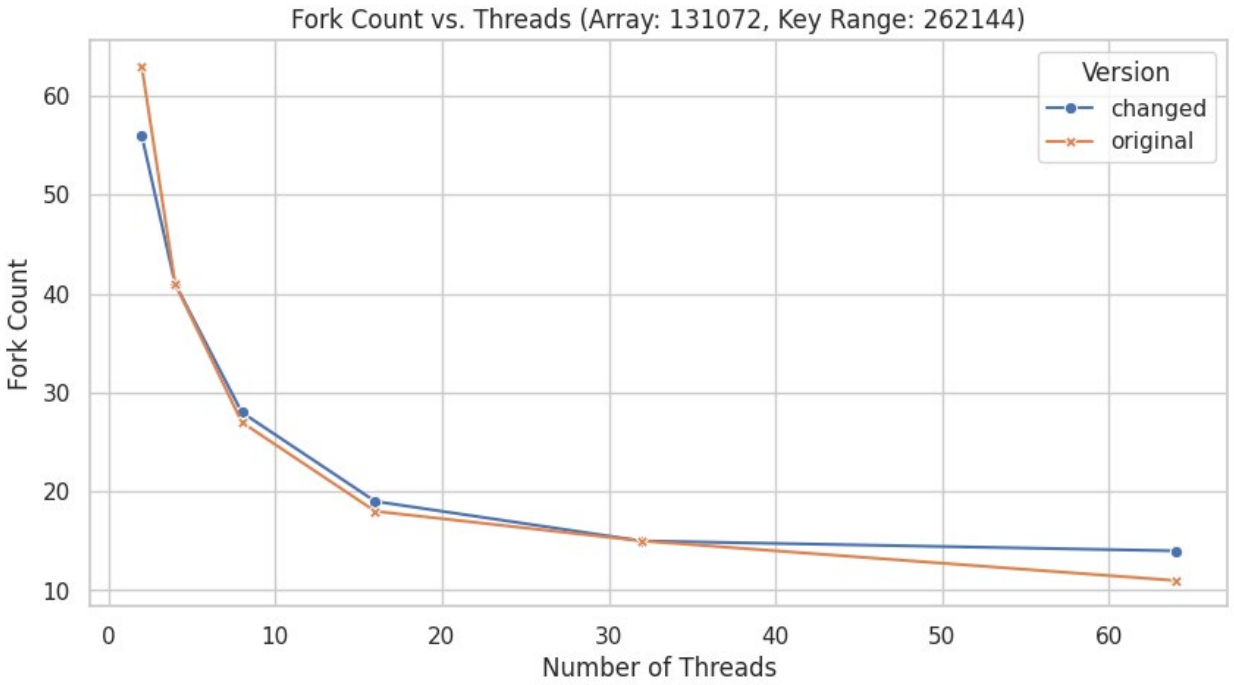
Average Fork Time - The average time spent during the memory reclamation process.

Fork Count - The total number of fork operations triggered during the run. It provides insight into how frequently the reclamation mechanism was activated.


Throughput vs. Threads (Array: 1024, Key Range: 2048)
1e7
Throughput (ops/sec)
Number of Threads
Version
changed
original
1.2
1.1
1.0
0.9
0.8
0.7
0
10
20
30
40
50
60


Throughput vs. Threads (Array: 131072, Key Range: 262144)
1e6
Throughput (ops/sec)
Number of Threads
Version
changed
original
2.2
2.0
1.8
1.6
1.4
1.2
1.0
0.8
0.6
0
10
20
30
40
50
60





For small data structures (1024 elements, key range 2048), the throughput of the changed version is comparable to the original, with only minor differences across thread counts. However, for larger data structures (131072 elements, key range 262144), the modified implementation—using an AVL tree to build the delete buffer—demonstrates small improvements.

References

1. Michael, M. M. (2004). *Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects*. IEEE Transactions on Parallel and Distributed Systems, 15(6), 491–504.
2. Alistarh, D., Leiserson, W. M., Matveev, A., & Shavit, N. (2015). *ThreadScan: Automatic and Scalable Memory Reclamation*. In Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2015).
3. Dan Alistarh, William Leiserson, Alexander Matveev, and Nir Shavit. 2017. *Forkscan: Conservative Memory Reclamation for Modern Operating Systems*. In Proceedings of the 12th European Conference on Computer Systems (Belgrade, Serbia) (EuroSys '17).