Optimizing Lock-Free Containers for Multithreaded Socially Oriented Information Systems*

Sergiy Yakovlev[1,2,†], Andrii Strelchenko[3,*,†], Artem Khovrat[3,*,†] and Volodymyr Kobziev[3,†]

[1] *Lodz University of Technology, 90-924 Lodz, Poland*
[2] *V.N. Karazin Kharkiv National University, 4, Svobody, Sq., Kharkiv, 61022, Ukraine*
[3] *Kharkiv National University of Radio Electronics, 14, Nauky, Ave., Kharkiv, 61166, Ukraine*

Abstract

This research examines lock-free containers designed for computationally intensive multithreaded intelligent solutions in socially oriented information systems, particularly financial platforms and social networks during periods of peak demand. Effective handling of large, rapidly changing data streams necessitates efficient buffering mechanisms. An expert evaluation identified queues as the optimal data structure for such sequential data processing tasks due to their inherent structural suitability. The study critically evaluated traditional lock-based synchronization methods commonly used in multi-threaded environments, uncovering significant drawbacks, including susceptibility to deadlocks, priority inversion, increased latency, and poor scalability. Given these limitations, the investigation pivoted towards lock-free synchronization methods, leveraging hardware-supported atomic operations and Compare-And-Swap (CAS) loops to facilitate concurrency without explicit locking mechanisms. To further optimize performance, memory locality principles were applied to lock-free queue implementations. Techniques such as strategic memory alignment, padding, and sequence numbering were introduced, significantly reducing cache misses and improving its efficiency. These enhancements aimed to minimize synchronization overhead, thus substantially increasing throughput and scalability under high contention scenarios. A rigorous benchmarking methodology was developed to evaluate the effectiveness of these optimizations, explicitly addressing multi-threaded measurement accuracy and correctness testing. Three distinct queue implementations were tested: a standard baseline lock-free queue, a volatile-based lock-free queue incorporating memory locality optimizations, and an atomic-based variant similarly optimized. Experimental results clearly indicated the volatile-based optimized queue significantly outperformed other implementations. It demonstrated notably lower latency, decreased performance variability, and superior scalability, underscoring the effectiveness of memory locality optimizations. These findings provide critical insights for developing efficient, scalable, and reliable synchronization solutions essential for contemporary high-load computing environments.

Keywords

cache optimization, computational intelligence, concurrent processing, high-loaded systems

1. Introduction

The rapid pace of globalization and the increasing digitalization of society have significantly amplified the importance of intelligent high-performance socially oriented systems. These systems, including social networks, financial platforms, and critical infrastructure applications, must effectively handle substantial and variable data streams. Furthermore, the resilience and responsiveness of these systems become particularly crucial during social crises, such as pandemics or military conflicts, when informational demands peak sharply.

International Workshop on Computational Intelligence, co-located with the IV International Scientific Symposium "Intelligent Solutions" (IntSol-2025), May 01-05, 2025, Kyiv-Uzhhorod, Ukraine
[1,*] Corresponding author.
[†] These authors contributed equally.
sergiy.yakovlev@p.lodz.pl (S. Yakovlev); andrii.strelchenko@nure.ua (A. Strelchenko); artem.khovrat@nure.ua (A. Khovrat); volodymyr.kobziev@nure.ua (V. Kobziev)
0000-0003-1707-843X (S. Yakovlev); 0009-0009-8332-4353 (A. Strelchenko); 0000-0002-1753-8929 (A. Khovrat); 0000-0002-8303-1595 (V. Kobziev)

A critical challenge faced by these systems is the management of high-load scenarios characterized by extensive concurrent data processing demands. The inability to manage such loads effectively results in back pressure, a phenomenon where data processing throughput cannot match incoming data rates, leading to potential system delays and reliability issues. Therefore, addressing high-load data processing requirements becomes vital for maintaining operational efficiency and data integrity within socially critical applications.

Traditionally, intelligent high-load systems have employed lock-based synchronization approaches, extensively researched and widely implemented due to their straightforward semantics and ease of use. However, lock-based mechanisms are inherently limited by issues such as deadlocks, priority inversion, increased latency, and poor scalability under high concurrency conditions. Conversely, lock-free synchronization strategies, which avoid explicit locking mechanisms by leveraging atomic hardware operations, remain relatively underexplored in the context of socially oriented high-load systems, despite their potential to offer significant performance improvements.

Given the aforementioned context, this study aims to comprehensively investigate the applicability and optimization potential of lock-free queue implementations specifically tailored for high-load socially oriented systems [1]. The primary objective is to evaluate and enhance lock-free queue structures through strategic memory locality optimizations, ultimately aiming to achieve maximal performance efficiency and data integrity. To achieve this goal, the following research tasks were formulated:

- Identification and detailed characterization of the back pressure phenomenon in high-load systems.
- Analysis and comparison of existing synchronization mechanisms (lock-based vs. lock-free) regarding their performance and limitations.
- Exploration and implementation of memory locality optimization techniques within lock-free data structures.
- Comprehensive benchmarking and correctness testing of optimized lock-free implementations under realistic high-load scenarios.

Importantly, the intent of this research is not to target specific throughput metrics; instead, the goal is to develop a lock-free queue optimized to deliver maximal efficiency within clearly defined operational constraints. Special emphasis is placed on guaranteeing robust data transfer, maintaining sequence integrity, and reducing synchronization overhead. This analysis will critically assess optimization strategies tailored to high-throughput data processing and evaluate the contribution of lock-free data structures toward enhancing performance in relevant high-load environments.

2. System description

Before proceeding with further analysis, it is critical to establish a clear understanding of high-load system characteristics, as their operational specifics significantly influence the selection and efficiency of buffering mechanisms and subsequent data handling methodologies. High-load environments often encounter substantial and unpredictable streams of data, necessitating specialized strategies to ensure stable, responsive, and reliable performance.

2.1. Back pressure problem

A primary challenge encountered by high-load systems is "back pressure." This phenomenon arises when consumer threads or processes fail to match the pace at which data producers generate information. Consequently, unprocessed data accumulates, potentially leading to significant delays or even system failures. Effectively managing back pressure is crucial, particularly in socially

oriented and critical infrastructure systems where data integrity, timeliness, and consistent throughput are non-negotiable requirements.

Typical software architectures address back pressure through several approaches:

- Producer Regulation: Slowing down data generation intentionally at the source. This is practical primarily in scenarios involving user interaction but is ineffective for systems reliant on automated, continuous data streams.
- Data Dropping: Discarding excess incoming data. Suitable for applications where minor data loss does not substantially degrade overall functionality, yet inappropriate for critical systems requiring strict data continuity.
- Data Buffering: Implementing intermediate buffers that temporarily store excess incoming data, smoothing out transient data surges and preventing overload scenarios. This solution is optimal for environments prioritizing data consistency and integrity [2].

2.2. Buffering received data

Given the limitations of producer regulation and data dropping in critical system scenarios, buffering received data (BRD) emerges as the most viable solution for managing back pressure effectively. As depicted in Figure 1, BRD employs a dedicated producer interface thread responsible for continuously retrieving and aggregating incoming data into a Shared Container. Simultaneously, multiple consumer threads independently access and process data from this shared container, distributing processing load effectively.

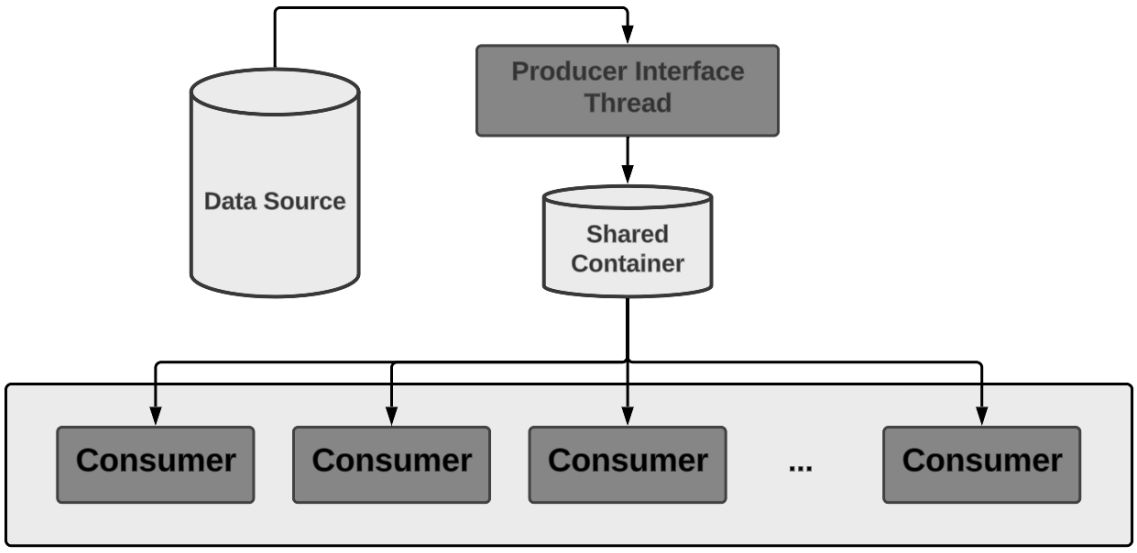

Figure 1: Scheme of buffering received data [created by the authors].

This strategy allows for substantial scalability. Theoretically, the number of consumer threads can increase indefinitely, provided sufficient hardware resources are available. Practically, however, hardware constraints impose limits on the number of simultaneous threads, and scaling beyond certain thresholds may result in diminishing returns or increased complexity related to synchronization and thread management.

2.3. Buffering received data with memory pool

A further optimization of the basic BRD method includes integrating an external memory pool into the buffering architecture, as illustrated in Figure 2. By employing a memory pool, operations are performed on memory pointers rather than directly on buffered data. This approach significantly

reduces container access bottlenecks and improves the overall efficiency and responsiveness of the data handling process [3, 4].

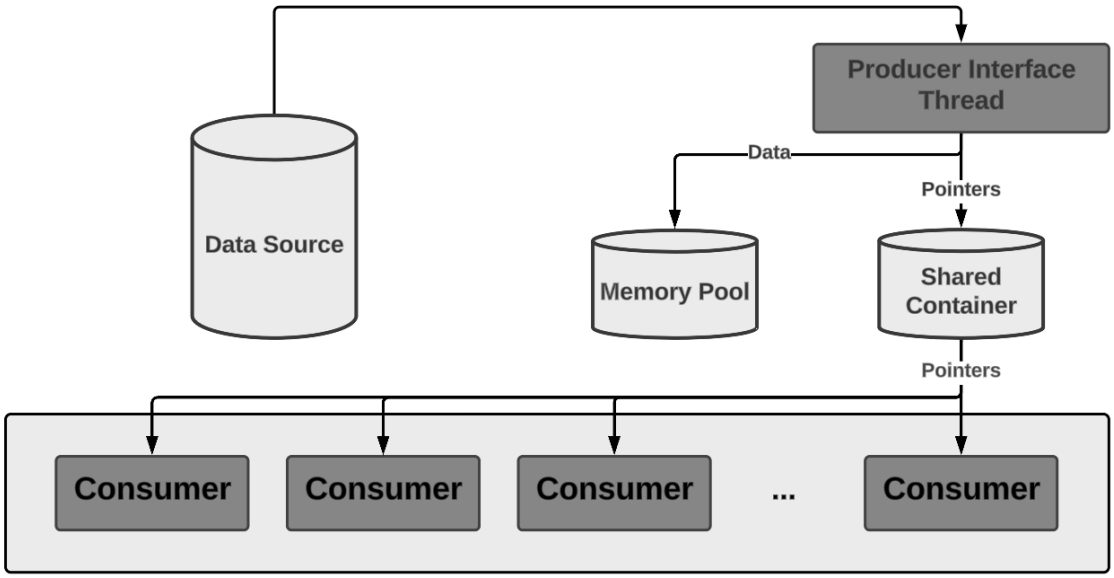

Figure 2: Scheme of buffering received data with memory pool [created by the authors].

Adopting this methodology necessitates clearly defining data sizes provided by producer threads, ensuring consumer threads effectively retrieve and manage data via pointer-based access. Common implementations utilize circular buffers with fixed capacities, overwriting old data with new entries without explicit clearing operations. Despite potential security considerations inherent in this approach, proper data management practices effectively mitigate these risks.

3. Buffer implementation

For effectively managing buffered data in high-load systems, an appropriate data structure must be selected based on its efficiency in handling concurrent data access and processing.

3.1. Container selection

To determine the most suitable container type, an expert assessment was conducted involving 100 specialists from various cities, including Kharkiv, Kyiv, Lviv, Vienna, Lisbon, Krakow, Dnipro, Odesa, New York, Toronto, and Tbilisi. The survey aimed to identify the most frequently utilized data structures in data flow scenarios. The majority of experts identified queues and stacks as the most common, each receiving maximum support (100 votes).

Considering the characteristics of typical data handling scenarios encountered in high-load systems – particularly where orderly data processing is critical – the queue was selected as the container type for further investigation. The fundamental principle guiding queue functionality is "First In, First Out" (FIFO), where data items are inserted at one end (tail) and retrieved from the opposite end (head). Common queue implementations include:

- Linked Lists: Consistently linked nodes that allow constant-time operations for data insertion and removal.
- Dynamic Arrays: Arrays expandable at both ends, requiring memory reallocation during extension.
- Hybrid Models: Combining linked lists and dynamic arrays, typically in the form of fixed-size array buckets that dynamically expand as needed.

Due to their structured approach to data management and inherent efficiency in handling sequential processing tasks, queues are widely utilized in information processing systems. Two fundamental approaches exist for queue implementation in multi-threaded systems: lock-based and lock-free paradigms.

3.2. Lock-based approach

The lock-based approach involves securing exclusive access to critical code sections, thereby preventing concurrent thread interference. This approach typically employs synchronization mechanisms such as mutexes or locks. A representative scenario is illustrated in Figure 3, demonstrating protection of a shared resource – such as a global counter – through mutex locking. The process involves explicitly acquiring a lock before performing the operation and releasing it afterward, ensuring data consistency at the cost of potential thread waiting times.

```
1: declare counter ← 0
2: declare counter_mutex as Mutex
3:
4: function increment_with_lock()
5:     LOCK(counter_mutex)
6:     counter ← counter + 1
7:     UNLOCK(counter_mutex)
8: end function
```

Figure 3: Increment with lock [created by the authors].

Another generalized implementation of this approach is shown in Figure 4, highlighting a lock-based queue implementation. Here, node creation and insertion operations are explicitly encapsulated as critical sections. Each thread must obtain exclusive access before modifying shared structures, thereby ensuring data integrity and preventing race conditions.

Despite its straightforward implementation and clear synchronization semantics, the lock-based method possesses several inherent drawbacks, including:

- Deadlocks and livelocks: Incorrect handling of locks can lead to situations where threads become indefinitely blocked, effectively halting system functionality.
- Latency issues: Threads may experience significant delays as they wait for locked resources to become available, resulting in reduced overall system responsiveness.
- Poor scalability: As the number of threads increases, contention for locks escalates, thereby degrading performance significantly in high-load environments.

Additionally, the lock-based approach can introduce priority inversion issues, where lower-priority threads hold locks required by higher-priority threads, causing increased waiting times and reduced predictability in real-time systems. The complexity of managing multiple locks can also lead to higher chances of programmer errors, making system maintenance and debugging more challenging.

```
1: class Queue:
2:     declare front as Pointer to Node
3:     declare rear as Pointer to Node
4:     declare mutex as Mutex
5:
6: function ENQUEUE(obj as Data)
7:     LOCK(mutex)
8:     new_node ← CREATE_NODE(obj)
9:     if rear = NULL then
10:        front ← new_node
11:        rear ← new_node
12:    else
13:        rear.next ← new_node
14:        rear ← new_node
15:    end if
16:    UNLOCK(mutex)
17:    return
18: end function
19:
20: function DEQUEUE()(returns DataPointer)
21:    LOCK(mutex)
22:    if front = NULL then
23:        UNLOCK(mutex)
24:        return NULL
25:    end if
26:    ret_ptr ← front
27:    front ← front.next
28:    if front = NULL then
29:        rear ← NULL
30:    end if
31:    UNLOCK(mutex)
32:    return ret_ptr.data
33: end function
```

Figure 4: Thread-safe queue using mutex [created by the authors].

3.3. Lock-Free approach

To mitigate the limitations associated with lock-based methods, lock-free techniques employ atomic operations provided by hardware, ensuring that data modifications occur indivisibly without being visible in intermediate states. A fundamental concept in lock-free programming is the Compare-And-Swap (CAS) operation [5], illustrated in Figure 5. CAS loops attempt atomic updates to shared variables, repeatedly verifying that the expected state remains consistent before applying modifications.

```
1: declare counter AS AtomicInteger ← 0
2:
3: function increment_lockfree()
4:     counter.atomic_increment()
5: end function
```

Figure 5: Lock-free increment [created by the authors].

Algorithm presented in Figure 6 is a basic example of a lock-free queue implementation using CAS operations. Unlike lock-based approaches, this method significantly reduces thread wait times and eliminates deadlock conditions by continuously retrying operations without explicit locking mechanisms.

Moreover, lock-free approaches inherently provide better adaptability to varying workloads. Since threads do not wait on locks, they can immediately attempt retries after a failure, allowing them to dynamically respond to system load fluctuations. This characteristic substantially contributes to achieving improved throughput, especially in high-contention environments typical

of high-load applications. Another significant advantage of lock-free methodologies is the inherent robustness against thread failures. Unlike lock-based mechanisms, where a thread failure while holding a lock could stall or degrade the entire system's operation, lock-free techniques ensure that individual thread failures do not adversely impact overall system performance [6]. This feature is particularly beneficial in critical real-time applications where uninterrupted service is essential.

Nevertheless, lock-free programming introduces unique challenges, notably the ABA problem, wherein memory states can revert to previously observed conditions undetected, causing erroneous system behavior. Although these challenges require sophisticated handling techniques, lock-free implementations generally provide superior scalability and responsiveness compared to lock-based methods.

```
1:  class Queue:
2:      declare head as AtomicPointer to Node
3:      declare tail as AtomicPointer to Node
4:
5:  function ENQUEUE(obj_ptr as ObjPointer)
6:      while true do
7:          tail_ptr ← tail
8:          next_ptr ← tail_ptr.next
9:          if tail_ptr = tail then
10:             if next_ptr = NULL then
11:                 if compare_and_swap(tail_ptr.next, NULL, obj_ptr) then
12:                     compare_and_swap(tail, tail_ptr, obj_ptr)
13:                     return
14:                 end if
15:             else
16:                 compare_and_swap(tail, tail_ptr, next_ptr)
17:             end if
18:         end if
19:     end while
20: end function
21:
22: function DEQUEUE()(returns ObjPointer)
23:     while true do
24:         head_ptr ← head
25:         tail_ptr ← tail
26:         next_ptr ← head_ptr.next
27:         if head_ptr = head then
28:             if head_ptr = tail_ptr then
29:                 if next_ptr = NULL then
30:                     return NULL
31:                 end if
32:                 compare_and_swap(tail, tail_ptr, next_ptr)
33:             else
34:                 obj ← next_ptr.data
35:                 if compare_and_swap(head, head_ptr, next_ptr) then
36:                     return obj
37:                 end if
38:             end if
39:         end if
40:     end while
41: end function
```

Figure 6: Lock-free thread safe queue [created by the authors].

The recognition of these nuanced challenges has encouraged further advancements, particularly incorporating memory locality optimization into lock-free techniques. Subsequent sections will thoroughly examine such advanced lock-free variations, emphasizing the critical role of memory locality in enhancing performance in high-load, multi-threaded environments.

4. Optimization of lock-free approach

4.1. Memory locality challenges in lock-free implementations

Additional Optimizing memory access patterns is critical to improving the performance of lock-free algorithms in high-load, multi-threaded systems. Two core principles drive cache efficiency and memory optimization [7, 8]:

- Temporal Locality: This principle implies that recently accessed data will likely be accessed again shortly. Effective utilization of temporal locality requires ensuring frequently accessed data remains available within fast cache memory, thus minimizing repetitive and costly accesses to slower main memory.
- Spatial Locality: Spatial locality refers to the tendency for a program to access data locations near previously accessed locations. Optimally leveraging spatial locality involves organizing data so that related data elements are stored within the same cache lines, thereby significantly reducing cache misses and enhancing overall processing speed [9].

Lock-free data structures frequently utilize linked list architectures due to their inherent flexibility. However, linked lists pose specific challenges for memory locality optimization. While they inherently offer some spatial locality advantages when nodes are allocated contiguously, practical constraints typically require separation of frequently updated components, such as queue head and tail pointers, to different cache lines. This separation mitigates cache contention and the false sharing phenomenon—an issue where multiple threads inadvertently cause cache invalidations due to modifications of adjacent memory locations.

To address false sharing, implementations commonly employ padding techniques, explicitly isolating frequently modified data by placing them on distinct cache lines.

4.2. Enhancing lock-free algorithms with sequence numbers

An advanced optimization strategy involves incorporating sequence numbers into the node structure of lock-free queues. Sequence numbers function as synchronization tools, significantly reducing unnecessary cache coherence traffic among threads. Proper alignment of node data and sequence numbers ensures operations by different threads occur on separate cache lines, improving parallel execution efficiency [10].

The key advantages of integrating padding and sequence numbers into lock-free structures include [11]:

- Prevention of false sharing by isolating updates to different memory regions.
- Enhanced cache efficiency through optimal data alignment.
- Improved scalability and performance in high-contention, multi-threaded scenarios due to reduced synchronization overhead.

Example code illustrating this optimization shown in Figure 7.

```
struct QueueNode:
    align(256) declare sequence as atomic_integer
    declare data as data_type

function ENQUEUE(data)
    declare position as integer
    declare index as integer
    declare node_sequence as integer
    position ← write_position.load()
    while true do
        index ← position & mask
        node_sequence ← queue[index].sequence.load()
        if node_sequence ¡ position then
            return false                                    ▷ Queue overflow
        end if
        if write_position.compare_and_swap(position, position + 1) then
            queue[index].data ← data
            queue[index].sequence.store(node_sequence + 1)
            return true
        end if
        position ← write_position.load()
    end while
end function
```

Figure 7: Enhanced version of lock-free approach [created by the authors].

By effectively employing padding and sequence numbers, lock-free algorithms can achieve significant performance improvements through optimized memory locality strategies. With these foundational concepts established, we can now proceed to a comprehensive evaluation of these strategies. Before conducting performance evaluations, clearly defining benchmarking principles to accurately measure and analyze algorithm efficiency under realistic high-load conditions is essential [11].

5. Benchmarking

5.1. Methodology and challenges

Accurately measuring performance metrics represents one of the critical challenges when evaluating multi-threaded applications, particularly those utilizing lock-free structures. A conventional approach for benchmarking performance involves capturing the execution time of a particular operation or set of operations. Typically, this approach can be outlined as shown in Figure 8:

```
function MEASURETIME
    declare start_time as timestamp
    declare end_time as timestamp
    declare elapsed_time as duration
    start_time ← getCurrentTime()        ▷ Capture the starting timestamp
    performTask()                ▷ Execute the operation being benchmarked
    end_time ← getCurrentTime()          ▷ Capture the ending timestamp
    elapsed_time ← end_time − start_time          ▷ Compute elapsed time
    return elapsed_time
end function
```

Figure 8: Typical benchmarking procedure [created by the authors].

While effective in single-threaded scenarios, this straightforward measurement approach becomes unreliable in multi-threaded contexts due to potential interference from concurrent operations and system scheduling.

Reliable performance measurement in multi-threaded scenarios necessitates addressing concurrent execution challenges and operating system scheduling interference [12]. Techniques such as thread affinity or "thread pinning" are employed to minimize inaccuracies by binding individual threads to specific CPU cores, thereby preventing context switches and ensuring consistent execution contexts. Furthermore, modern computing systems exhibit nondeterministic behavior caused by scheduling and resource contention, complicating reliable performance measurements. Thus, isolating benchmark tests from external system processes is essential to achieving accurate, reproducible results. "Shielding," a strategy involving reserving specific CPU cores exclusively for benchmark threads, significantly reduces variability in measurements resulting from external interference.

5.2. Producer-consumer model

Benchmarking producer-consumer scenarios, particularly those with multiple producers and consumers, introduces additional complexities. To ensure accurate and meaningful benchmarking results, the following considerations are essential:

- Minimizing interference from unrelated system processes.
- Consistent core allocation for benchmark threads.
- Capturing synchronization overhead, as contention among threads significantly affects performance.

For a robust evaluation of performance in real-world settings, the benchmarking framework should record comprehensive statistical data, including minimum and maximum latencies, standard deviation, and percentile thresholds (such as the 95th and 99th percentiles). This detailed statistical approach provides a deeper understanding of system behavior, particularly under worst-case conditions and peak load scenarios. Additionally, integer-based data types are commonly utilized in benchmarks due to their minimal memory overhead and simplicity, mimicking pointer operations effectively. External memory allocation strategies, such as memory arenas, offer significant performance benefits and simplify data management, further enhancing benchmarking reliability.

Correctness testing is equally critical to maintaining data integrity in concurrent data structures. Verifying the correctness typically involves validating the order and accuracy of processed elements. For instance (see Figure 9), one-to-one (1:1) producer-consumer tests involve a producer sequentially inserting values into a queue and a consumer subsequently retrieving and verifying the integrity of these values.

```
1: function PRODUCER
2:     for i = 0 to batchSize - 1 do
3:         queue.enqueue(i)
4:     end for
5: end function
6:
7: function CONSUMER
8:     for i = 0 to batchSize - 1 do
9:         declare value as integer
10:        value ← queue.dequeue()
11:        if value ≠ i then
12:            print("Error: Expected", i, "but got", value)
13:        end if
14:    end for
15: end function
```

Figure 9: One-to-one correctness test [created by the authors].

Two-to-two (2:2) correctness tests (see Figure 10) further enhance validation, where two producers concurrently insert data, and two consumers verify the processed data. Such tests help

identify race conditions, synchronization issues, and unexpected behaviors emerging in concurrent execution environments.

```
1: function PRODUCER(id)
2:     for i = 0 to batchSize - 1 do
3:         queue.enqueue(i)
4:     end for
5: end function
6:
7: function CONSUMER
8:     for i = 0 to batchSize × 2 - 1 do
9:         declare value as integer
10:        value ← queue.dequeue()
11:        counts[value] ← counts[value] + 1
12:    end for
13:    for i = 0 to batchSize - 1 do
14:        if counts[i] ≠ 2 then
15:            print("Error: Expected 2 for value", i, "but got", counts[i])
16:        end if
17:    end for
18: end function
```

Figure 10: Two-to-two correctness test [created by the authors].

6. Evaluation results

To thoroughly assess the effectiveness of the lock-free queue implementations optimized for memory locality, extensive experiments were conducted. The evaluation focused on insertion (enqueue) and retrieval (dequeue) operations within queues, covering internal object sizes ranging from 1 byte to 128 bytes. The chosen size increments followed powers of two, aligning with common cache line sizes in processor architectures, specifically targeting ARM and comparable systems [13].

Experiment results are illustrated for Dequeue and Enqueue on Figure 11 and Figure 12, respectively.

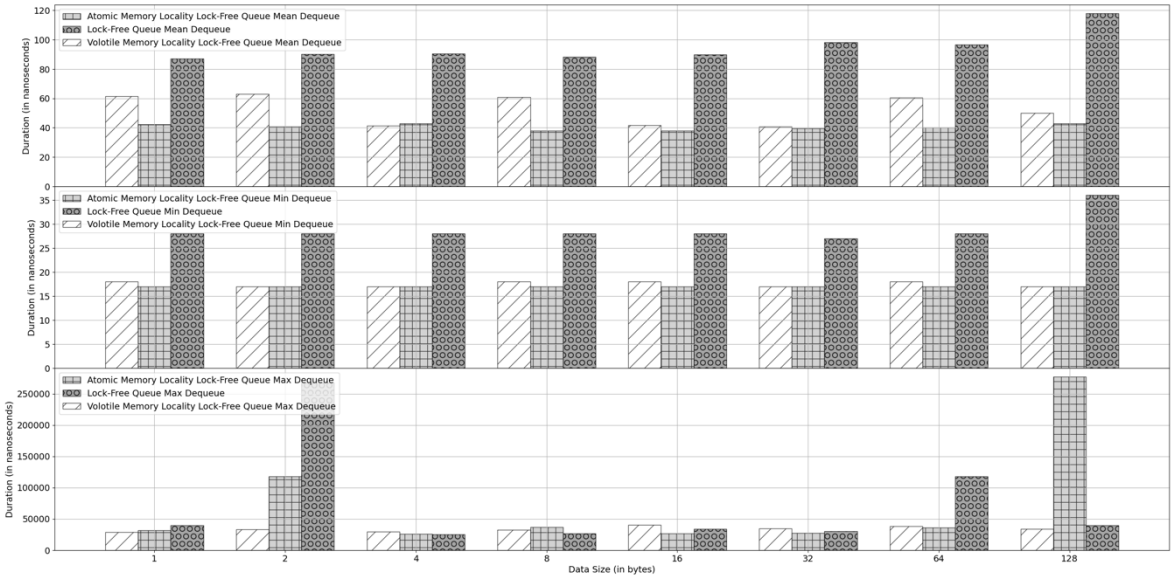

Figure 11: Experimental Results for Dequeue Operations [created by the authors].

Figures 11 and Figure 12 present the performance measurements for the dequeue and enqueue operations, respectively, under the 2:2 producer-consumer configuration. Due to the structural similarities between the two operations, both figures exhibit comparable performance trends.

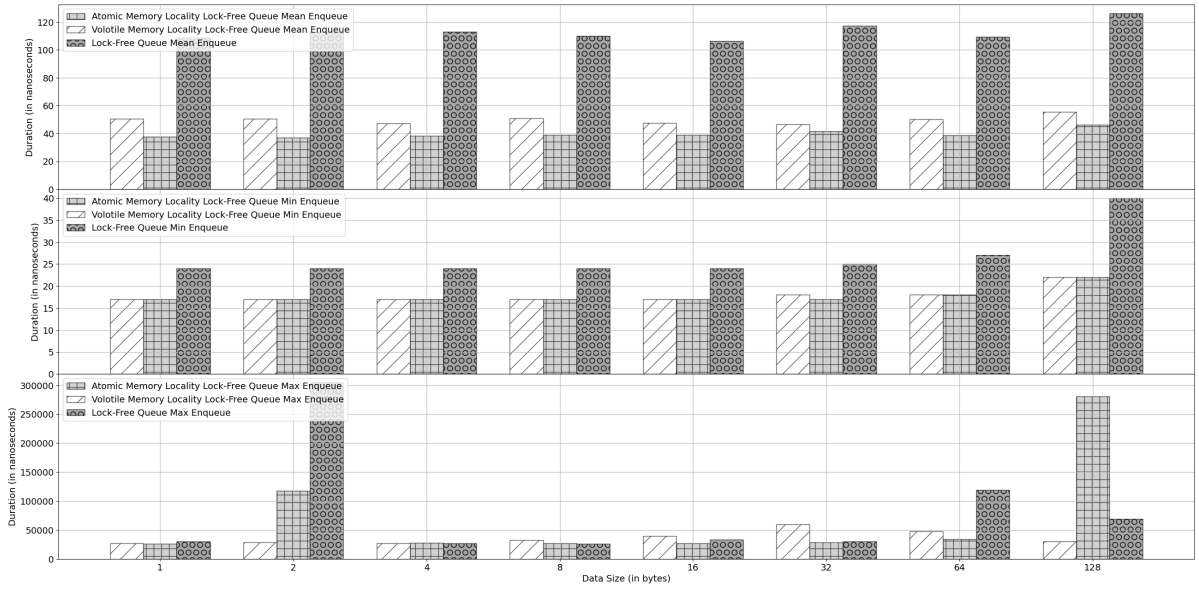

Figure 12: Experimental Results for Enqueue Operations [created by the authors].

Key findings from the experimental data include:

- The distribution of minimum execution times across varying data sizes remained consistently stable for each tested queue implementation, exhibiting only slight performance degradation at the largest size (128 bytes).
- Notable peaks in maximum latency were observed for object sizes of 2 bytes (across all queue implementations) and at 128 bytes specifically within atomic-based implementations.

The performance degradation at the 2-byte size is attributed to suboptimal processor-level handling, as this size does not align naturally with typical processor architectures [15]. The latency increase observed at 128 bytes corresponds to the tested processor's actual cache line size (64 bytes). While volatile-based and traditional lock-free implementations appeared unaffected by this discrepancy, atomic-based implementations were significantly impacted due to additional atomic operation overhead. The experimental outcomes underline the considerable advantage of memory locality optimizations in lock-free approaches. Specifically, improved data placement within memory significantly enhances access efficiency, thereby directly influencing the fundamental performance characteristics of lock-free algorithms [14]. The subsequent section will offer a comprehensive discussion and interpretation of these performance results, considering their implications for practical high-load system scenarios.

Aside from synthetic experiments, this optimization approach was also applied to a system performing statistical analysis of market data from an automated electronic exchange. The results demonstrated a significant improvement in system stability, particularly in addressing the backpressure problem.

7. Conclusion

This research conducted a comprehensive analysis and evaluation of lock-free queue implementations, emphasizing memory locality optimization strategies. Extensive experimental studies were carried out, assessing performance across various internal queue object sizes, ranging from 1 to 128 bytes. Three distinct implementations were systematically compared: a conventional lock-free queue serving as a baseline, a volatile-based approach incorporating memory locality optimizations, and an atomic-based implementation similarly enhanced by locality techniques.

The principal findings indicate that while conventional lock-free queues maintain theoretical non-blocking properties, they demonstrate significant performance variability under high

contention due to inherent cache coherence overhead and frequent synchronization needs. Conversely, implementations optimized for memory locality – particularly those leveraging volatile variables – exhibited marked performance improvements under intensive concurrent conditions, attributed to reduced memory barriers and enhanced cache efficiency. However, atomic-based queues, despite benefiting from structured memory alignment strategies, introduced additional overhead linked to atomic synchronization operations, thus occasionally diminishing performance compared to their volatile counterparts. These experimental outcomes underscore the critical role of memory locality in lock-free queue optimization and highlight the delicate balance required between cache optimization and synchronization overhead.

Additionally, the study explored broader aspects relevant to practical deployment, including expert evaluations identifying queue data structures as optimal for sequential data processing in high-load systems. Benchmarking and correctness testing methodologies were rigorously defined and applied, ensuring comprehensive assessment of performance and data integrity under realistic multi-threaded scenarios.

In conclusion, memory locality optimizations substantially enhance the scalability and performance of lock-free queues. Further investigations should focus on developing hybrid synchronization mechanisms dynamically adaptable to varying levels of contention, thereby further enhancing scalability and efficiency in real-world, high-load, multi-threaded environments.

Acknowledgements

The authors would like to thank the Armed Forces of Ukraine for the opportunity to write a valid work during the full-scale invasion of the Russian Federation on the territory of Ukraine. Also, the authors wish to extend their gratitude to Kharkiv National University of Radio Electronics for providing licences for additional software to prepare algorithms and the paper.

Declaration on Generative AI

During the preparation of this work, the authors used Grammarly Edu and submodule of Microsoft 365 in order to check grammar and spelling. After using these services, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] S. Yakovlev, A. Khovrat, V. Kobziev, D. Uzlov, Decision support algorithm in the development of information sensitive socially oriented systems. in: Proceedings of the 4th International Workshop of IT-professionals on Artificial Intelligence, Cambridge, USA, 25–27 September 2024, pp. 315–326.

[2] A. Honorat, M. Dardaillon, H. Miomandre, J.-F. Nezan, Automated Buffer Sizing of Dataflow Applications in a High-Level Synthesis Workflow. ACM Transactions on Reconfigurable Technology and Systems 17/1 (2024) pp. 1–26. doi: 10.1145/3626103.

[3] M. Pohnl, Shared-Memory-Based Lock-Free Queues: The Key to Fast and Robust Communication on Safety-Critical Edge Devices. in: Proceedings of Cyber-Physical Systems and Internet of Things Week, San Antonio, USA, 9–12 May 2023, pp. 179–184.

[4] R. Ando, Y. Kadobayashi, H. Takakura, Clustering Massive Packets using a Lock-Free Algorithm of Tree-Based Reduction on GPGPU. International Journal of Computer Science and Information Security 19/3 (2021) pp. 19–24.

[5] N. Ben-David, Lock-free locks revisited. in: Proceedings of Symposium on Principles and Practice of Parallel Programming, Seoul, Republic of Korea, 2–6 April 2022, pp. 278–293.

[6] B. Assiri, Lock-free Fill-in Queue. in: Proceedings of International Conference on Computer and Information Sciences, Sakaka, Saudi Arabia, 13–15 October 2022, pp. 1–6.

[7] D. Bakhvalov, Performance Analysis and Tuning on Modern CPUs, Easyperf, available at: https://faculty.cs.niu.edu/~winans/notes/patmc.pdf (last accessed 31.03.2025).

[8] P. Bilokon, B. Gunduz, C++ Design Patterns for Low-latency Applications Including High-frequency Trading, Arxiv, available at: https://arxiv.org/abs/2309.04259 (last accessed 31.03.2025).

[9] Y. Lin, W. Lin, J. Xu, Y. Chen, Z. Jin, J. Qin, J. He, S. Cai, Y. Zhang, Z. Wang, W. Chen, PARS: A Pattern-Aware Spatial Data Prefetcher Supporting Multiple Region Sizes. International IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 43/11 (2024) pp. 3638–3649. doi: 10.1109/TCAD.2024.3442981.

[10] Q. C. Liu, J. Shun, I. Zablotchi, Lock-free Fill-in Queue. in: Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, Edinburgh, UK, 2–5 March 2024, pp. 286–300. doi: 10.5281/zenodo.10253798.

[11] A. Blot, J. Petke, A Comprehensive Survey of Benchmarks for Improvement of Software's Non-Functional Properties. ACM Computing Surveys 57/7 (2025) pp. 1–36. doi: 10.1145/3711119.

[12] M. Herlihy, N. Shavit, The Art of Multiprocessor Programming, Morgan Kaufmann Publishers (2021), San Francisco, USA.

[13] Intel, Intel® Xeon® CPU Max Series Configuration and Tuning Guide, IDZ Technical Library, available at: https://www.intel.com/content/www/us/en/content-details/769060/intel-xeon-cpu-max-series-configuration-and-tuning-guide.html (last accessed 31.03.2025).

[14] K. Klenk, M. M. Moayeri, J. Guo, M. P. Clark, R. J. Spiteri, Mitigating synchronization bottlenecks in high-performance actor-model-based software. in: Workshops of the International Conference for High Performance Computing, Atlanta, USA, 17–22 November 2024, pp. 1274–1287. doi: 10.1109/SCW63240.2024.00168.

[15] P. Moreno, M. Areias, R. Rocha, On the implementation of memory reclamation methods in a lock-free hash trie design. Journal of Parallel and Distributed Computing 155 (2021) pp. 1–13. doi: 10.1016/j.jpdc.2021.04.007.