Dissertation for Achieving the Academic
Degree of Doktoringenieur (Dr.-Ing.)

High Performance and Dependable Asynchronous Communication on Multi-Core Systems

Jiawei Wang

Born on: April 4, 1995

Primary Supervisor
Prof. Dr. Hermann Härtig

Co-supervisor
Dr. Ming Fu

Fakultät Informatik, Institut für Systemarchitectur
Technische Universität Dresden
Dresden, Deutschland, 2024

Abstract

Asynchronous communication plays a crucial role in multi-threaded applications such as operating systems, databases, networks, and language runtimes, enabling data transfer, task distribution, and component decoupling. However, with the waning of Moore's Law and the rise of heterogeneous multi-core architectures, existing methods face challenges, including performance issues when dealing with cache and memory hierarchies and leveraging modern hardware instructions, as well as correctness issues due to the increasing complexity when coevolving with modern architectures (e.g., weak memory models).

This thesis investigates the problem of how to design and implement high-performance and dependable asynchronous communication components for multi-core systems and presents novel methods and algorithms to address these challenges. We have invented several queues, including block-based queue (BBQ), concurrent nested queue (CNQ), and block-based work-stealing queue (BWoS), which are verified and optimized using a model-checking-based framework. These queues have been successfully integrated into real-world applications, including DPDK, Linux IO_uring, Java GC, and Go and Rust runtimes. Our experiments demonstrate significant end-to-end performance improvement for industrial software over state-of-the-art approaches.

List of Publications

This thesis is based on the following publications:

- BWoS: Formally Verified Block-based Work Stealing for Parallel Processing. Jiawei Wang, Bohdan Trach, Ming Fu, Diogo Behrens, Jonathan Schwender, Yutao Liu, Jitang Lei, Viktor Vafeiadis, Hermann Härtig and Haibo Chen. In *proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation* (OSDI), pages 833-850, 2023.

- BBQ: A Block-based Bounded Queue for Exchanging Data and Profiling. Jiawei Wang, Diogo Behrens, Ming Fu, Lilith Oberhauser, Jonas Oberhauser, Jitang Lei, Geng Chen, Hermann Härtig, and Haibo Chen. In *2022 USENIX Annual Technical Conference* (USENIX ATC), pages 249-262, 2022.

Acknowledgment

I wish to extend my heartfelt gratitude to my supervisor, Professor Dr. Hermann Härtig, for his unwavering mentorship throughout my doctoral journey. Our discussions have been invaluable, and I have greatly benefited from his rigorous approach and steadfast passion for research. I am profoundly thankful to Dr. Ming Fu for his support and guidance. It was through his recommendation that I had the opportunity to pursue an industrial doctorate jointly supervised by Technische Universität Dresden and Huawei Dresden Research Center. I am grateful for his assistance during my doctoral studies, particularly the memorable discussions we had about paper writing during our walks along the Elbe River on numerous occasions. With his support, I was able to swiftly immerse myself in research activities and publish two research papers within 2 years. I extend my appreciation to my colleagues at Huawei Dresden Research Center for their invaluable assistance and camaraderie during my doctoral study. From collaborative problem-solving and manuscript revisions to setting up the experimental environments, their support has been indispensable. I am also thankful for their assistance in handling matters such as accommodation and enrollment when I first arrived in Germany. Special thanks are due to Professor Viktor at the Max Planck Institute and Professor Haibo Chen at Shanghai Jiao Tong University for their guidance in paper writing. I am grateful to my colleagues at Huawei headquarters for their support in commercializing my research findings. Lastly, I would like to express my deepest gratitude to my friends and family for their unwavering support and encouragement throughout my doctoral journey. Their moral support has been a constant source of strength for me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Multi-core Processors

In 1965, Gordon Moore introduced Moore's Law [156] based on empirical evidence, which characterizes the growth trend in both the number and performance of transistors within integrated circuits, particularly microprocessor chips. This law observes that the number of transistors approximately doubles every 18 to 24 months. Over the past several decades, Moore's Law has served as a guiding principle in the microprocessor industry, driving continuous innovation and progress in the fields of computer technology and electronic devices.

Over time, the applicability of Moore's Law has steadily waned [122, 179]. For instance, Fig. 1.1 presents a 42-year span of microprocessor trend data [1], showing the observation that transistors are approaching their physical limits as their sizes diminish. This imposes limitations on the performance growth of individual cores, given that sustaining frequency and typical power becomes challenging.

In this context, the advent of multi-core processors has emerged as a viable solution. These processors integrate multiple cores onto a single chip, allowing for improved performance through parallel processing instead of relying solely on increasing clock speeds or exceeding power consumption limits. Consequently, multi-core processors have assumed a pivotal role in this post-Moore's Law

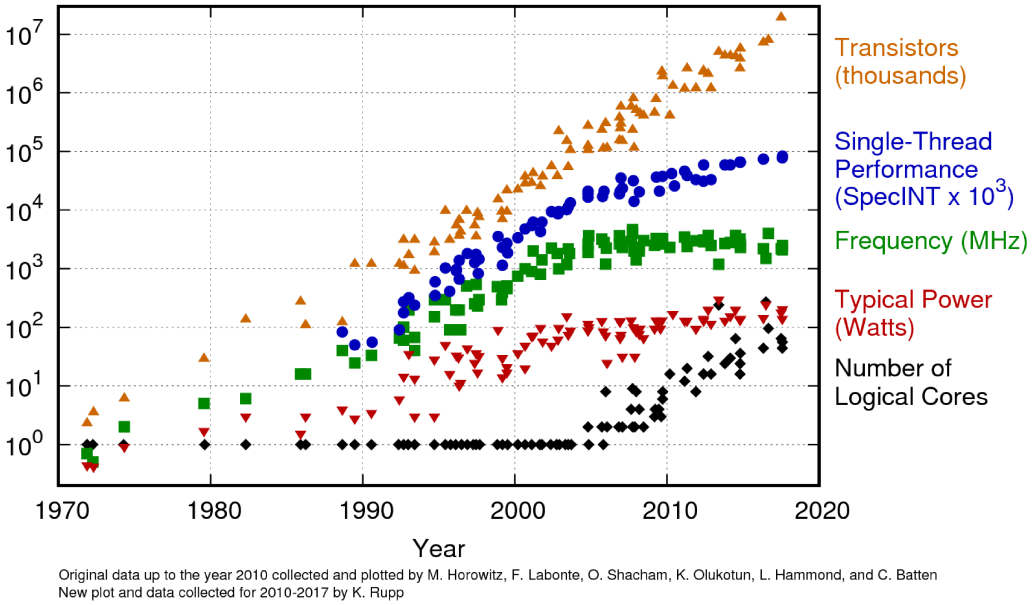


Figure 1.1: 42 years of microprocessor trend data.

era. The multi-core processor has progressed from its initial dual-core configuration [22] to contemporary architectures featuring hundreds of cores. For instance, the Kunpeng 920 processor [48], rooted in the ARM architecture [8], touts a remarkable 64 cores, while the AMD EPYC 9754 processor [5], built upon the x86 architecture [87], boasts 128 cores and 256 hyper-threads.

1.2 Concurrency Control

1.2.1 Overview

As delineated in §1.1, multi-core hardware has achieved widespread adoption. To keep pace, multi-threaded software, including operating systems [52, 59], databases [61], and network libraries [18], persistently emerges. Between low-level multi-core hardware and high-level multi-threaded applications, concurrency control technology serves as a vital bridge, providing an abstraction that empowers developers to seamlessly and efficiently leverage the advantages of multi-core architectures, allowing them to concentrate on the development of their applications without needing to delve into intricate hardware details.

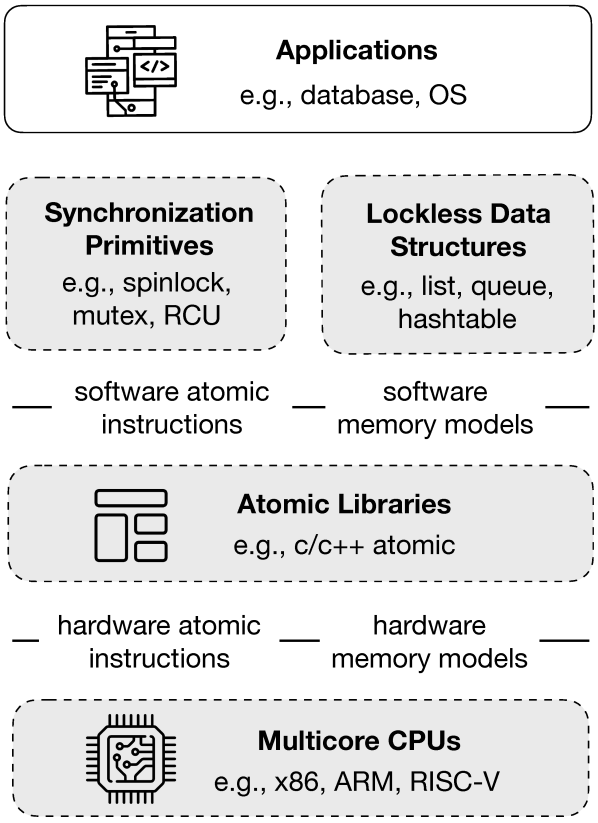


Figure 1.2: A typical software and hardware stack for multi-core systems.

Fig. 1.2 shows a typical software and hardware stack for multi-core systems, which consists of the following components:

- The lowest tier encompasses multi-core hardware, including architectures such as x86 [87], ARM [8], and RISC-V [68]. This hardware layer comprises two fundamental components. Firstly, hardware atomic operations ensure that when memory access occurs concurrently from multiple cores, a sequence of instructions from one core takes place in a single step atomically. Secondly, to boost overall performance, modern hardware often reorders instructions. Hardware memory models [168] provide guidelines on how memory access instructions can be reordered.

- The second tier from the bottom includes software atomic libraries [14], which provide software atomic instructions and memory models [140], based on their hardware counterparts. The primary objective of this tier is to offer

an abstraction to achieve cross-platform portability, ensuring that software can run seamlessly on different hardware platforms.

- The third tier from the bottom, contains common high-level libraries for concurrency control, such as synchronization primitives like spinlock [93], mutex [71], and RCU [150], as well as lockless data structures such as lists [120], queues [151], and hash tables [132].

- The topmost tier is the application layer, wherein developers can harness the underlying concurrency control technology to craft high-performance concurrent applications.

1.2.2 Importance

Leveraging the aforementioned concurrency control technology can facilitate effective communication among multiple cores for the utilization of parallel computing resources. Nevertheless, it is important to note that this approach also brings forth fresh challenges in the realm of software development. The absence of well-designed and properly implemented communication can potentially serve as a performance bottleneck for the system, thereby constraining overall performance speedups.

Amdahl's Law [112], introduced by computer scientist Gene Amdahl in 1967, stands as a foundational principle in the domains of computer science and parallel computing. This law serves as a metric for quantifying the extent of performance improvement achievable when introducing multi-core solutions [125]. The mathematical expression of Amdahl's Law is as follows:

$$S(n) = \frac{1}{1 - p + \frac{p}{n}} \tag{1.1}$$

where n stands for the number of processors, p denotes the proportion of execution time that runs in parallel, and $S(n)$ signifies the speedup achieved through n processors.

Proportion of the parallel code (p)	100%	99%	90%	80%	50%
Speedup ($S(10)$)	10	9.17	5.26	3.57	1.82

Table 1.1: The speedup of a 10-core system varies with different proportions of the parallel code.

```
1  thread() {
2   do_something_locally();
3   // critical section
4   lock_acquire(&lock);
5   sequential_code();
6   lock_release(&lock);
7   do_something_locally();
8  }
```

(a) Synchronous communication with locks.

```
9  if(FAA(counter) > N)
10  do_something(data)
```

(b) Ad-hoc communication with atomic instructions.

```
11 thread1() {
12  do_something_locally();
13  enqueue(q, data);
14  do_something_else_locally();
15 }
16
17 thread2() {
18  data_t *data;
19  data = dequeue(q);
20  if (data != NULL)
21      do_something_locally(data);
22 }
23
```

(c) Asynchronous communication with data structures.

Figure 1.3: Example code of synchronous, asynchronous, and ad-hoc communication.

Amdahl's Law elucidates that the performance enhancement achieved through parallelization is fundamentally limited by the proportion of the parallel code in the program. For instance, as depicted in Table 1.1, the speedup of a 10-core system varies with different proportions of the parallel part. It is noteworthy that even if 99% of the code is parallel, the speedup reduces to 9.17. This underscores that inadequate concurrency control limits performance, making the desired speedup unattainable. Therefore, in pursuit of substantial performance gains, it becomes imperative to employ effective concurrency control methodologies, thereby fully unleashing the performance potential inherent in multi-core systems.

1.2.3 Asynchronous Communication

As emphasized in §1.2.2, concurrency control stands as a crucial feature guaranteeing the performance of concurrent software, and ineffective communication

can significantly harm performance. In this subsection, we will introduce the common ways to perform the communication. There are three primary methods: synchronous communication, asynchronous communication, and ad-hoc communication. Fig. 1.3 illustrates an example code showcasing the utilization of these methods.

- Synchronous communication involves the concurrent access and manipulation of shared resources by multiple threads. This method relies on locks to ensure mutual exclusion among threads, ensuring that only one thread can access the shared resources at any given time. As depicted in Fig. 1.3a, a `thread` acquires the lock (line 4) before interacting with the critical section resources and releases it (line 6) upon completing the operation.

- Asynchronous communication typically relies on concurrent queues for its implementation. In this approach, threads communicate by enqueuing and dequeuing data from these queues. As illustrated in Fig. 1.3c, `thread1` transfers data to `thread2` by enqueuing it onto the queue (line 13). Following the enqueuing operation, `thread1` proceeds to execute local tasks. Subsequently, `thread2` retrieves the data by dequeuing the queue (line 19).

- Ad-hoc communication is a method that directly employs atomic instructions to synchronize threads. In this approach, threads execute these instructions for shared memory locations. As depicted in Fig. 1.3b, a thread conducts an atomic fetch-and-add (FAA) operation (line 9) on a shared counter. A certain operation is executed by the thread only if the counter reaches the limit `N`.

Among these approaches, asynchronous communication stands out as a simple and effective method for managing concurrency in multi-core software. Programming languages such as Rust [69] and Go [74] offer channels for asynchronous communication, using message passing to transfer data between threads, providing a clearer and more straightforward approach to handling concurrency.

In storage and networking systems like DPDK [18], SPDK [72], and io_uring [23], asynchronous communication is facilitated through the utilization of a ring buffer. The ring buffer serves as a conduit for passing commands between different components, such as between user space and kernel space, network card and CPU, or memory and disk. This thesis is dedicated to discussing the design and implementation of asynchronous communication in modern software systems.

1.3 Challenges of Asynchronous Communication

The evolution of hardware, particularly multi-core hardware, introduces novel challenges for asynchronous communication, encompassing both performance and correctness aspects. Performance issues stem from the intricate cache and memory hierarchy in contemporary multi-core architectures, resulting in problems such as cache misses, cache coherence overhead, and long non-uniform memory access (NUMA) [83] latency. Furthermore, modern multi-core hardware increasingly converts software atomic instructions into hardware-implemented ones to enhance their performance. Effectively harnessing them presents its own set of challenges. Additionally, correctness challenges arise due to the weak memory models employed in specific architectures like ARM and RISC-V. These models allow for aggressive memory access instruction reordering, necessitating precise implementation of memory barriers to ensure data consistency. In the subsequent subsections, we will explore these challenges individually.

1.3.1 Cache and Memory Hierarchies

Cache Hierarchy. Multi-level cache is a prevalent cache architecture utilized in multi-core processors, comprising three cache tiers: L1, L2, and L3 (Fig. 1.4). The L1 cache, characterized by its minimal capacity yet maximum speed. The L2 cache offers increased capacity at a slower speed, typically dedicated to individual cores. Meanwhile, the L3 cache, being the largest and slowest, is shared

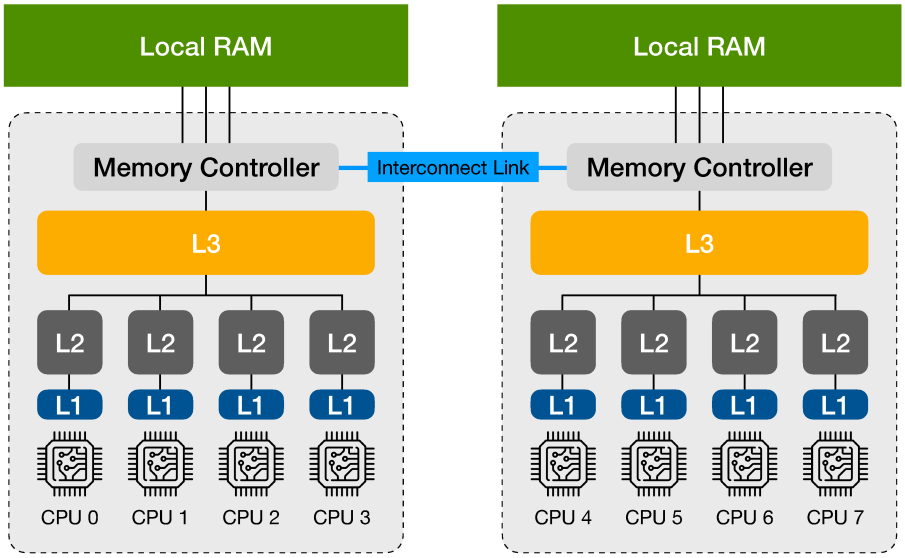


Figure 1.4: Multi-level caches and non-uniform memory access.

among cores within the same CPU socket. This hierarchical cache structure effectively mitigates data access latency from the CPU, thereby augmenting processor efficiency: during memory access instructions of the CPU, it first searches for the required data in L1, then L2, and subsequently L3. In the event that the data is not located in any of these cache levels, it is retrieved from the main memory.

Non-uniform memory access. Non-uniform memory access (NUMA) is commonly employed in multi-core processors (Fig. 1.4). Each processor is associated with a designated local memory node, while a high-speed interconnect network (e.g., QuickPath [41], HyperTransport [7]) links various memory nodes to optimize data transfer and communication. In NUMA systems, the duration of memory access varies depending on the physical proximity of memory to the processor. Generally, accessing local memory offers the shortest latency. Conversely, accessing remote memory may lead to increased latency.

Cache Coherence. To ensure data consistency among various cores in the aforementioned multi-level caches and NUMA architectures, cache coherence protocols are required. One of the most prominent protocols is MESI [58], which delineates four states (Modified, Exclusive, Shared, Invalid) to monitor

the status of data within cache lines and to elucidate how these states transition during read and write operations. Processors must scrutinize the states of cache lines and engage in communication with other cores, which introduces certain performance overhead that can potentially lead to significant performance degradation, particularly in highly parallel multi-core systems.

For instance, considering the scenario where two threads $\mathtt{t}_0$ and $\mathtt{t}_1$ modify data from two different cores. On the one hand, if $\mathtt{t}_1$ writes to the same data as $\mathtt{t}_0$, $\mathtt{t}_0$ will receive a bus message from $\mathtt{t}_1$, prompting it to invalidate its local copy of the data. Subsequently, when $\mathtt{t}_0$ writes the data, it will send bus messages to invalidate the copy in $\mathtt{t}_1$. On the other hand, if $\mathtt{t}_1$ reads the same data as $\mathtt{t}_0$, $\mathtt{t}_0$ will receive a bus message from $\mathtt{t}_1$ and reply with the value of the data. These bus communications increase the latency of reading and writing the data, thereby causing performance degradation.

1.3.2 Leveraging Modern Hardware Instructions

As depicted in Fig. 1.2, the implementation of asynchronous communication relies on the use of atomic instructions. These instructions ensure that, in the context of multi-threaded execution, where interleaving may occur, certain instructions can still be executed atomically without interruption by instructions from other threads. One of the most representative atomic instructions is the compare-and-swap (CAS). It operates with the following semantics: CAS (v, x, y) atomically checks if the value at memory address v is x. If true, it updates it to y and returns the old value x.

With the continuous advancement of hardware, certain common software atomic instructions are now integrated directly into hardware to bolster performance. For instance, through the ARM Large System Extension (LSE) [9], hardware atomic instructions such as fetch-and-add (FAA) and maximum (MAX) were introduced. Similarly, within the x86 architecture, there exists a dedicated hardware FAA instruction. For certain software components like concurrent counters, using FAA yields significant performance improvements compared to

CAS. This is primarily because in scenarios where multiple CAS operations attempt to update a variable simultaneously, if one succeeds, the others are all most likely to fail, necessitating user code to retry; whereas multiple FAA operations updating a variable simultaneously can be executed sequentially without the need for retries.

Although employing FAA greatly improves performance compared to CAS, directly substituting it for CAS presents a notable obstacle. The reason is that FAA does not conditionally update a variable but always succeeds. Consequently, this can lead to situations where an update that is too old to be valid would fail if CAS is used because the compared value has changed, but it will still succeed when using FAA, which requires careful handling.

1.3.3 Weak Memory Models

With hardware advancements, contemporary architectures such as Arm and RISC-V are gaining increased attention. These architectures employ sophisticated methods to rearrange memory access instructions, aiming to enhance performance. Weak Memory Models (WMMs) establish guidelines for optimizing memory access instructions in concurrent programs running on modern multicore processors. While these optimizations can accelerate program execution, uncontrolled reordering may result in erroneous behavior in concurrent programs dependent on specific memory access sequences. Consider the following code snippet:

```
1  // Thread 1          // Thread 2
2  x = 1;              r1 = y;
3  y = 1;              r2 = x;
```

Initially variables x and y are set to 0. In a WMM scenario, it is feasible for thread 2 to observe r1 = 1 and r2 = 0, despite thread 1 writing to x before y. This occurrence arises due to the potential reordering of writes to x and y, or reads from y and x, by either the hardware or the compiler, or both.

To prevent such erroneous reordering, which can lead to incorrect behavior,

programmers must integrate memory barriers into their code. Memory barriers are specialized hardware instructions that enforce either a partial or total order on memory load and store operations on a specific core. Typically, there are four types of barriers: load-load barriers, load-store barriers, store-store barriers, and store-load barriers, each crafted to prevent the reordering of corresponding memory accesses.

Furthermore, there are four common modes of fences built upon the combination of these instructions: sequentially consistent, acquire, release, and relaxed (i.e., plain memory access). An acquire fence prevents the memory reordering of any read that precedes it in program order from any read or write that follows it in program order. Similarly, a release fence prevents the memory reordering of any read or write from any write that follows it in program order. The relaxed fence imposes no barrier, whereas the sequentially consistent fence is the strongest.

Utilizing memory fences presents challenges: The excessive use of fences can introduce performance overhead and scalability issues [146], as they restrict the hardware's capability to leverage parallelism and optimize memory access. Conversely, using fewer fences may result in reordering bugs. Thus, it is imperative to select the appropriate fences judiciously to maximize the performance benefits of modern architectures while ensuring correctness.

1.4 Asynchronous Communication through Concurrent Queues

As previously mentioned, asynchronous communication typically relies on concurrent queues. These queues serve as fundamental data structures and find extensive application across diverse domains, including network libraries, operating systems, and high-performance computing. Their pivotal role lies in enhancing the performance and scalability of upper-layer components. In the subsequent sections, we will introduce typical queue categories, review existing

works on them, examine how they address the aforementioned challenges, and delve into their associated limitations.

FIFO queues. A FIFO queue is a data structure that adheres to the principle of first-in, first-out. This means that the element that is added first to the queue is also the one that is removed first from the queue. However, unlike linked lists or trees where operations on different parts may have no interference (i.e., reading/writing the same metadata) [120], concurrent enqueue (enqueue) and dequeue (dequeue) operations of FIFO queues inevitably demand simultaneous updates to the head or tail of the queue to prioritize data, presenting a theoretical challenge to their scalability. The main factor determining the performance of a FIFO queue is the interference between concurrent operations, *i.e.*, between enqueues, between dequeues, or between enqueues and dequeues. We refer to these as *enq-enq*, *deq-deq*, and *enq-deq* interference, respectively. Interferences manifest in the form of 1) cache-line bouncing when control variables are frequently updated by one thread and read by another, *e.g.*, to check if the queue has data, and 2) serialization of contended updates to control variables, *e.g.*, when multiple threads try to create or read the same entry.

Existing queue designs often employ optimizations to reduce enq-enq and deq-deq interference, *e.g.*, updating control variables with "always-successful" atomic instructions such as *fetch-and-add* (FAA) [28, 161, 185, 162] because, in principle, they can be serialized in hardware and thus perform better under high contention than software solutions with *compare-and-swap* (CAS) [157, 166]. However, existing designs tend to neglect the enq-deq interference even though it substantially impacts performance, in particular in the common single producer or single consumer scenarios, *e.g.*, ringbuffers for asynchronous I/O in Linux io_uring [23].

In fact, some queue optimizations from the literature [157, 161] inadvertently increase the enq-deq interference or introduce undesirable side-effects that degrade performance in uncontended cases. For example, dequeue operations using FAA must either block in a pessimistic way [28], or risk overtaking

slow concurrent enqueue operations; to avoid data corruption, such enqueue operations must be invalidated and repeated later [161]. Besides harming performance, such strategies cannot be applied for online profiling where enqueues writing a log should not be delayed by dequeues that read the log. Similarly, some techniques that avoid concurrent enqueue operations from waiting for each other also require dequeue operations to invalidate parts of the queue [157]. Strategies to improve performance of these techniques by reducing the number of invalidations, *e.g.*, busy-looping before invalidating [157, 161], drastically increase the latency of dequeue calls on empty queues, making them unsuitable for certain workloads, *e.g.*, multiplexing across multiple message queues.

Relaxed queues. In certain scenarios, strict adherence to the FIFO property may not be crucial. For instance, in network servers that buffer and process tasks, tasks originating from different sources do not necessarily need to be handled in a strictly sequential order but rather within a reasonable timeframe. As a result, several studies have proposed relaxed specifications as a means to enhance performance, including quasi-linearizability [90], local linearizability [114], and random balancing [116]. Among these specifications, the k-FIFO [90] specification strikes a balance between ordering semantics and simplicity for non-experts to comprehend. In this specification, the oldest data in the queue is not necessarily retrieved first but is guaranteed to be dequeued within a maximum of k dequeue operations.

In addition to specifications, there are queue implementations that satisfy these specifications, primarily falling into two categories. The first category utilizes a random strategy [90, 131], which aims to reduce interference between producers and consumers by randomly assigning them to different entries within a given range. The second category is distributed queues [116, 114], where the entire queue is partitioned into multiple queues.

Nevertheless, with the advancements in modern CPU architectures, hardware atomic instructions have been introduced to boost the performance of upper-layer software. State-of-the-art FIFO queues [157, 161, 185, 159] leverage

these hardware instructions, employing FAA to handle interference on the head or tail, resulting in substantial performance improvements. Nonetheless, the aforementioned relaxed queues face challenges in effectively leveraging scalable hardware instructions (**P1**), as data is either placed randomly or in a distributed manner instead of adhering to strict sequentiality. For instance, LCRQ [157], a FIFO queue that utilizes FAA, achieves significantly faster performance than the classic Michael-Scott queue (MSQ) [152] that utilizes CAS. However, when it comes to distributed queue implementations, the performance of a distributed LCRQ is similar to that of a distributed MSQ [114]. As a result, state-of-the-art queues with relaxed specifications do not demonstrate better performance compared to state-of-the-art FIFO queues (§3.9.2). Moreover, they also suffer from issues such as memory space inefficiency, the inability to achieve higher performance despite employing weaker semantics, and limitations in scenarios (§3.2).

Work stealing queues. Among all distributed queues, work stealing stands out as a widely-used scheduling technique for parallel processing on multicore systems. Each core owns a queue (*owner*) and acts as both the *producer* and the *consumer* of its own queue to put and get tasks. When a core completes its tasks and the queue is empty, it then steals another task from the queue of another processing core to avoid idling (*thief*). A number of stealing policies [138, 145, 137, 176, 127, 155] have been proposed to choose the proper queue (*victim*) to steal from, which can bring significant speedups depending on the use-cases. Due to these features, work stealing is widely used in parallel computing [119, 118, 73, 167, 148, 172, 104, 40], parallel garbage collection [126, 127, 171, 177, 109], GPU environment [100, 180, 175, 102, 181], programming language runtimes [98, 76, 74, 49, 142, 117, 143], networking [149] and real-time systems [144].

However, as parallel processing is applied to more workloads, current implementations of work stealing become a bottleneck, especially for small tasks. For example, web frameworks running over lightweight threading abstractions,

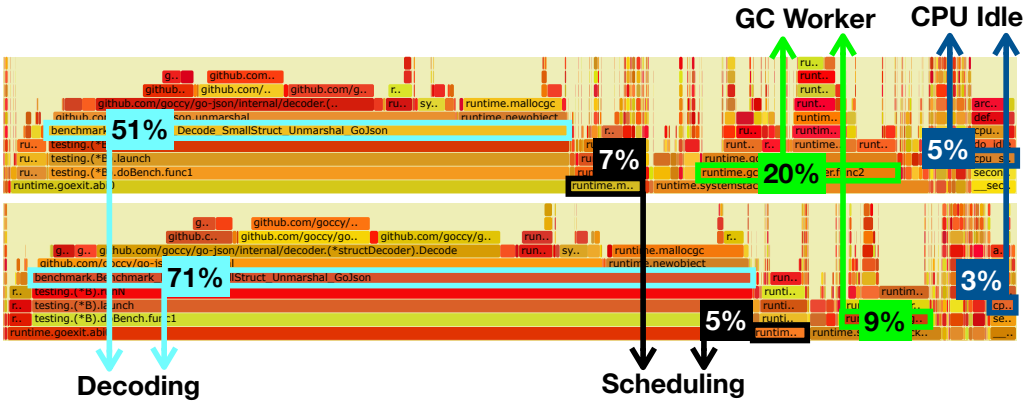


Figure 1.6: Profiling results for go-json complex object decoding (~1μs/op) benchmark [24], with the original work stealing queue (up) and with BWoS (down).

such as Rust's Tokio and Go's goroutines, often contain many very small tasks, leading the Tokio runtime authors to observe that "the overhead from contending on the queue becomes significant" and even affects the end-to-end performance [57]. Similarly, high-performant garbage collectors, such as Java G1GC [30], rely on work stealing for parallelizing massive mark/sweep operations, which comprise only a few instructions. The work stealing overhead becomes a performance bottleneck for GC [171, 127, 126, 177].

As a third example, in Fig. 1.6, we profile the GoJson object decoding benchmark, which uses goroutines both for GC workers and for parsing complex objects. Only 51% of all CPU cycles constitute the useful workload (JSON decoding). The remaining cycles are spent on the runtime code, including 7% on lightweight thread scheduling, 20% on GC, 5% on kernel code idling the CPUs, *etc*. As both the scheduler and the GC code rely on work stealing, improving its performance can result in massive efficiency gains. Furthermore, the benefit is not limited to the above-mentioned scenarios, but expands to all fine-grained tasks parallel processing scenarios. Thus, we ask the following question: *How can we improve performance of work-stealing queues for fine-grained tasks to the benefit of a large range of common applications*?

Existing work-stealing queues suffer from four main sources of inefficiency:

P1: Synchronization overhead. Due to the possibility of a steal, local

queues must use stronger atomic primitives (*e.g.*, atomic compare-and-swap and memory barriers) than a purely sequential queue. Queues with a FIFO policy are generally implemented as single-producer multiple-consumer (SPMC) queues [86, 34, 77, 94], thereby treating *steal* similarly to *get*, and thus distributing the costs of stealing equally between owner and thieves. This also applies to the existing block-based queues [184], which lack any optimizations specific to the work-stealing use case to achieve high performance in the presence of thieves (§4.6.2, §4.7). Queues with a LIFO policy, such as the well-known and widely-used ABP queue [95, 182, 73], suffer from memory barrier overhead [173, 145] to avoid the conflict between the owner and thieves, even when they operate on different tasks.

P2: Thief-induced cache misses. Since steals update the metadata shared between the owner and the thieves, they cause cache misses on subsequent accesses to the queue by its owner. This problem is especially apparent on unbalanced workloads, which feature high steal rates—for example, in the JVM Renaissance benchmarks [170], 10% of all items are stolen on average. Although strategies such as batching (*e.g.*, *steal-half* [121]) can reduce the frequency of steals, they often cause *overstealing* which introduces additional overhead (§4.2.1.3).

P3: Victim selection. To improve the workload distribution, advanced policies for selecting the victim queue to steal from require scanning the metadata of several queues, *e.g.*, to find the longest queue. Doing so, however, causes contention for its owner and severely limits the improvement from advanced victim selection policies (§4.2.1.3).

P4: Correctness under weak memory models (WMMs). Correctly implementing concurrent work-stealing queues on weak memory architectures, such as Arm servers for datacenters [129, 92], is very challenging because it requires additional memory barriers to prevent unwanted reordering. Using redundant barriers can greatly reduce the performance of work-stealing [141], while not including enough barriers can lead to errors, such as in the C11 [14]

version of the popular unbounded Chase-Lev deque translated from formally verified ARMv7 assembly [163]. Even the popular Rust Tokio runtime required a fix to its implementation of work stealing [3].

1.5 Contributions

We investigate the problem of how to design and implement high-performance and dependable queues in the above-mentioned categories and present novel methods and algorithms to address the challenges in existing state-of-the-art works. We have developed several queues, including block-based queue (BBQ), concurrent nested queue (CNQ), and block-based work-stealing queue (BWoS), which are validated and optimized using a model-checking-based framework. These queues have been successfully integrated into real-world applications, including DPDK, Linux IO_uring, Java GC, and Go and Rust runtimes. Our experiments demonstrate significant end-to-end performance improvement for industrial software over state-of-the-art approaches. We list the following contributions in this thesis:

FIFO queues. We present Block-based Bounded Queue (BBQ), a novel ring-buffer design that dramatically reduces the enq-deq interference by splitting the entire buffer into multiple blocks and splitting the control variables into the block-level and queue-level variables. In the common case, enqueue and dequeue only access block-level control variables of their current blocks. When enqueue and dequeue work on different blocks, the disjoint control variables avoid any interference between these operations. That is particularly important in reducing the cache-line bouncing of head and tail pointers when determining whether the queue is full or empty. Furthermore, we use hardware-serialized FAA operations to update block-level control variables for allocating entries inside blocks, while queue-level control variables on the other hand are updated with slower, software-serialized CAS operations; since this is only necessary in the rare event that operations need to move to the next block, the performance impact of these operations is negligible. Our block-based approach allows us to

perform these optimizations without incurring undesirable side-effects. Finally, to ensure that BBQ correctly works on weak memory models (WMMs) — including those from Arm [8] and RISC-V [68] architectures — we have verified and optimized the barriers and fences of BBQ with the VSync framework [165].

In contrast to previous work, our block-based approach is applicable to a large spectrum of scenarios. BBQ supports single or multiple producers/-consumers, fixed- or variable-sized entries, and retry-new and drop-old modes. Retry-new is the typical producer-consumer mode for message passing and work distribution scenarios; drop-old is a lossy/overwrite mode for profiling/tracing [54, 13] and debugging [174] scenarios, in which producers may overwrite unconsumed data if the buffer is full.

In our experimental evaluation, BBQ outperforms several industrial queues and ringbuffers. In single-producer/single-consumer micro-benchmarks, BBQ yields 11.3x to 42.4x higher throughput than Linux circular buffer [51], DPDK ring buffer [18], Boost lock-free queue [12], and Meta's Folly queue [28]. In real-world scenarios, BBQ achieves up to 1.5x, 50.5x, and 11.1x performance improvements in benchmarks of DPDK, Linux io_uring [23], and LMAX Disruptor [53], respectively. In our profiling benchmarks, BBQ enabled with the lossy/overwrite mode achieves up to 4.7x higher throughput than Google's Guava EvictingQueue [35] and Apache Commons CircularFifoQueue [6], and can sustain up to 143.2x lower enqueue latency than the other two queues.

Relaxed queues. We propose Concurrent Nested Queue (CNQ). CNQ is an MPMC queue where each entry does not directly store data. Instead, it contains a single producer single consumer (SPSC) FIFO queue (denoted as *capsule*) which can hold multiple data items. This loosely coupled design allows for the construction of CNQ by integrating the capsule into an existing MPMC FIFO queue (denoted as *tube*), thereby preserving its features, such as utilizing the scalable FAA instruction. CNQ provides the k-FIFO relaxed ordering guarantee by imposing a constraint on the distance between the oldest and newest capsules that producers and consumers can interact with.

The design of CNQ ensures its performance by using a fast-path-slow-path mode [133]. CNQ consists of two levels: the capsule level and the tube level. In the common case, enqueue and dequeue operations occur inside the capsule, resulting in the interference as well as the locality of metadata and data comparable to that of an SPSC queue. Furthermore, when producers and consumers operate on different capsules, there is no interference between them. When they advance to other capsules, the interference becomes comparable to that of its tube. However, unlike MPMC FIFO queues that contend for each entry, the tube of CNQ only contends for each capsule, which is a rare occurrence and does not become a performance bottleneck when using a reasonable k.

To demonstrate the generality of CNQ, we integrated CNQ into several MPMC FIFO queues, including the ring buffer in the Data Plane Development Kit [18], the Scalable Circular Queue [161], the Block-based Bounded Queue (BBQ) [184], and FAAQ, our optimized version of BBQ tailored for multiple consumers workloads. Benchmarking results demonstrate that CNQ achieves significant performance improvements of up to 162x, 51x, 16x, and 12x compared to these queues, respectively, when k was set to 10^3.

To further evaluate CNQ, we conducted comparisons with ten FIFO and relaxed queues using the Scal benchmark framework [115]. Across different workloads and with k values ranging from 10^2 to 10^4, CNQ consistently outperformed state-of-the-art relaxed queues, achieving higher throughput ranging from 6.0x to 14.5x times while reducing data latency by 1.3x to 79.4x. Additionally, we have formally verified CNQ on weak memory models with a model-checking-based framework [136, 135, 165].

Work-stealing queues. We introduce BWoS, a *block-based work stealing* (BWoS) bounded queue design, which provides a practical solution to these problems, drastically reducing the scheduling overhead of work stealing. Our solution is based on the following insights.

First, we split each per-core queue into multiple blocks with independent metadata and arrange for the owner and the thieves to synchronize at the

block level. Therefore, in the common case where operations remain within a block, we can elide synchronization operations and achieve almost single-threaded performance (§4.3.2). Similarly, since a queue owner and the thieves share only block-local metadata, they do not interfere when operating on different blocks (§4.2.1). We can arrange for that to happen frequently by allowing stealing tasks from the middle of the queue.

Second, we improve victim selection with a *probabilistic policy*, which approximates selecting the longest queue (§4.3.4), while avoiding the severe interference typical of the prior state-of-the-art (§4.2.1), to which we can integrate NUMA-awareness and batching.

Finally, we ensure correctness under WMMs by verifying BWoS with the GenMC model checker [136, 135] and optimizing its choice of barriers with the VSync toolchain [165] (§4.5).

As a result, BWoS offers huge performance improvements over the state-of-the-art (§4.6). In microbenchmarks, BWoS achieves up to 8-11x throughput over other algorithms. In representative real-world macrobenchmarks, BWoS improves performance of Java industrial applications by up to 25% when applied to Java G1GC, increases throughput by 12.3% with 6.74% lower latency and 60.9% lower CPU utilization for Rust Hyper HTTP server when applied to the Tokio runtime, and speeds up JSON processing by 25.8% on average across 9 different libraries when applied to the Go runtime.

Returning to our motivating example (Fig. 1.6), applying BWoS to the Go runtime removes 29% of scheduling time, 55% of GC time, and 40% of CPU idle time, while increasing the CPU time ratio for useful work from 51% to 71%.

Chapter 2

BBQ: A Block-based Queue

2.1 Overview

Concurrent bounded queues have been widely used for exchanging data and profiling in operating systems, databases, and multithreaded applications. The performance of state-of-the-art queues is limited by the interference between multiple enqueues (enq-enq), multiple dequeues (deq-deq), or enqueues and dequeues (enq-deq), negatively affecting their latency and scalability. Although some existing designs employ optimizations to reduce deq-deq and enq-enq interference, they often neglect the enq-deq case. In fact, such partial optimizations may inadvertently increase interference elsewhere and result in performance degradation.

We present Block-based Bounded Queue (BBQ), a novel ringbuffer design that splits the entire buffer into multiple blocks. This eliminates enq-deq interference on concurrency control variables when producers and consumers operate on different blocks. Furthermore, the block-based design is amenable to existing optimizations, *e.g.*, using the more scalable *fetch-and-add* instruction. Our evaluation shows that BBQ outperforms several industrial ringbuffers. For example, in single-producer/single-consumer micro-benchmarks, BBQ yields 11.3x to 42.4x higher throughput than the ringbuffers from Linux kernel, DPDK, Boost, and Folly libraries. In real-world scenarios, BBQ achieves up to 1.5x, 50.5x,

and 11.1x performance improvements in benchmarks of DPDK, Linux io_uring, and Disruptor, respectively. We verified and optimized BBQ on weak memory models with a model-checking-based framework.

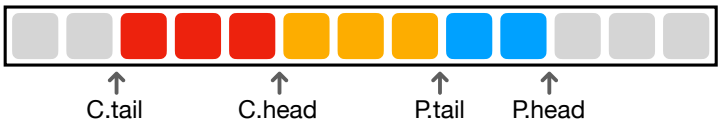
The remainder of this chapter is organized as follows. In §2.2, we gradually introduce the challenges of reducing the interference between enqueue and dequeue operations, discussing how existing queues tackle these challenges, and the limitations of their solutions. In §2.3, we present our block-based approach and the high-level design of BBQ. In §2.4, we describe BBQ implementation including the support for retry-new and drop-old modes and variable-sized entries. In §2.5, we report our results in verifying BBQ on WMMs and relaxing its memory barriers. In §2.6, we experimentally compare the performance of BBQ and several industry-grade concurrent queues. In §2.7, we conclude our work.

2.2 Background and Related Work

We now introduce scalability challenges of bounded queue designs and discuss related work and their limitations. Figure 2.1 illustrates the discussion in this section depicting a simple lockless bounded queue with multiple-producer multiple-consumer (MPMC) support, which is the algorithm behind the widely-used DPDK ringbuffer [18].

Producers first check whether the queue is full (line 4) and then try to *allocate* the next entry via CAS (line 6). Upon success, the producer copies the data into the entry and *commits* it (line 11). Similarly, consumers first try to *reserve* an entry (line 22). Upon success, the consumer copies the data back and confirms that the data has been *consumed* (line 31).

(P1) Consumer contention on C.head. A straightforward form of deq-deq interference is caused by multiple consumers concurrently calling dequeue and contending on updates to C.head. Several works (on bounded and unbounded queues) tackle this contention using FAA to update the head [157, 161, 166] because FAA is more scalable than CAS on common architectures. However, since FAA cannot conditionally update the memory location, it may break, for




```
1  enqueue(data){
2  again:
3    ph = LOAD(P.head);
4    pn = ph + 1;
5    if (pn > LOAD(C.tail) + SZ)
6      return FULL;
7    if (!CAS(P.head, ph, pn))
8      goto again;
9    entry[pn % SZ] = data;
10   while(LOAD(P.tail) != ph);
11   STORE(P.tail, pn);
12   return OK;
13 }
```

```
14 dequeue(){
15 again:
16   ch = LOAD(C.head);
17   cn = ch + 1;
18   if (cn > LOAD(P.tail))
19     return EMPTY;
20   if (!CAS(C.head, ch, cn))
21     goto again;
22   data = entry[cn % SZ];
23   while(LOAD(C.tail) != ch);
24   STORE(C.tail, cn);
25   return data;
26 }
```

Figure 2.1: A simple MPMC bounded queue. CAS, LOAD, and STORE are atomic operations with sequentially consistent semantics on WMMs. C.head and C.tail refer to consumers; P.head and P.tail refer to producers.

example, the invariant of C.head never exceeding P.tail. To address that, Meta's Folly queue [28] implements the partial, not total dequeue method, which spins until dequeue succeeds [123]. With such interface, dequeue calls return only if there is an entry to consume, otherwise blocking the thread indefinitely.

Another solution is to "fix the state" when C.head exceeds P.head by invaliding entries between them, as done by LCRQ [157]. Unfortunately, that causes consumers to hamper the progress of producers. SCQ [161] solves the producer starvation by limiting the number of consecutive invalidations with a threshold, as we describe below. Nevertheless, the remaining invalidations still degrade the enqueue performance, and the SCQ implementation [160] employs a trick to reduce the probability of invalidating entries: Consumers check several times in a loop whether the entry has been committed before actually invalidating it. Unfortunately, this *delayed invalidation* trick increases the latency of dequeue when the queue is empty by several orders of magnitude — we experimentally demonstrate this empty-deq in §2.6.2.3.

(P2) Producer contention on P.head. A straightforward form of enq-enq

interference is caused by multiple producers concurrently calling enqueue and contending on updates to P.head. Here, Folly queue again resorts to FAA and turns enqueue into a partial method, which waits until free entries are available, potentially blocking the thread forever.

Nikolaev proposes a novel idea to implement a total queue while using FAA [161]: SCQD combines two SCQ index queues (fq and aq) with a data array. Upon enqueue on SCQD, the thread gets an index from fq, copies the data in the corresponding entry of the data array, and then puts the index into aq. Dequeueing works the other way around. The index queues are total on dequeue but partial on enqueue, *i.e.*, dequeue returns `EMPTY` if the queue is empty, whereas enqueue loops until it succeeds. Nevertheless, the combined SCQD is still total since index queues are never full, *i.e.*, the number of indexes is fixed and matches the maximum size of the index queues. Besides the constant overhead introduced by index queues, the high latency caused by the empty-deq issue in each SCQ index queue translates into high latency in SCQD for both empty-deq and full-enq cases (see §2.6.2).

(P3) Delayed P.tail and C.tail updates. Another typical enq-enq or deq-deq interference arises from the *in-order* policy to commit (resp. consume) entries — the default policy in DPDK ringbuffer. The `head/tail` mechanism, which resembles a ticket lock, brings issues analogous to Lock-Holder- and Lock-Waiter-Preemption [178]. For example, the preemption of a thread that is about to update P.tail (resp. C.tail) causes other enqueue calls (resp. dequeue calls) to uselessly spin (lines 14 and 30) for arbitrary time periods.

Several queues implement, instead, *out-of-order* policies, allowing producers (resp. consumers) to commit (resp. consume) entries independently. In LCRQ, once consumers increment C.head such that it reaches P.tail, they start invalidating entries until C.head reaches P.head. That prevents producers from committing entries at indexes preceding C.tail, ensuring linearizability [124]. This approach starves producers and even livelocks the queue. For example, a consumer in an ongoing dequeue invalidates an entry when

C.head $=$ P.tail $<$ P.head; the producer in the ongoing enqueue increments P.head to retry; the consumer realizes C.head still did not reach P.head and retries consuming, potentially invalidating the new entry if not yet committed; and so on.

SCQ uses a threshold T to restrict the number of consecutive entries invalidated, and, thus, avoid livelocks. When a consumer invalidates an entry, it atomically decrements T. A successful enqueue resets T to its initial constant $3n - 1$, where n is twice the queue capacity. This constant is carefully derived to guarantee linearizability is never violated [161]. However, it introduces additional contention among producers and consumers updating the threshold variable, which has to be again mitigated by delayed invalidation.

DPDK [18] ringbuffer implements a more practical out-of-order policy called RTS mode [65], which trades linearizability to avoid invalidations. Consumers never move C.tail forward if C.tail would reach P.tail, returning `EMPTY` despite of any committed entry between P.tail and P.head; thus, violating linearizability. Producers employ the reciprocal strategy.

To enable out-of-order commits, RTS records whether all entries between P.tail and P.head are committed. The *prohibited window* between P.tail and P.head has dynamic length because P.tail is moved forward only once the last producer writing between P.tail and P.head commits. If producers would keep allocating entries, they would keep incrementing P.head and extending the prohibited window up to the total capacity of the queue. To prevent consumers from starving, RTS sets up a threshold to limit the maximum distance between P.tail and P.head. If that distance is reached, enqueue blocks until all producers between P.tail and P.head have committed. RTS enables out-of-order consumes with the recipocral approach.

(P4) Causes of enq-deq interference. There are two sources of interference between enqueues and dequeues: algorithmic and cache-related. While focusing on enq-enq or deq-deq cases, previous techniques introduce algorithmic interferences between enqueue and dequeue, *e.g.*, requiring producers and consumers

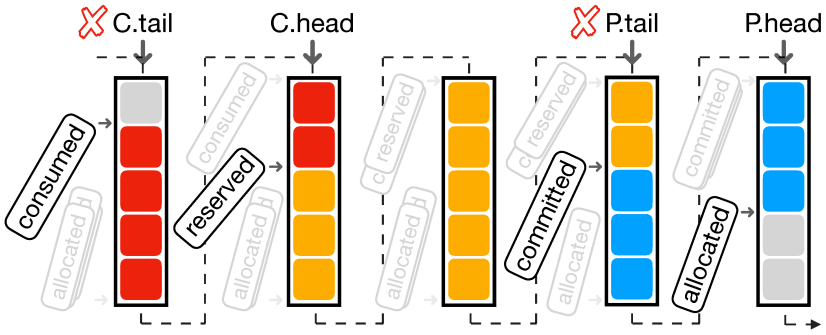


Figure 2.2: Block-based bounded queue (BBQ).

to retry operations, increasing latency, potentially causing thread starvation, or even livelock.

Let us again consider the simple algorithm of Fig. 2.1. Even though cache misses caused by writing or reading the data cannot be eliminated, cache misses on the control variables are relevant. Every time a producer calls enqueue, it allocates an entry and increments P.tail. Every time a consumer calls dequeue, it potentially suffers a cache miss by reading P.tail. If the producer is far ahead the consumer, the cache misses at P.tail seem unjustifiable. Similarly, the producer suffers cache misses on C.tail even when there is plenty space in the queue between producers and consumers. In contrast to enq-enq and deq-deq, enq-deq interference is relevant even to the single-consumer/single-producer scenario, an important scenario for the industry. In §2.6, we experimentally show a correlation between a strong decrease of L1 cache misses and the performance improvements of BBQ (§2.3 and §2.4).

2.3 Design of BBQ

2.3.1 The Block-based Approach

BBQ splits the ringbuffer into blocks, as shown in Fig. 2.2. Each block contains one or more entries, usually multiple, depending on the configuration. The queue control variables are also split into queue-level and block-level variables. Control variables C.head and P.head now point to blocks instead of entries;

P.tail and C.tail are unnecessary for the algorithm. The block-level control variables include four cursors called `allocated`, `committed`, `reserved`, and `consumed`, which track the corresponding actions within each block.

The block-based approach greatly reduces the enq-deq interference. After a block is fully committed, its producer cursors (`allocated`/`committed`) remain unmodified until the block is fully consumed, causing no additional cache misses for consumers. Moreover, producers can always determine whether a block is fully allocated without accessing consumer cursors (`reserved`/`consumed`).

Multiple producers in the same block still contend on `allocated` and `committed`. Fortunately, the block-based approach enables the enqueue operation to use FAA, avoids costly invalidations, and still allows for a total method. Producers start using a block only once it has been fully consumed. Therefore, inside the block, FAA never allocates an entry that is not consumed yet, allowing enqueue to be total. FAA may make `allocated` out of the bound of the block, but state fixing is not required. Since each block has its own control variables, an out-of-bound cursor in one block does not affect the following block.

Although our dequeue operation employs CAS to avoid consumers from invalidating entries currently used by producers, BBQ still achieves similar or better performance than other designs with FAA-based dequeues. To further improve performance for machines with Armv8.1 [8] processors supporting Large System Extensions [9] (LSE), BBQ uses the atomic maximum instruction MAX instead of CAS in dequeue.

Finally, the block-based approach enables a practical out-of-order policy similar to RTS mode of DPDK. In this case, instead of updating control variables with double-width CAS, BBQ employs more scalable FAA and MAX instructions.

2.3.2 BBQ from a Bird's-eye View

In this section, we describe the high-level algorithm of BBQ, as shown in Fig. 2.3.

Producers. To enqueue data, a producer first retrieves the current value of

```
1  status := OK(T) | FULL | EMPTY | BUSY
2  status BBQ<T>::enqueue(T data){
3  loop:
4    (ph, blk) = get_phead_and_block();
5    switch (allocate_entry(blk)){
6      case allocated(ety):
7        commit_entry(ety, data);
8        return OK();
9      case BLOCK_DONE:
10       switch (advance_phead(ph)){
11         case NO_ENTRY: return FULL;
12         case NOT_AVAILABLE: return BUSY;
13         case SUCCESS: goto loop;
14       }
15   }
16 }
17 status BBQ<T>::dequeue(){
18 loop:
19   (ch, blk) = get_chead_and_block();
20   switch (reserve_entry(blk)){
21     case reserved(ety):
22       data = consume_entry(ety);
23       if (data != NULL) return OK(data);
24       else goto loop;
25     case NO_ENTRY: return EMPTY;
26     case NOT_AVAILABLE: return BUSY;
27     case BLOCK_DONE(vsn):
28       if (advance_chead(ch, vsn)) goto loop;
29       else return EMPTY;
30   }
31 }
```

Figure 2.3: High-level design of BBQ.

P.head and the corresponding block identifier (line 4). Next, it tries allocating an entry in the block (line 5). If successful, the producer writes the data into entry `ety` and commits it (line 7). The allocation fails if the block has already been fully allocated (line 9). In this case, the producer tries to advance P.head to the next block (line 10). If successful, the producer jumps back to the `loop` label and retries the allocation (line 13). In retry-new mode, advancing P.head fails if the next block is not yet fully consumed, *i.e.*, the whole queue is considered full. BBQ distinguishes the failure reason: `BUSY` when some dequeue operation is ongoing and `FULL` otherwise. Returning `BUSY` allows for custom

backoff implementations at the caller side, *e.g.*, parking threads after a number of retries. In drop-old mode, advancing P.head does not fail except for a seldom case discussed in §2.4.3, for which `BUSY` is returned.

Consumers. The dequeue operation is somewhat analogous to enqueue. The consumer starts by retrieving the current value of C.head and the corresponding block identifier. Next, it attempts to reserve an entry to consume (line 20), advancing `reserved`. If the reservation succeeds, the consumer reads the data (line 22) and advances `consumed`. In drop-old mode, the consumer may have to retry consuming if the producers have overwritten the block (line 24). Reserving an entry can fail in several ways. When the next entry in `blk` is allocated but not yet committed, dequeue returns `BUSY` (line 26). When `blk` is not fully allocated and all committed entries were already consumed, dequeue returns `EMPTY` (line 25). Finally, when `blk` is fully committed and fully consumed, the consumer tries advancing C.head (line 28). Upon success, it retries reserving an entry, jumping to `loop`. Otherwise, dequeue returns `EMPTY`.

Progress guarantees. Similarly to DPDK ringbuffer, BBQ is a deadlock-free queue, its progress depend on a fair scheduler. In contrast to DPDK, BBQ is less affected by CPU oversubscription (see Fig. 2.9g and Fig. 2.9n, §2.6). To see why this is the case, consider the situation where DPDK producers form a waiting chain: the last to allocate an entry can only commit once the previous has committed its entry, and so on. This waiting chain hampers the performance because the scheduler is unlikely to unpark the preempted producers in the chain's order. In BBQ, there is no such waiting chain, *i.e.*, producers commit independently. Consumers may wait for producers only in seldom cases. For example, a consumer waits for a preempted producer on the same block if the producer has allocated but not yet committed an entry. Nevertheless, any order in which the fair scheduler unparks preempted producers allows the consumer to make progress.

2.3.3 Two-Level Control Variables

Essentially, BBQ splits the control variables into two levels, namely the queue-level and block-level variables (see Fig. 2.2). Queue-level control variables point to blocks (C.head and P.head), whereas block-level control variables to entries (`allocated`, `committed`, `reserved`, and `consumed`).

Versions. As in other queues such as DPDK ringbuffer, control variables have to be versioned to identify multiple reuses of the same memory locations and, in this way, avoid ABA problems[1]. Therefore, queue-level control variables have two fields, an *index* pointing to a block and a *version* identifying how many rounds the whole queue has been reused. Similarly, block-level control variables have an *offset* field pointing to an entry within the block and a *version* field identifying how many times the block has been reused.

Phantom heads. Before producers can allocate entries in a block B, one producer has to reset B's `allocated` cursor as well as advance P.head to point to B. Without making both updates atomic, whichever update executes first may trigger an ABA problem as well. To allow both being updated atomically, we introduce the concept of *phantom head*, which is based on the following observation. The index and version values of P.head can be inferred from the versions of all `allocated` cursors in the queue (as described in §2.4.2.2). Similarly, the phantom C.head can be inferred from the versions of all `reserved` cursors. Since the phantom P.head (resp. phantom C.head) is implicitely updated whenever any `allocated` cursor (resp. `reserved` cursor) is updated, we use them instead of queue-level head variables.

Cached heads. In principle, phantom heads allow us to eliminate the C.head and P.head variables altogether. Unfortunately, phantom heads are costly: To infer them, one needs to compare the cursors of every block. Instead of eliminat-

[1] Often algorithms try to guarantee operation atomicity by reading from a control variable before and after the operation. If the same value A is read both times, the programmer assumes absence of concurrent updates and hence that the operation was atomic. This assumption breaks if other threads can temporarily change the value to $B \neq A$ and then back to A; algorithms in which this situation can occur are said to suffer from the ABA problem [107].

ing C.head and P.head, we consider them to be *cached heads*, *i.e.*, potentially stale values of the phantom heads. Cached heads only exist for performance reasons; their staleness does not affect correctness.

2.4 Implementation of BBQ

Figure 2.5 shows the low-level detail of BBQ, including data-fields, enqueue and dequeue operations for retry-new mode and drop-old mode. The drop-old mode will be introduced in §2.4.3.

2.4.1 Structure

Heads and cursors. `BBQ` has two queue-level `Head` variables and four block-level `Cursor` variables in each block. `Head` and `Cursor` types are 64-bit integers, which can be atomically updated. We reserve two bit-segments in `Head` to represent the version and index and two bit-segments in `Cursor` to represent the version and offset. Given a total number of blocks (`BLOCK_NUM`) and the capacity of a block (`BLOCK_SIZE`), the segments have the following bit-lengths:

$$\begin{aligned} |\texttt{Index}| &= log_2(\texttt{BLOCK\_NUM}) \text{ bits} \\ |\texttt{Offset}| &> log_2(\texttt{BLOCK\_SIZE}) \text{ bits} \\ |\texttt{Version}| &= 64 - max(|\texttt{Index}|, |\texttt{Offset}|) \text{ bits} \end{aligned}$$

The bit-length of `Offset` is larger than $log_2(\texttt{BLOCK\_SIZE})$ to allow for FAA-overflow detection. The `Index` and `Offset` are the least significant bits of `Head` and `Cursor`, respectively; `Version` bits immediately follow them; and reminder bits, if existent, are set to 0 and ignored. That allows us to easily manipulate these fields with FAA and MAX instructions.

For convenience, we access the bit-segments from `Head` and `Cursor` variables as if they were regular fields named `idx`, `off`, and `vsn`, *e.g.*, `allocated.idx`. Moreover, we construct variables (*e.g.*, `Head`) with the short-hand notation `Head{.vsn=version, .idx=index}`, initializing unspecified fields with

```
1  <Head, Block> BBQ<T>::get_phead_and_block(){
2    ph = LOAD(phead);
3    return (ph, blocks[ph.idx]);
4  }
5  state BBQ<T>::allocate_entry(Block blk){
6    if (LOAD(blk.allocated).off >= BLOCK_SIZE)
7      return BLOCK_DONE;
8    old = FAA(blk.allocated, 1).off;
9    if (old >= BLOCK_SIZE)
10     return BLOCK_DONE;
11   return ALLOCATED(EntryDesc{.block=blk, .offset=old});
12 }
13 void BBQ<T>::commit_entry(EntryDesc e, T data){
14   e.block.entries[e.offset] = data;
15   ADD(e.block.committed, 1);
16 }
17 state BBQ<T>::advance_phead(Head ph) {
18   nblk = blocks[(ph.idx + 1) % BLOCK_NUM];
19   cons = LOAD(nblk.consumed);
20   if (cons.vsn < ph.vsn ||
21       (cons.vsn == ph.vsn && cons.off != BLOCK_SIZE)) {
22     reserved = LOAD(nblk.reserved);
23     if (reserved.off == cons.off) return NO_ENTRY;
24     else return NOT_AVAILABLE;
25   }
26   cmtd = LOAD(nblk.committed);
27   if (cmtd.vsn == ph.vsn && cmtd.off != BLOCK_SIZE)
28     return NOT_AVAILABLE;
29   MAX(nblk.committed, Cursor{.vsn=ph.vsn + 1});
30   MAX(nblk.allocated, Cursor{.vsn=ph.vsn + 1});
31   MAX(phead, ph + 1);
32   return SUCCESS;
33 }
34 class BBQ<T> {
35   shared<Head> phead, chead;
36   Block<T>[] blocks;
37 }
38 class Block<T> {
39   shared<Cursor> allocated, committed;
40   shared<Cursor> reserved, consumed;
41   T[] entries;
42 }
43 class EntryDesc {
44   Block block; Offset offset; Version version; }
```

retry-new mode | drop-old mode

Figure 2.4: Low-level details of BBQ (1).

```
45 <Head, Block> BBQ<T>::get_chead_and_block(){
46   ch = LOAD(chead);
47   return (ch, blocks[ch.idx]);
48 }
49 state BBQ<T>::reserve_entry(Block blk){
50 again:
51   reserved = LOAD(blk.reserved);
52   if (reserved.off < BLOCK_SIZE) {
53     committed = LOAD(blk.committed);
54     if (reserved.off == committed.off)
55       return NO_ENTRY;
56     if (committed.off != BLOCK_SIZE){
57       allocated = LOAD(blk.allocated);
58       if (allocated.off != committed.off)
59         return NOT_AVAILABLE;
60     }
61     if (MAX(blk.reserved, reserved + 1) == reserved)
62       return RESERVED((EntryDesc){.block=blk,
63         .offset=reserved.off, .version=reserved.vsn});
64     else goto again;
65   }
66   return BLOCK_DONE(reserved.vsn);
67 }
68 T BBQ<T>::consume_entry(EntryDesc e){
69   data = e.block.entries[e.offset];
70   ADD(e.block.consumed, 1);
71   allocated = LOAD(e.block.allocated);
72   if (allocated.vsn != e.version) return NULL;
73   return data;
74 }
75 bool BBQ<T>::advance_chead(Head ch, Version vsn){
76   nblk = blocks[(ch.idx + 1) % BLOCK_NUM];
77   committed = LOAD(nblk.committed);
78   if (committed.vsn != ch.vsn + 1)
79     return false;
80   MAX(nblk.consumed, Cursor{.vsn=ch.vsn + 1});
81   MAX(nblk.reserved, Cursor{.vsn=ch.vsn + 1});
82   if (committed.vsn < vsn + (ch.idx == 0))
83     return false;
84   MAX(nblk.reserved, Cursor{.vsn=committed.vsn});
85   MAX(chead, ch + 1);
86   return true;
87 }
```

retry-new mode | drop-old mode

Figure 2.5: Low-level details of BBQ (2).

0. We may omit the type when clear from the context.

Initially, `idx` and `off` in the first block are zero and for remaining blocks `off` is set to `BLOCK_SIZE`. The initial value of `vsn` will be introduced in §2.4.2.2.

Other types. `Block` has shared cursors, annotated with `shared<>`, and an array of entries of type `T` (line 38). `EntryDesc` is an entry descriptor; it points to a block and contains `offset` to location the actual entry and a `version` data-consistency checks used in drop-old mode (line 41). Finally, `BBQ` contains the shared heads and an array of `Block<T>`.

2.4.2 Operations

Enqueue and dequeue operations are divided into different cases: First, when the allocation in the enqueue or the reservation in the dequeue do not fail. Second, when enqueue or dequeue have to advance respective heads to the next block.

2.4.2.1 Successful allocation/reservation

The producer uses FAA to allocate an entry (line 8) and returns its location as `EntryDesc` if there is enough space in the current block (line 11). A pre-check (line 6) avoids endless increasing of `allocated` when the queue is full, which could cause FAA overflows and impact performance negatively. For the consumer, the entry is reserved through MAX[2] (line 61), which atomically sets a variable if the given value is greater than the variable's value and returns the old value. Consumers never pass producers (line 54) and can read when out-of-order commit are not ongoing in the same block, which means all allocated entries are committed (line 58).

[2]Unlike FAA, MAX provides conditional update semantics. Moreover, for some cases, MAX has similar semantics to CAS but better performance observed from experimental results. We use CAS and while loop to achieve the same functionality for architectures that do not support MAX such as x86 [87].

2.4.2.2 Advancing to the next block

Monotonic version updates. Head and cursor versions are initially zero. Both enqueue and dequeue calls start by reading the current cached head (`phead` and `chead`, respectively) into a local variable (`ph` and `ch` in Fig. 2.3). After failing to allocate or reserve an entry, these calls try to advance the respective phantom heads by calling advance_phead or advance_chead. These functions *try* to reset the cursors of the next block with the previously read version of the cached head plus one (lines 29, 30, 80, and 81 in Fig. 2.5). Subsequently, the functions *try* update the cached head itself (lines 31 and 85).

The reset of cursors and the update of cached head may not always succeed. Consider the following example. Two producers try to allocate entries at block B_0 and fail. Both have read `phead` with value $\{$`.vsn=0,.idx=0`$\}$. Now both call advance_phead concurrently and are at line 30. Producer P_1 stalls while producer P_2 succeeds updating the `allocated` cursor of block B_1. P_2 also allocates one or more entries such that now B_1's `allocated` has the value $\{$`.vsn=1,.off=16`$\}$. If now P_1 would be able to succeed resetting `allocated`, then the allocations of P_2 would be lost. Nevertheless, to avoid such ABA situations, the reset of cursors and update of cached head do not have to be performed with an expensive CAS. The recent MAX atomic instruction from Armv8.1-LSE can provide the required monotonicity.

Invariants. Producers have to ensure they advance `phead` only if the next block that has no unconsumed data. Consumers have to ensure they advance `chead` only if the next block has committed data.

We guarantee these invariants by ensuring that the version difference between phantom `phead` and phantom `chead` never exceeds 1. When producers advance `phead` and reset the `allocated` and `committed` cursors of the next block with version `ph.vsn+1` (line 30), the `consumed` cursor must have version `ph.vsn` (line 21). Similarly, when consumers advance `chead` and reset the `reserved` and `consumed` cursors of the next block with version `ch.vsn+1` (line 80), the `committed` cursor must have version `ch.vsn+1` (line 79).

Order matters. Often the order in which shared variables are accessed is crucial for correctness. For example, reading `reserved`, `committed`, and `allocated` variables (lines 51, 53, and 57) in a different order can cause the consumer to read garbage. Moreover, updating cached heads (lines 31 and 85) must happen after updating block-level variables (lines 29, 30, 80, and 81), otherwise blocks may be fully skipped.

To guarantee shared variables are accessed in the program order of Fig. 2.5 on architectures with weak memory models, C/C++ implementations of BBQ can employ atomic LOAD, STORE, MAX, FAA, and CAS instructions with sequentially consistent memory barriers (see C11/C++11 atomics [14]). In §2.5, we report a correct relaxation of these barriers.

2.4.3 Drop-old Mode

Unlike the retry-new mode, where producers cannot insert data when the queue is full, in drop-old mode, producers continue to write even if the data is not yet consumed. Consequently, producers no longer depend on the consumers to make progress. The FIFO property still holds, except that some data might be lost. In other words, entries are consumed in the order in which they were allocated, but some committed entries may not be consumed.

Speculative reads. Drop-old mode is widely used in profiling scenarios, where enqueue calls writing a log should not be delayed by dequeue calls that read the log. To reduce the chances of dequeue calls interfering with enqueue calls, consumers read data in a speculative fashion. The consumer first reads the data and then checks whether it has been overwritten. If so, it discards the data and tries reserving another entry.

From retry-new to drop-old mode. A few differences exist between retry-new and drop-old mode. First, producers avoid advancing to blocks that are still not fully committed in the previous round, returning `BUSY` (lines 27 and 28).

Second, consumers guarantee FIFO order by checking if the version of the next block is greater than or equal to the current one (line 82). If that is the

case, `reserved` is reset with the version of `committed` (line 84), indicating the block is ready to be read. The first block is a special case because, in contrast to other blocks, its version is always off-by-one. Therefore, we add 1 to the comparison if `ch.idx == 0`.

Third, the data-consistency check is based on the fact that a block is not overwritten as long as `allocated` and `reserved` versions are equal. Therefore, before reading data, we record the `reserved` version (line 63), and after copying the data from the entry, we check if corresponding `allocated` version still matches the `reserved` (line 72).

2.4.4 Variable-sized Entries

BBQ can support variable-sized entries with minor algorithmic changes. Each entry has an additional metadata `size` to support different entry sizes in one queue. Block-local cursors and `BLOCK_SIZE` indicate the space of entries instead of their number. MAX at line 61 is no longer sufficient; CAS must be used instead.

Dummy entry. Unlike the fixed-size version of BBQ, where entries can exactly fill up a block, here, the remaining space of a block might not be enough to contain the new entry. In such cases, we mark the space with a dummy entry and return `BLOCK_DONE` to trigger a retry in the next block. Since enqueue uses FAA, the producers that cause `allocated` go over the boundary marks the dummy entry by setting its `size` to zero and commits it. Consumers that read an entry with size zero ignore the dummy entry and retry in the next block. Upon reading the dummy entry, the consumer also sets `consumed` to be equal to `BLOCK_SIZE`.

2.4.5 Other Implementation Details

We have implemented BBQ in C and Java. We have also implemented a wrapper with the Java Native Interface (JNI) [45] to call the C version from Java.

Finally, we have optimized BBQ for SPSC scenarios as follows: (1) `phead` and `chead` are no longer shared variables and can be accessed with non-atomic loads/stores. (2) `allocated` and `committed` (resp. `reserved` and `consumers`) are merged into one variable and updated with STORE.

2.5 Verification and Optimization of BBQ

Concurrent data structures are complicated beasts and are easy to get wrong [80]. To increase confidence in our C implementation and find intricate bugs, we generate a series of small hand-crafted tests that can trigger corner cases in the algorithm and then use VSync [165], an extension to the GenMC model checker [135]. The tool generates all executions of the algorithm on those tests, including executions that can only happen on WMMs, exercising the following critical corner cases: (1) queue full or empty, (2) FIFO, (3) wrap-around, and (4) termination of bounded loops with bounded effect [165, 139].

Bugs. We found three concurrency bugs in an earlier version of the drop-old mode of BBQ.

1. A test revealed a bug in which enqueue operations incorrectly returned `BUSY`. The block was detected as `NOT_AVAILABLE` because, in that version, the condition at line 27 was `committed.off == BLOCK_SIZE && committed.vsn == ph.vsn`. Therefore, even if other producers reset the next block and have the space to allocate, the block would still be `NOT_AVAILABLE`. That violated linearizability.
2. We have found a termination bug in which the checking in line 82 was written as `blk.committed.vsn >= nblk.committed.vsn`, missing the special case of the version number in the first block, which may let consumers advancing the block forever if the queue is empty and all blocks happen to have the same version number.
3. The wrap-around test revealed a bug due to a missing fence, where readers could return incorrect data when a fast writer overwrote the entry they were currently reading.

We found these bugs through the verification with model checking. They were not found during stress testing, nor by running the small test cases directly on hardware. However, we could retrospectively construct test cases that reproduce these bugs on real hardware. Concurrent algorithms, especially those using complicated synchronization such as drop-old mode, are hard to get right using traditional methods.

Barrier optimization. We used VSync to run the memory barrier optimization for WMMs. The results consistent with the order analyze of reading/updating shared variables in §2.4.2.2. For the fixed entry size version, 14 atomic instructions with full memory barriers are optimized to 3 release barriers, 3 acquire barriers, and 8 relaxed barriers, respectively.

2.6 Evaluation

2.6.1 Environment Setup

Hardware. All of our experiments are performed on three x86 machines with 88, 96, and 12 hyperthreads, respectively (denoted as x86-88T, x86-96T, and x86-12T), and an ARM machine with 96 cores (arm-96T). x86-88T and x86-96T are connected through two 10Gbps links.

Software. On these servers, we installed Ubuntu 20.04.3 LTS, with Linux kernel 5.4.0. We use Linux perf [62] to get results of L1 cache misses, the version of it is the same with the Linux kernel. Java-based experiments use JDK v11 [44].

2.6.2 Microbenchmarks

Workloads. We have the following 3 workloads for microbenchmarks implemented in C/C++ and Java:

- simple: Each producer or consumer has its own thread, where they keep executing enqueue or dequeue operations in a loop. Data is validated after each dequeue.

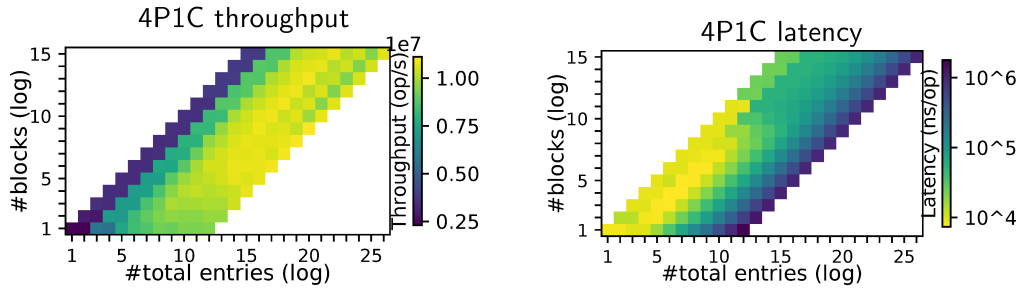


Figure 2.6: BBQ throughput and latency varying number of blocks and entries (x86-88T).

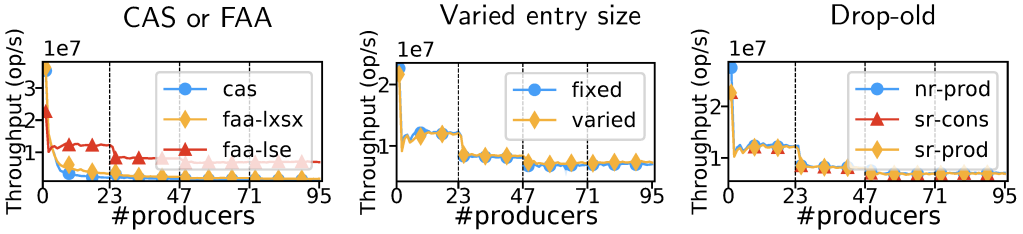


Figure 2.7: BBQ throughput with CAS and FAA; with support for variable-sized entries; and with drop-old mode (arm-96T).

- complex: Based on the simple workload. Producers and consumers allocate space for data, preform enqueue and dequeue then manually free (C/C++ version) or let JVM garbage collection it [182] (Java version). Additionally, each operation also performs a deterministic random busy-loop of at most one hundred `nop` instructions.
- profiling: Based on the simple workload. The throughput of producers and consumers is fixed at $10k$op/s and $1k$op/s, respectively.

Thread affinity. For MPSC or SPMC scenarios, we assign a single producer or consumer at the first core/hyperthread and then distribute the other threads sequentially to cores/hyperthreads. For MPMC, we assign producers and consumers interleaved one by one; if their number differs, the surplus is assigned at the end.

Experiments. We perform the following experiments, each measuring a different metric:

- throughput: Total number of consumed entries per second.
- data-latency: Average time each data stays in the queue.

- op-latency: Average latency of each enqueue or dequeue operation.
- cache-miss: Average number of L1 cache misses per consumed entry, measured with Linux `perf`.
- fairness: Throughput of each producer and consumer (only for MPSC and SPMC).
- full/empty: Latency of enqueue when the queue becomes full and latency of dequeue when the queue becomes empty (only used with simple workload).
- oversubscription: Throughput with more producers and consumers than than cores/hyperthreads.

Each experiment runs 3 times. If not specified otherwise, solid lines represent average results; shaded area represents standard deviation; and vertical dashed lines indicate when threads cross NUMA nodes, are assigned to hyperthreads in the same core, or are oversubscribed.

Configuration. The data size is always 8 bytes, a size all queues can support. For the data-latency experiment, the number of entries is around 128. For the other experiments, the buffer size is $32k$ bytes unless specified otherwise.

2.6.2.1 BBQ Parameters and Design Choices

We start by evaluating parameters and design choices of BBQ.

Configuring the number of blocks. Figure 2.6 shows throughput and data-latency experiments for BBQ with four producers and one consumer. The color scale shows the existing trade-off between number of entries and number of blocks; users have to be aware of that when choosing the buffer size and number of blocks. We use the following heuristic function to determine the number of blocks in all rest experiments: *number of blocks (log)* $= \max(1, \lfloor$*number of entries (log)*$/4 \rfloor)$.

Performance impact of FAA. Figure 2.7 shows the results of an MPSC throughput experiment on our Arm machine. BBQ is configured to use FAA instruction from Armv8.1 LSE, standard FAA and CAS implemented with load-exclusive and store-exclusive instructions. Except for the 1 thread case, LSE-

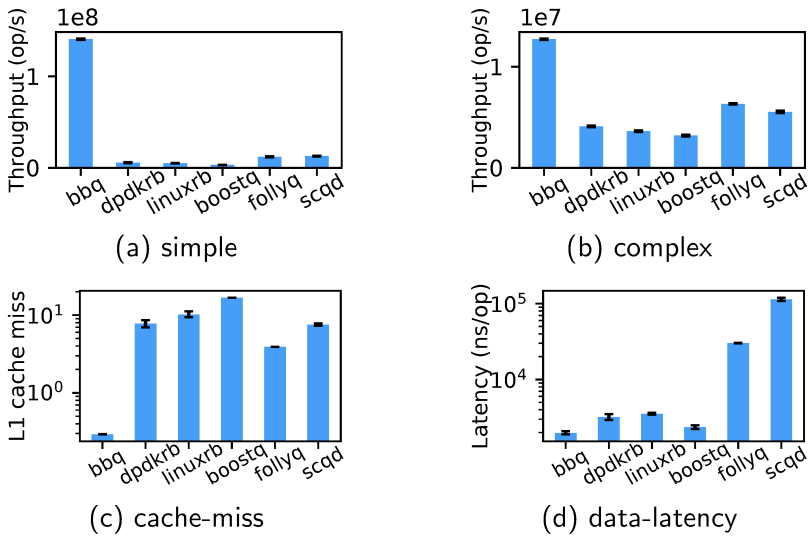


Figure 2.8: SPSC comparison of BBQ against state-of-the-art on x86-88T.

based FAA shows the best scalability, outperforming the other two by at least 5 times.

Support for variable-sized entries. Figure 2.7 also shows the throughput of the BBQ with fixed- and variable-sized entries. The size of each data is the same, yet the varied entry version has to store additional size information for every entry. Nevertheless, the throughput difference between both is negligible.

Consumer-producer interference in drop-old mode. Finally, Figure 2.7 shows the throughput of BBQ with drop-old mode in two configurations: MPSC and MPNC (multiple-producer/no-consumer). The throughput of the producers (nr-prod) with no consumers is less than 8% higher than with consumer (sr-prod). Moreover, the consumers manage to consume at least 99.97% of the entries except for the 1 thread case (sr-cons). These results illustrate that consumers with the speculative-read method incur a rather minor interference on producers — please refer to §2.6.2.3 for a baseline with existing implementations.

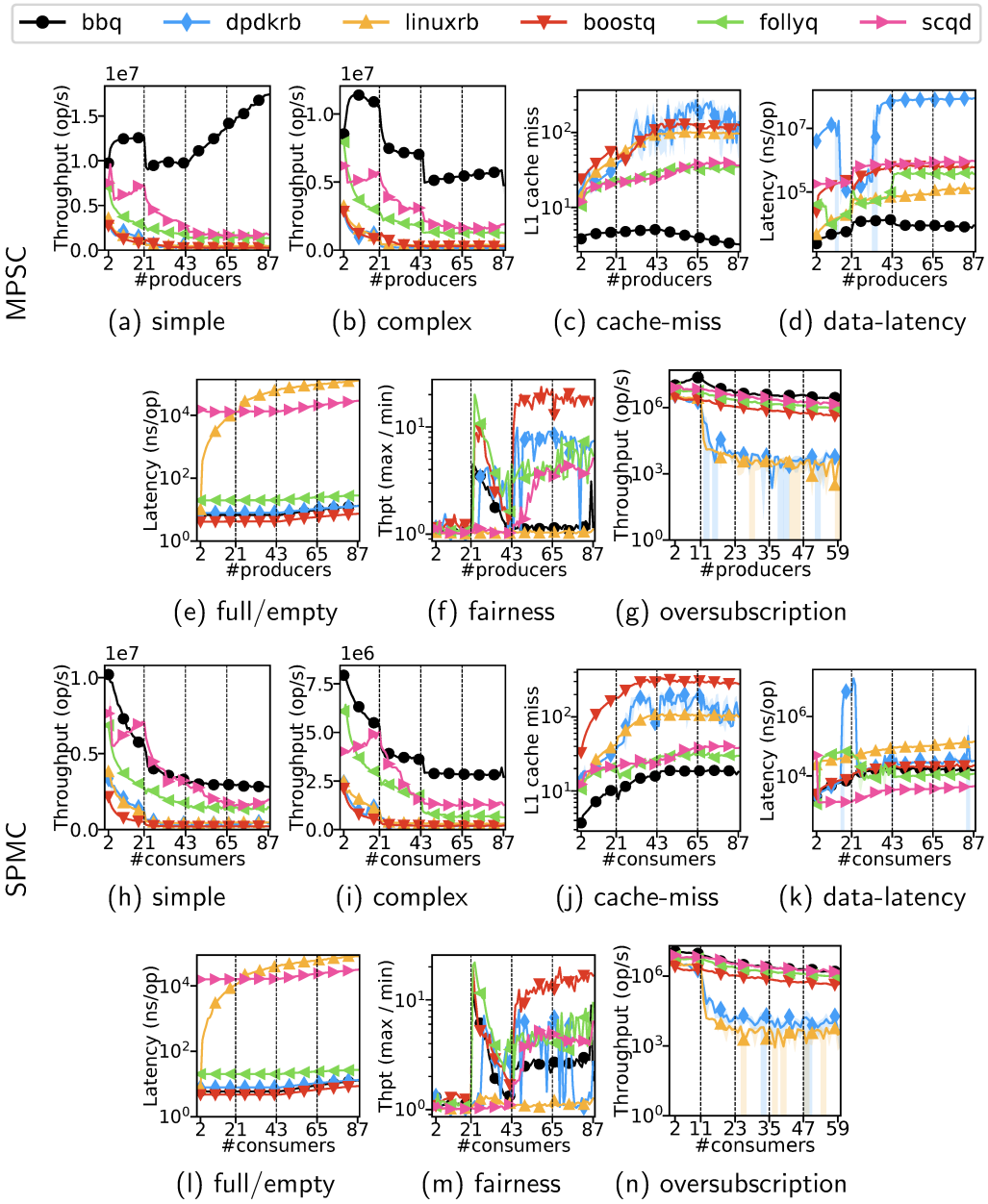


Figure 2.9: MPSC and SPMC comparison of BBQ against state-of-the-art on x86-88T (and x86-12T for oversubscription).

2.6.2.2 State-of-the-art Comparison: Retry-new Mode

We now compare BBQ against 5 state-of-the-art bounded queues: dpdkrb, DPDK ringbuffer v21.08 [18]; scqd, a lock-free bounded queue [161]; linuxrb, the ringbuffer in the Linux kernel v5.16 [51]; boostq, the bounded queue in C++ Boost libraries v1.71 [12]; and follyq, the bounded queue (with total

method) [60] in Meta's open-source Folly library v2021.11.8. For dpdkrb and follyq, we use their SPSC versions to run corresponding SPSC experiments.

Effectiveness of the Block-based Approach. To isolate the effect of the blocks, we first focus on SPSC experiments because BBQ do not profit from FAA in such scenarios. Figure 2.8 shows that BBQ greatly outperforms all other bounded queues in all experiments. For the simple workload, BBQ yields 11.3x to 42.4x higher throughput than other libraries. The throughput of BBQ is $1.41 \cdot 10^8$ op/s, while the second-best one follyq is $1.24 \cdot 10^7$ op/s. For the complex workload, which has a random busy-loop to limit the maximum throughput, BBQ still outperforms follyq by 2x. BBQ's better performance is mainly due to the massive decrease in L1 cache misses with the block-based approach (notice the y-axis log scale).

Throughput in MPSC and SPMC scenarios. Figures 2.9a, 2.9b, 2.9h, and 2.9i show BBQ performing on par or better than other queues in the simple and complex workloads. For MPSC scenarios, BBQ performs up to 10.13x and 3.65x faster than the second-best queue, respectively. For SPMC scenarios, BBQ performs up to 1.88x and 2.39x faster than the second-best queue, respectively.

The throughput difference between MPSC and SPMC results can be attributed in part to the different L1 cache misses measurements (see Fig. 2.9c and Fig. 2.9j). BBQ consumers employ CAS operations in every dequeue, and these can fail and have to be retried, each time suffering another cache miss.

Data latency. We measure the average time data stays in the queue in the complex workload, as shown in Fig. 2.9d and Fig. 2.9k. For MPSC case, BBQ performs consistently better than other bounded queues; up to 17.22x lower latency than the second-best queue. For the SPMC case, `scqd` performs best, up to 7.45x lower latency than BBQ. That is an artifact of the delayed invalidation trick (see §2.2): Once the queue is empty (C.head = P.tail), consumers invalidate the entries pointed by C.head after a delay. Since consumers first increment C.head and then wait, multiple consumers will be pending on different entries. As soon as the producer commits a new entry, one consumer aborts its

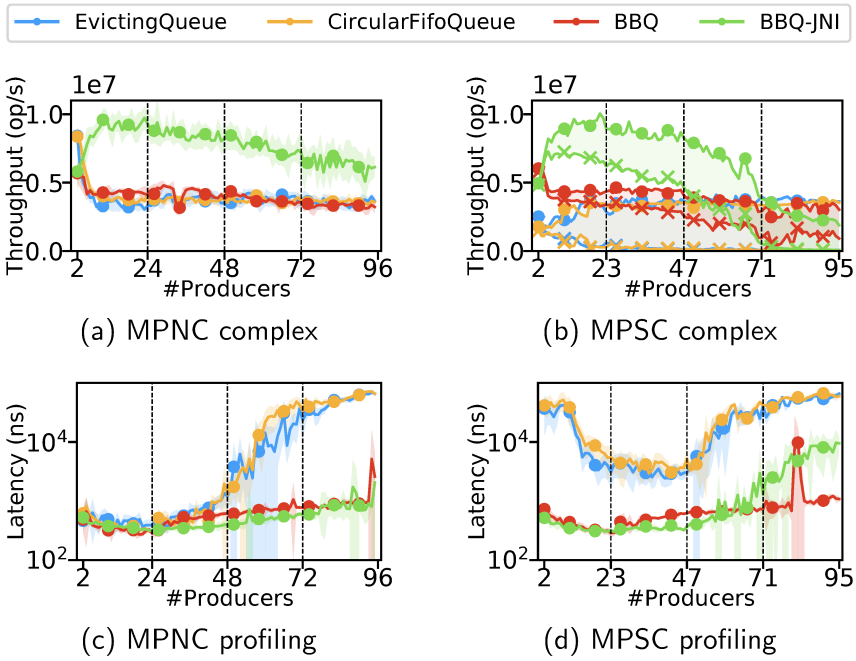


Figure 2.10: Cross comparison results for drop old mode on arm-96T.

delay and immediately returns the data.

Full and empty queues. Figures 2.9e and 2.9l shows the latency for failed enqueue on a full queue (top figure), and failed dequeue on an empty queue (bottom figure). In such scenarios, the delay invalidation of scqd incurs a high cost: the latency of failed operations in scqd is around 1000x higher than in most other queues. For linuxrb, the latency increases with the number of producers/consumers due to its coarse-grained locking.

Fairness between producers or consumers. Figures 2.9f and 2.9m shows the relation between maximum and minimum throughput of producers (top figure) and consumers (bottom figure). linuxrb provides exceptional fairness because it relies on a fair spinlock[3]. Other queues show unfair throughput after crossing the first NUMA node (at 22 producers/consumers) except for scqd, which becomes unfair when hyperthreads of the same cores start being used (at 44 producers/consumers).

Oversubscription effects. Figures 2.9g and 2.9n shows the results of our

[3]In our userspace port of linuxrb, we employ a ticket lock.

oversubscription experiment on x86-12T with up to 5x more threads than hyperthreads. Both dpdkrb and linuxrb are highly affected by oversubscription; the former due to their in-order policy (see §2.2), the latter due to its coarse-grained locking. Under oversubscription (*i.e.*, with more than 12 threads), BBQ outperforms the second-best queue by a small margin: 2.22x in MPSC and 1.23x in SPMC scenarios.

2.6.2.3 State-of-the-art Comparison: Drop-old Mode

We now compare BBQ with other two bounded queues that support overwriting old values, namely EvictingQueue from Google Core Libraries Guava [35], and CircularFifoQueue from Apache Commons [6]. The experiments are conducted on the arm-96T machine.

Producer performance. Figure 2.10a shows the enqueue throughput with *no consumers* for the complex workload. On the one hand, BBQ-JNI yields 3.2x higher enqueue throughput than EvictingQueue and CircularFifoQueue. On the other hand, BBQ yields an enqueue throughput rather similar to them. Intuitively, BBQ-JNI has a better performance since employs real FAA instructions, whereas, in the Java version of BBQ, the JVM translates FAA into CAS [128].

Figure 2.10c shows the enqueue latency, again with *no consumers*, for the profiling workload. Remember that producers issue $10k$ enqueue calls per second in the profile workload. With BBQ and BBQ-JNI, the enqueue latency slowly increases: 147.9ns and 176.4ns with 1 thread, respectively, to 965.6ns and 914.3ns with 94 threads, respectively. Up to 44 producers, EvictingQueue and CircularFifoQueue perform similar to BBQ variants. With more than 44 producers, however, their enqueue latency quickly increases up to 70 μs (72x higher than BBQ). From Fig. 2.10a, we know that their maximum enqueue throughput is about 450$k$op/s. Hence, these queues already reached throughput limit with 44 producers, and any additional producers can only increase the latency. We believe the spike at 95 threads (BBQ with 5.1μs and BBQ-JNI with 2.0μs) may caused by garbage collection, but further investigation is necessary.

Enq-deq interference. We now introduce a single consumer to understand the interference of dequeue on the enqueue operations. Ideally, the enqueue operations should incur a small overhead (latency) to the profiled program; and this overhead should be minimally affected by concurrent dequeue calls. Moreover, if enqueue calls interfere with dequeue calls too frequently, more data may be dropped, *i.e.*, overwritten before being consumed.

Figure 2.10b shows the enqueue and dequeue throughput (marked with • and $\times$, respectively) for the complex workload. Comparing Figs. 2.10a and 2.10b, we observe that the enqueue throughput of BBQ is similar in both figures and of BBQ-JNI is similar up to 47 threads, but after that it drops to about $1.89 \cdot 10^6$ op/s. The enqueue throughput of EvictingQueue and CircularFifoQueue is initially lower when a consumer is concurrently calling dequeue. The reason for this lower enqueue throughput can explained by observing the difference between enqueue and dequeue in Fig. 2.10b.

First, note that with more than a few producers, the enqueue and dequeue throughput of each queue do not match, *i.e.*, the consumer is not fast enough to read out all the data before the producers start overwriting the oldest entries. Also note that the more the enqueue throughput of EvictingQueue and CircularFifoQueue recovers (by increasing producers), the lower is the dequeue throughput. Once their throughput is back to the level of Fig. 2.10a with 15 threads, their dequeue throughput is no more than $4.92 \cdot 10^5$ op/s. In contrast, BBQ and BBQ-JNI sustain a much higher dequeue throughput up to 46 threads ($2.89 \cdot 10^6$ op/s and $5.07 \cdot 10^6$ op/s, respectively).

Figure 2.10d shows the enqueue latency for the profiling workload. BBQ and BBQ-JNI provide enqueue latencies varying from 730.6ns and 519.9ns with 2 producers, respectively, up to 1082ns and 9503ns with 95 producers — we ignore the noisy region with 81 producers. Comparing the results of Figure 2.10c and Figure 2.10d reveal that the enqueue latency of EvictingQueue and CircularFifoQueue, for example with 8 producers increase by 124.97 times when adding a single consumer with a relatively low dequeue frequency.

The latency increases as well as the throughput decreases of BBQ-JNI after

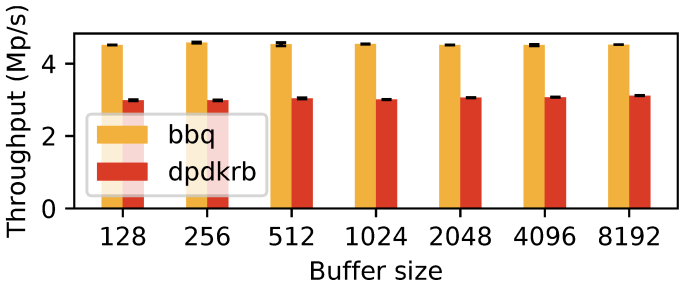


Figure 2.11: Throughput comparison between BBQ or DPDK ring buffer.

47 producers could be related to the JNI overhead of calling C code from Java.

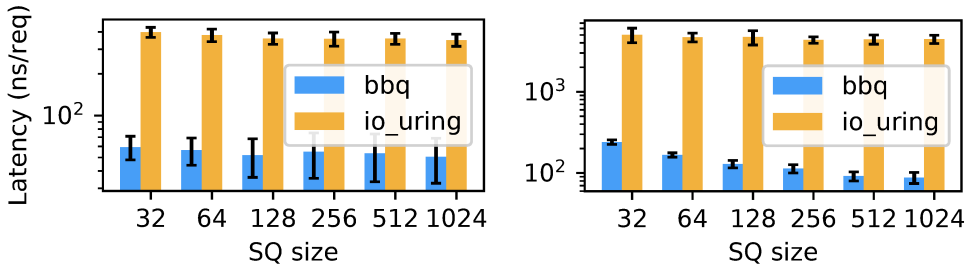
2.6.3 Macrobenchmarks

We now explore three benchmarks that represent the real-world usage queues.

2.6.3.1 DPDK's End-to-end Benchmark

We replace the ring buffer in DPDK's event library [21] and network driver [20] with BBQ, and run the multiprocess benchmark [17] from the DPDK Test Suite [19] (DTS). The benchmarks consists of one server process receiving and distributing packets, and two client processes performing level-2 packet forwarding [16]. These processes run on the device under test (DUT), our x86-88T machine. The tester and traffic generator TRex [79] run on our x86-96T machine. The packet size is 64 bytes (along with the UDP header) as well as the entry size of the queue. The versions of DPDK, DTS, and TRex are 21.08, 21.02, and 2.92, respectively. We report the end-to-end throughput (in million packets per second) measured by the traffic generator.

Figure 2.11 shows our experimental results. BBQ provides 1.5x higher throughput with different buffer sizes in the driver. We observed no further improvements with larger buffer sizes, indicating that the ring buffer may not be a bottleneck any more. We also replaced the so-called software queue in the multiprocess benchmark, and observed no improvement.



(a) Batchsize = 1 (b) Batchsize ∈ [1, 32]

Figure 2.12: Latency per request comparison of BBQ and Linux io_uring on x86-88T.

2.6.3.2 **Linux** io_uring

Linux io_uring [23] is a new asynchronous I/O [97] API for kernel-user space communication. It consists of two ring buffers, one for request submissions (SQ) and another for completion confirmations (CQ). It supports batched submission and batched confirmations with configurable batch sizes [43].

We port io_uring from Linux kernel (v5.14-rc6) [42] to userspace, omitting I/O related functionality and replacing its ring buffers with BBQ. To avoid unstable results, we disable the option of overflowing entries into an additional linked list. We set the CQ size to twice the SQ size as recommended [50]. Our benchmark runs three threads: The first submits request batches (via SQ); the second (representing the kernel) consumes them and immediately produces confirmations (via CQ); and the third consumes the confirmation batches. We configure submission and confirmation batches with size 1 or with a random value from 1 to 32. Each experiment runs 10 times, measuring the time to submit 1M requests.

Figure 2.12 shows a significant improvement of the latency per request when using BBQ. For example, with batch size of 1 and SQ size of 32 and 1024, BBQ yields 6.7x and 6.9x lower latency than the original ring buffer, respectively. For random batch size and the same SQ sizes, BBQ yields even lower latencies: 20.9x and 50.5x, respectively.

2.6.3.3 LMAX Disruptor Benchmarks

Disruptor [53] is concurrency mechanism used for high-performance financial exchange. Its core component is a ring buffer. We compare its throughput with the Java and JNI versions of BBQ with three official Disruptor benchmarks: `OneToOneThroughputTest`, `ThreeToOneThroughputTest`, and `OneToThreeThroughputTest`. We modify these benchmarks to support not just three, but more producers or consumers. Apart from this modification, all other parameters (*e.g.*, number of iterations, sleep time between operations, number of repetitions) are unchanged.

Disruptor can randomly change the batch size based on the workload. To make the comparison as fair as possible, we first run the benchmark with Disruptor to get the average batch size used, and then run BBQ with that batch size. Figure 2.13 shows the throughput of Disruptor, BBQ, and the baseline Java queue (`java.util.concurrent.BlockingQueue`) for several scenarios. The number on each bar refers to the (average) batch size, and the label pPcC indicates the number of producers (p) and consumers (c).

In the 1P1C scenario, Disruptor yields almost 3x higher throughput than the Java queue (3 Mop/s versus 1.3 Mop/s). BBQ and BBQ-JNI, however, yield an order of magnitude higher throughput (14.1 Mop/s and 12.1 Mop/s, respectively). The higher performance of BBQ over BBQ-JNI is due to the JNI call overheads. With 8 producers, the difference between Disruptor and BBQ is lower (2.2 Mop/s and 3.7 Mop/s, respectively). BBQ-JNI yields 3x Disruptor's throughput (6.7 Mop/s) due to its use of FAA. With 32 producers, BBQ and BBQ-JNI again outperform Disruptor by an order of magnitude (3.0 Mop/s, 3.3 Mop/s, and 0.6 Mop/s, respectively).

With a single producer and multiple consumers, BBQ-JNI has no opportunity to gain performance by using FAA. Due to that, its performance pays the penalty of the JNI call overheads. Nevertheless, BBQ still outperforms Disruptor in most configurations. For example, BBQ yields 1.23x higher throughput than Disruptor with 2 consumers (1P2C); and 1.68x higher throughput with 8

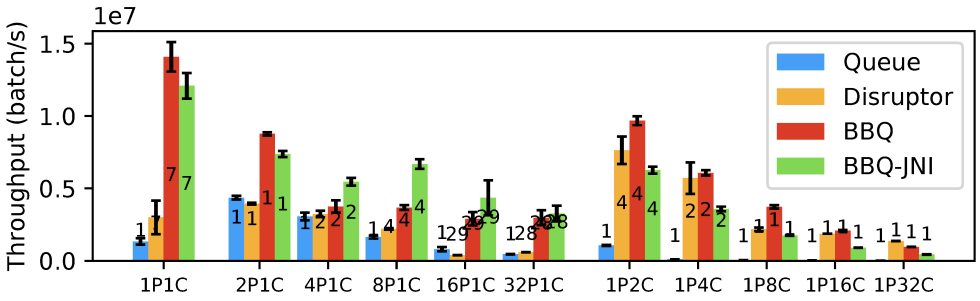


Figure 2.13: Throughput comparison of BBQ and BBQ-JNI against LMAX Disruptor on x86-88T.

consumers. With 32 consumers, Disruptor yields 1.42x higher throughput than BBQ.

2.7 Summary

In this chapter, we presented BBQ, a novel ringbuffer design that dramatically reduces the enq-deq interference by splitting the entire ringbuffer into multiple blocks. BBQ is applicable to a large spectrum of scenarios, from exchanging data to profiling, with single or multiple producers/consumers, sending fixed- or variable-sized entries, among others. Our experimental results show that BBQ outperforms several industrial ringbuffers (*e.g.*, DPDK, LMAX Disruptor, Linux io_uring, Meta's Folly queue) in the great majority of workloads.

To support modern architectures such as Armv8.1, we verified and optimized BBQ with a model checker for weak memory models. Even though far from sound, verification with model checkers has proven a valuable, low-cost method of catching bugs.

Chapter 3

CNQ: A Concurrent Nested Queue

3.1 Overview

Concurrent queues play a pivotal role in various domains, including operating systems and networking, enabling data management, synchronization, coordination, and component decoupling. While strict adherence to the First-In-First-Out (FIFO) specification may not always be necessary, existing approaches that adopt weaker specifications struggle to leverage the scalable Fetch-and-Add (FAA) instruction, limiting their ability to surpass the performance of modern FIFO queues that utilize it.

We introduces Concurrent Nested Queue (CNQ), a multiple-producer, multiple-consumer (MPMC) queue where each entry does not directly store data but represents a single-producer, single-consumer FIFO queue (capsule) capable of holding multiple data items. By leveraging existing MPMC FIFO queues, CNQ inherits their scalable FAA instruction. Additionally, CNQ provides a k-FIFO ordering guarantee by constraining the distance between the oldest and newest capsule being used. Through benchmarking experiments with k set to 10^3, the results demonstrate that when integrated into state-of-the-art MPMC FIFO queues, CNQ exhibits a speedup of up to 51x. Moreover, compared to other

queues with weak specifications, CNQ achieves up to 8.3x higher throughput. We have formally verified CNQ on weak memory models with a model-checking-based framework.

The rest of this chapter is organized as follows. §3.2 introduces related works and their limitations. §3.3 provides an overview of the background before presenting our design and properties for CNQ in §3.4 and §3.5. Implementation details are discussed in §3.6. §3.7 focuses on how to integrate CNQ into MPMC FIFO queues. §3.8 presents the results of verifying CNQ. In §3.9, we experimentally compare the performance of CNQ with various FIFO queues and relaxed queues. Finally, §3.10 concludes our work.

3.2 Related Work

Batching. One straightforward approach for reducing interference between operations of queues is batching [154, 85], where producers (or consumers) enqueue (or dequeue) multiple items at once. However, this method leads to a significant increase in latency for the initial data that arrives in each batch, especially when the data flow is inconsistent (**P2**). This occurs because the consumer cannot access the existing data until the batch is full, as the producer transfers the data in a coarse-grained manner, treating each batch as a unit. Reducing the batch size to mitigate the issue will increase interference, which contradicts the original intention of using batching.

Random enqueue and dequeue. The random method allows producers and consumers to randomly select their entries, thereby reducing the likelihood of interference. A feasible implementation [131] involves accessing a slot containing multiple entries for each enqueue or dequeue operation. The entry selection process starts by randomly choosing one entry from the slot. If the selected entry is already in use, the producer or consumer scans the slot to find an available entry. When the current slot is fully utilized, a new slot is activated. Indeed, the random method effectively reduces interference among threads, which is advantageous. However, it is important to note that the performance improvement

```
1  Bool enqueue(data){
2    e = allocate();
3    if (!e) return false;
4    put(e, data);
5    produce(e);
6    return true;
7  }
```

```
8  T dequeue(){
9    e = reserve();
10   if (!e) return null;
11   T data = get(e);
12   consume(e);
13   return data;
14 }
```

Figure 3.1: enqueue and dequeue operations of the three-phase queue.

achieved from this approach is limited due to a trade-off [131]. Specifically, when larger slots are used to reduce interference, it results in longer search times to find an available entry when the slot is nearly full or empty (**P3**).

Distributed queues. Another approach involves the utilization of distributed queues. In this approach, a single queue is divided into multiple queues, each with its own entries and metadata. A naive solution would be to statically assign queues to specific producers and consumers. However, this approach is limited to scenarios where the number of producers and consumers differs (**P4**). Additionally, the binding method may lead to a significant number of wasted entries when queues are bounded (**P5**): when a producer fills up one queue, it must report a full status directly, disregarding the potential space available in other queues. These issues can be mitigated by an alternative approach where enqueue and dequeue operations are distributed among multiple queues managed by a load balancer. However, the load balancer itself could become a bottleneck (**P6**), particularly in scenarios where producers or consumers have varying throughputs. Furthermore, this approach requires each queue to be MPMC competitive (**P7**), which may result in noticeably slower performance compared to SPSC queues [184, 86].

3.3 Preliminaries

3.3.1 Three-phase Queue

MPMC FIFO queues typically involve three phases in enqueue and dequeue operations [85, 161, 184]. As shown in Fig. 3.1, when a producer intends to

```
Class rb<T> {AtomicInt phead,ptail,chead,ctail; T[] array; Int SZ;}
```

```
1  Int rb::allocate() {
2  again:
3    ph = LOAD(phead);
4    if (ph ≥ LOAD(ctail) + SZ)
5      return null;
6    if (!CAS(phead, ph, ph+1))
7      goto again;
8    return ph;
9  }
10 Void rb::put(Int ph, T data) {
11   array[ph % SZ] = data;
12 }
13 Void rb::produce(Int ph) {
14   while (LOAD(ptail) != ph);
15   STORE(ptail, ph+1);
16 }
```

```
17 Int rb::reserve() {
18 again:
19   ch = LOAD(chead);
20   if (ch ≥ LOAD(ptail))
21     return null;
22   if (!CAS(chead, ch, ch+1))
23     goto again;
24   return ch;
25 }
26 T rb::get(Int ch)
27   return array[ch % SZ];
28 }
29 Void rb::consume(Int ch) {
30   while (LOAD(ctail) != ch);
31   STORE(ctail, ch+1);
32 }
```

Figure 3.2: A implementation of the three-phase queue.

enqueue data, it calls the *allocate* function (line 2) to acquire the next available entry e while notifying other producers of its exclusive access. Upon successful allocation, the producer *put*s the data and notifies consumers that the entry is ready to be read by invoking the *produce* function (line 5). Similarly, a consumer utilizes the *reserve* function to obtain exclusive access to an entry. Once successfully reserved, the consumer *get*s the data from the entry and calls the *consume* function to inform producers that the entry can be reused.

Figure 3.2 presents an implementation of the three-phase queues, which is extracted from the ring buffer in DPDK [18, 184]. Primitives such as LOAD, STORE, and CAS are used to provide atomic semantics for reading, writing, and compare-and-swap operations on shared metadata denoted by the type `AtomicInt`. The entry `e` is instantiated as indexes of the data array (`ph` and `ch`). Producers and consumers ensure exclusive access to the entry by utilizing CAS (lines 6 and 22) on their respective head variables (`phead` and `chead`), and they notify each other when the entry is ready by updating their tail variables (`ptail` and `ctail`).

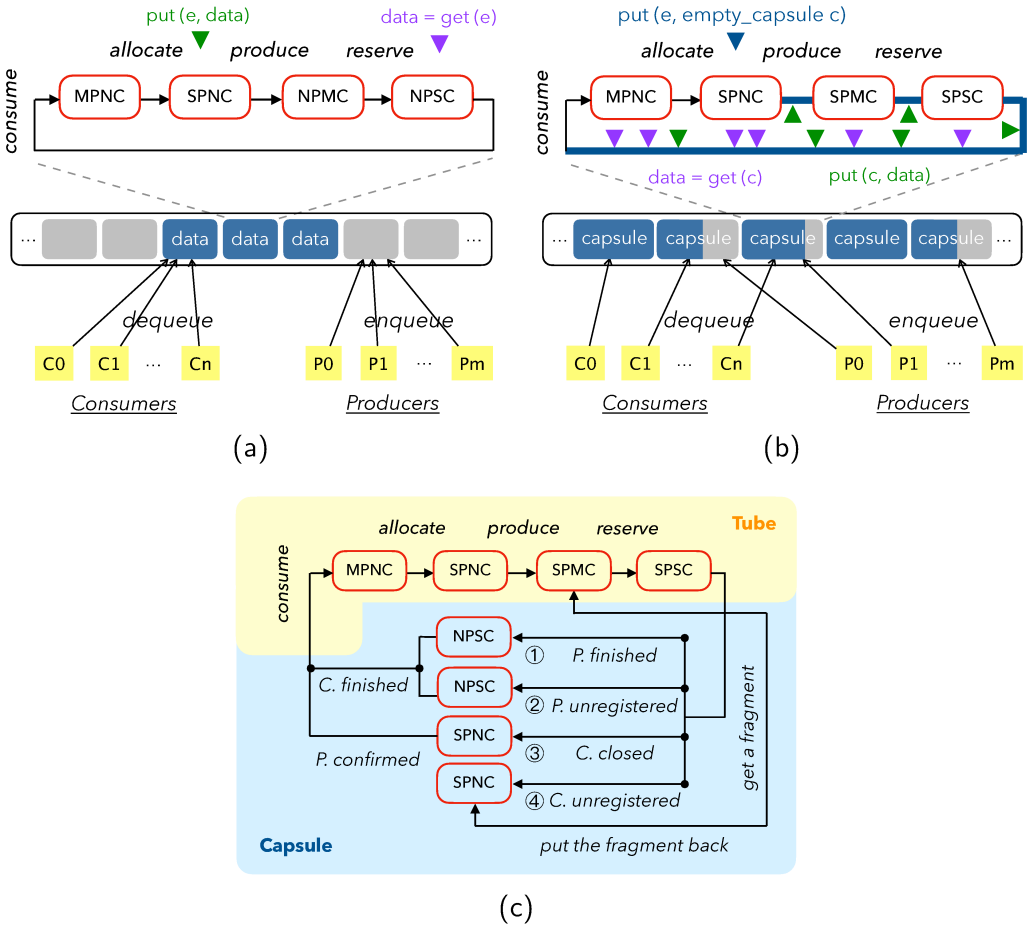


Figure 3.3: Ownership of entries for three-phase queues and CNQ.

3.3.2 Queue as Ownership Controller

In §3.3.1, we introduced how producers and consumers manage the ownership of queue entries by updating metadata in multiple phases, focusing on the perspective of operations. Now, let us proceed to examine this ownership control from the entry's perspective. Figure 3.3a illustrates the process of ownership transfer for an entry within a three-phase queue. Initially, the entry is collectively owned by all producers, as indicated by the MPNC label (multiple-producer, no-consumer). When a producer invokes the *allocate* function, ownership of the entry is transferred exclusively to that producer, represented by the SPNC label. Once the *produce* function is called, ownership of the entry is then transferred to all consumers, denoted by the NPMC label (no-producer, multiple-consumer).

Through a successful *reserve* call from a consumer, ownership is subsequently transferred to that consumer, indicated by the NPSC label. Finally, after the *consume* function is invoked, the entry becomes available for reuse by all producers, marked as MPNC.

3.4 Design

3.4.1 Ownership Sharing

As previously mentioned, batching is an effective approach for reducing interference between operations. However, it synchronizes data in a coarse-grained manner, which may not be suitable for all scenarios. To address this, a modification to the ownership control mechanism can enable fine-grained data transfer through *ownership sharing*. Figure 3.3b illustrates this concept. When the *produce* function is invoked, the producer does not transfer the ownership of the entry to the consumer; instead, it shares ownership of the entry with the consumer, indicated by the SPMC label. Once the entry is reserved, it is jointly owned by a producer and a consumer, denoted as SPSC. Prior to sharing the ownership, the producer initializes the entry as an empty capsule, specifically an SPSC FIFO queue. Afterward, the producer and the consumer who own this capsule can put multiple items into it until they invoke the *consume* function.

Through the aforementioned approach, the queue has now evolved into a nested design—an MPMC FIFO queue (referred to as the "tube") where each entry does not directly store data but serves as an SPSC FIFO queue (referred to as the "capsule"), capable of holding multiple data items. This novel design offers several advantages over state-of-the-art solutions:

- Compared to random and distributed queues that are hard to benefit from FAA due to the loss of strict sequentiality (P1), our composition approach integrates the capsule into state-of-the-art FIFO queues that utilize FAA, thereby enabling CNQ to take advantage of it.
- In comparison to batching, for CNQ, the consumer gains the ability to access

and read data from the capsule (similar to a batch in the batching approach) before the producer has completed writing the entire capsule. Consequently, it effectively resolves the issue of long latency that is commonly associated with the batching approach (P2).

- Unlike random queues, CNQ does not exhibit a trade-off on k (P3). On the contrary, a larger k results in improved performance (§3.9.1) due to the increased size of each capsule, which reduces contention on the tube.
- In comparison to the binding solution of distributed queues, CNQ binds them to capsules, providing a flexible solution that effectively addresses the limitations of specific scenarios (P4). Additionally, the size of each capsule is considerably smaller than the size of each queue in a distributed queue, effectively mitigating the waste of entries (P5).
- Compared to the load balancer solution that requires MPMC competition for each queue (P7), the capsule of CNQ only requires being SPSC compatible. Moreover, in scenarios with producers having varying throughput, load balancing is automatically ensured by allowing the producer with higher throughput to allocate more capsules, without the need for a load balancer that could potentially become a bottleneck (P6).

Overall, our approach not only improves performance but also provides solutions to various issues associated with other relaxed approaches, making it a versatile and efficient choice for a range of scenarios.

3.4.2 Life Cycle of the Capsule

Once the capsule is initialized as an SPSC queue, it becomes jointly owned by both a producer and a consumer. As a result, the invocation of the *consume* function cannot be solely initiated by the consumer as it was before. Instead, it can only be called when both the producer and the consumer mutually agree that the writing and reading processes for the capsule are complete. The coordination for this is managed by the capsule itself, as illustrated in Fig. 3.3c.

Case 1: capsule fully utilized. In the case that the producer completes

writing it, and then the consumer completes reading it. They proceed to their respective next capsules and exit the current one.

Case 2: producer unregistered. The producer has the option to *unregister* itself and exit the capsule prematurely if there is no additional data to be written. Consequently, the capsule remains partially filled. In this situation, the consumer reads the available data and then exits as well. The capsule is reused in the next round (i.e., wrap around) rather than allowing other producers to continue writing on the unused entries.

Case 3: capsule closed. In addition to cases where producers voluntarily leave the capsule, they may also be evicted due to slow writing speeds. This eviction is initiated by consumers to ensure that they are not blocked from reading data that has already been written in subsequent capsules, while the current one is causing a blockage. Once the capsule is closed, the consumer proceeds to the next one without the need to wait for the producer's response. Producers must be aware of this to prevent writing data to a closed capsule. Similar to case 2, the closed capsule will be reused in the next round.

Case 4: consumer unregistered. When a consumer finishes reading the data from the capsule, it unregisters itself. However, there might still be some remaining data within the capsule. In such cases, the consumer places the capsule into a *fragment list*. When other consumers move to a new capsule, they first attempt to retrieve a capsule from the fragment list before invoking the *reserve* function.

The cases provided above demonstrate four typical ways in which producers and consumers operate and exit a capsule. However, it is important to acknowledge that combinations of these cases can occur, adding complexity to the handling process. For instance, there may be a situation where a consumer closes the capsule while its producer unregisters itself. Such scenarios require careful consideration and appropriate handling to ensure proper synchronization between the producer and consumer.

3.5 Ordering and Progress Guarantees

k-FIFO semantics. CNQ provides k-FIFO semantics by constraining the number of capsules that producers and consumers can interact with. Specifically, suppose each capsule stores NE entries; CNQ ensures that producers can write to at most D consecutive capsules simultaneously, and consumers can read from at most D consecutive capsules simultaneously. Here, the parameter D is defined as $\mathsf{D} = \frac{k}{2\times\mathsf{NE}}$. Note that the consecutive capsules for producers and consumers are not necessarily identical.

Theorem 1. *CNQ ensures k-FIFO.*

Proof. To prove that CNQ is k-FIFO, we only need to prove that the oldest element in the queue must be dequeued within k steps. Let $\mathsf{E_u}$ denote the *set* of all elements. Additionally, let $\mathsf{S_{enq}}$, $\mathsf{S_{deq}}$, and $\mathsf{S_{pos}}$ be *sequence*s of $\mathsf{E_u}$, representing the *history* [124] in which elements are enqueued, dequeued, and the positional order of the elements in the queue, respectively. Then, we only need to prove $\mathrm{G0}$: $\forall$ t $\in$ *suffix*($\mathsf{S_{deq}}$), $\forall$ s $\in$ *subseq*($\mathsf{S_{enq}}$), *set*(t) = *set*(s) $\Rightarrow$ s[0] $\in$ t[0, k), where *suffix*, *subseq*, and *set* refer to the suffix set, the subsequence set, and the element set of a sequence.

Since the producer can write to at most D consecutive capsules, each capsule's size is NE, and $\mathsf{D} = \frac{k}{2\times\mathsf{NE}}$, we know that $\forall\ e_a, e_b$, $\mathsf{S_{enq}}$.*index*(e_a) < $\mathsf{S_{enq}}$.*index*(e_b) $\Rightarrow$ $\mathsf{S_{pos}}$.*index*(e_a) - $\mathsf{S_{pos}}$.*index*(e_b) < D $\times$ NE = $k/2$, where *index* means the subscript of the element in a sequence. Then, we have $\mathrm{L0}$: $\forall$ t $\in$ *suffix*($\mathsf{S_{enq}}$), $\forall$ s $\in$ *subseq*($\mathsf{S_{pos}}$), *set*(t) = *set*(s) $\Rightarrow$ t[0] $\in$ s[0, $k/2$).

Then, we have $\mathrm{L1}$: $\forall$ t $\in$ *subseq*($\mathsf{S_{enq}}$), $\forall$ s $\in$ *subseq*($\mathsf{S_{pos}}$), *set*(t) = *set*(s) $\Rightarrow$ t[0] $\in$ s[0, $k/2$). This can be proved by first constructing sequence $\mathsf{t_s}$ such that $\mathsf{t_s} \in$ *suffix*($\mathsf{S_{enq}}$) $\wedge$ $\mathsf{t_s}$[0] = t[0] $\wedge$ *set*(t) $\subseteq$ *set*($\mathsf{t_s}$), and set $\mathsf{s_s} = \mathsf{t_s} \setminus$ t, and then prove ($\mathsf{t_s} \setminus \mathsf{s_s}$)[0] $\in$ s[0, $k/2$) holds by induction on $\mathsf{s_s}$, where the base case $\mathsf{s_s} = \varnothing$ can be proved by applying $\mathrm{L0}$.

Since the consumer can read from at most D consecutive capsules, we know that $\forall e_a, e_b$, $\mathsf{S_{pos}}$.*index*(e_b) - $\mathsf{S_{pos}}$.*index*(e_a) $\geq k/2 \Rightarrow$ $\mathsf{S_{deq}}$.*index*(e_a) <

S_{deq}.*index*(e_b). Then we have L2: $\forall$ t $\in$ *subseq*(S_{pos}), $\forall$ s $\in$ *subseq*(S_{deq}), *set*(t) = *set*(s) $\Rightarrow \forall e_a, e_b$, t.*index*($e_b$) - t.*index*($e_a$) $\geq k/2 \Rightarrow$ s.*index*(e_a) $<$ s.*index*(e_b).

To prove G0, suppose these two subsequences are t_0 and s_0. Let $e_0 := s_0[0]$. We only need to prove G1: $t_0 \in$ *suffix*(S_{deq}) $\wedge$ $s_0 \in$ *subseq*(S_{enq}) $\wedge$ *set*(t_0) = *set*(s_0) $\Rightarrow e_0 \in t_0[0, k)$. Let r_0 satisfies $r_0 \in$ *subseq*(S_{pos}) $\wedge$ *set*(r_0) = *set*(s_0). By applying L1 (t $\leftarrow s_0$, s $\leftarrow r_0$), we know L3: r_0.*index*(e_0) $< k/2$.

To prove G1, by L3, we only need to prove G2: t_0.*index*(e_0) $\leq$ K where K = $k/2$ + r_0.*index*(e_0). We prove it by contradiction. Suppose H0: t_0.*index*(e_0) $>$ K. Since we know $|r_0[0{:}K)| = |t_0[0{:}K)| =$ K, $e_0 \in r_0[0{:}K)$, $e_0 \notin t_0[0{:}K)$, then $\exists$ e_1, $e_1 \in t_0[0{:}K) \wedge e_1 \notin r_0[0{:}K)$, thus L4: t_0.*index*(e_1) $<$ K $\wedge$ r_0.*index*(e_1) $\geq$ K. By L3, we know r_0.*index*(e_1) - r_0.*index*(e_0) $\geq k/2$. By L2, we know t_0.*index*(e_0) $<$ t_0.*index*(e_1). By L4, we know t_0.*index*(e_0) $<$ K, which contradicts the assumption H0. Therefore, G2 holds. □

Emptiness Check. Our nested approach does not ensure a linearizable emptiness check [131]. This means that the dequeue operation might return empty even if there are items in the queue (for instance, when all data-containing capsules are reserved by other consumers). However, we guarantee that if CNQ holds more than k items, the dequeue operation will consistently return a non-empty result.

Progress Guarantee. In terms of progress guarantees, we ensure deadlock-free. Producers and consumers won't be blocked since conflicts (e.g., caused by CAS) always result in success for a producer or consumer, ensuring progress. With a finite number of threads and enqueue/dequeue operations, the system will eventually terminate. This termination property has also been verified by the model checker (see §3.8).

```
1  Struct CNQ<T, K> {
2    Tube <Capsule<T>, K> t;
3    List <CQRef> f;
4   where CQRef := Ref<Capsule<T>>
5  }
6
7  Bool CNQ::enqueue(T e) {
8    CQRef c = get_handler();
9    if (!c) goto advance;
10 again:
11   switch (c.put(e))
12   case success: return true;
13   case finished:
14     half_consume(c);
15     c = null;
16     return true;
17   case closed:
18     half_consume(c);
19     goto advance;
20 advance:
21    c = t.allocate_k();
22    if (!c) return false;
23    c.reset();
24    t.produce_k(c);
25    goto again;
26 }
27
28 T CNQ::dequeue() {
29   CQRef c = get_handler();
30   if (!c) goto advance;
31 again:
32   switch (c.get())
33   case success(e): return e;
34   case finished(e):
35     half_consume(c);
36     c = null;
37     return e;
38   case closed:
39     half_consume(c);
40     goto advance;
41   case empty: return null;
42   case conflict: goto again;
43 advance:
44   c = f.remove();
45   if (c) goto again;
46   c = t.reserve_k();
47   if (!c) return null;
48   goto again;
49 }
50
51 Void CNQ::producer_unregister() {
52   CQRef c = get_handler();
53   if (!c) return;
54   c.close(); half_consume(c);
55 }
56
57 Void CNQ::consumer_unregister() {
58   CQRef c = get_handler();
59   if (c) f.insert(c);
60 }
61
62 Void CNQ<T>:half_consume(CQRef c)
       {
63   do {
64     b = LOAD(c.half);
65     if (b) {
66       t.consume_k(c); break;
67     }
68   } while (!CAS(c.half, false,
       true));
69 }
70
```

Figure 3.4: Structure and operations of CNQ.

3.6 Implementation

3.6.1 CNQ

Interfaces. Beside enqueue and dequeue operations, each producer and consumer must register and unregister themselves before and after enqueueing and dequeueing data, respectively. Registration and unregistration are one-time operations and should only be performed during the initialization or destruction

```
71  let D := K / (2 * NE);
72
73  Struct Tube<E, K> {
74     Queue <E> q;
75  }
76  Ref<E> Tube::allocate_k() {
77     i = q.producer_newest() + 1;
78     c = q.at(i - D);
79     c.close();
80     return q.allocate_at(i);
81  }
82  Void Tube::produce_k(E c) {
83     q.produce(c);
84  }
```

```
85  Ref<E> Tube::reserve_k() {
86     i = q.consumer_newest() + 1;
87     c = q.at(i - D);
88     if (!LOAD(c.finished))
89       return null;
90     c' = q.reserve_at(i);
91     if (c')
92       STORE(c.finished, false);
93     return c';
94  }
95  Void Tube::consume_k(E c) {
96     STORE(c.finished, true);
97     q.consume(c);
98  }
```

Figure 3.5: Structure and operations of the tube.

```
99  Struct Capsule<T> {
100    AtomicInt prod
101              <closed:1, pidx:_>;
102    AtomicInt cidx;
103    AtomicBool half, finished;
104    T entries[NE];
105 }
106 Status Capsule::put(T data) {
107   i = LOAD(prod).pidx;
108   entries[i] = data;
109   b = FAA(prod, 1).closed;
110   if (b) return closed;
111   return (i + 1 == NE) ?
112           finished : success;
113 }
114 Status Capsule::get() {
115   i = LOAD(cidx);
116   p = LOAD(prod);
117   if (i < p.pidx) {
118     data = entries[i];
119     STORE(cidx, i + 1);
120     return (i + 1 == NE) ?
121             finished(data):
122
```

```
123             success(data);
124   }
125   if (p.closed) return closed;
126   if (!can_advance())
127     return empty;
128    p' = {.closed=true,.pidx=p.
          pidx};
129    return (CAS(prod, p, p')) ?
130            closed : conflict;
131 }
132
133 Void Capsule::reset() {
134   STORE(prod, <false, 0>);
135   STORE(cidx, 0);
136   STORE(half, false);
137 }
138
139 Void Capsule::close() {
140   do {
141     p = LOAD(prod);
142     p' = {.closed=true, .pidx=p.
          pidx};
143   } while (!CAS(prod, p, p'));
144 }
```

Figure 3.6: Structure and operations of the capsule.

of the producer and consumer, without the need for each individual enqueue or dequeue operation.

Structure. CNQ is structured around a tube t, capable of accommodating multiple capsules (line 2). In a CNQ configuration with a capacity of C and

each capsule having a capacity of NE, the tube can store C/NE capsules. Additionally, CNQ includes a list of capsule references f (line 3), used to store the fragments generated by the unregistered consumer. This list supports linearizable *insert* and *remove* operations and ensures that the order of its capsules corresponds to their positional order in the tube.

Enqueue and dequeue operations. Lines 7-49 in Fig. 3.4 present a high-level overview of enqueue and dequeue operations in CNQ. Each producer and consumer is associated with a handler that points to a specific capsule. The *get_handler* function is utilized to retrieve the respective handler. For enqueue, a producer first retrieves its capsule from the handler (line 8). If the handler is null (line 9), indicating the absence of a capsule, the producer attempts to allocate a new capsule by jumping to the label *advance*. Otherwise, the producer proceeds with the enqueue operation inside the capsule (line 11), which can have various outcomes. (1) If enqueue is successful (*success*), it directly returns true. (2) If enqueue is successful but the capsule becomes fully utilized (*finished*), the producer exits the capsule by invoking the half_consume function, sets the handler to null, and then returns. (3) If the capsule is closed by the consumer (*closed*), the producer exits the capsule and proceeds to allocate a new one.

When the producer attempts to advance to the next capsule (line 20), it tries to allocate a new one. If the allocation is successful, it assigns the newly allocated capsule to its handler (line 21). Subsequently, the producer resets the capsule by resetting all its variables (line 133) and then invokes the produce_k function to make the capsule visible to consumers (lines 23 and 24). Finally, the producer attempts to enqueue again. If the allocation fails, indicating that the queue is already full, the producer returns false (line 22).

The consumer follows a similar procedure for when attempting to dequeue from a capsule. In addition to the *success*, *finished*, and *closed* cases, there may be no data available in the entire queue (*empty*). In this case, the consumer returns null to indicate that the queue is empty. Furthermore, the consumer may encounter conflicts with the producer (*conflict*), such as when the producer

resumes producing while the consumer is trying to close the capsule. In such cases, the consumer retries the dequeue operation. When consumers attempt to advance to a new capsule (line 43), they first check if there are any available capsules in the fragment list. If there are, they retrieve one from there. However, if no capsules are available in the fragment list, the consumers try to reserve a new capsule.

Function half_consume. Since the *consume* function is only called when both the producer and the consumer have exited the capsule, we introduce the `half_consume` function (line 62), which is invoked whenever either the producer or the consumer exits the capsule. The `consume_k` function is called when the `half_consume` function has been called twice on the same capsule, indicating that both the producer and consumer have exited the capsule.

Registeration and unregisteration. The registration process for producers and consumers involves simply initializing their handlers to null. Capsules are assigned to producers and consumers once they call enqueue or dequeue operations. Conversely, during unregistration, the producer closes its capsule and invokes the half_consume function before exiting. Consumers, on the other hand, add their capsule to the fragment list (line 59).

3.6.2 The Tube

k-FIFO support. CNQ ensures the k-FIFO property by enforcing a constraint on the distance between the newest and oldest capsules belonging to either the producer or the consumer. To achieve this, as illustrated in Fig. 3.5, the tube of CNQ extends the functions of the three-phase queue, identified by the _k suffix (lines 76-98). These extended functions ensure that producers and consumers write to and read from at most D consecutive capsules simultaneously. When a producer calls `allocate_k` to allocate the capsule with index i, it closes the capsule c with index i-D, thereby upholding the stated ordering guarantee. On the other hand, `reserve_k` reserves the capsule c′ with index i and checks if the capsule c with index i-D has finished consuming or not, as indicated by

the finished flag. If true, it returns c′. The finished flag is set when calling `consume_k`.

3.6.3 The Capsule

Structure. As shown in Fig. 3.6, a capsule (line 99) represents an SPSC queue consisting of NE entries. Each capsule is equipped with two atomic variables: pidx and cidx, storing the positions of the producer and consumer within the capsule, respectively. Furthermore, pidx has an associated boolean variable closed indicating whether the capsule is closed. Additionally, we introduce a boolean variable half to keep track of whether either the producer or the consumer has left the capsule, which is utilized in the `half_consume` function. Also, there is a finished flag indicating whether all its data has been consumed.

Put and get operations. Inside the capsule, the producer begins by reading its index (line 107) and making a copy of the data (line 108). It then updates the index and checks whether the capsule is closed. If the capsule is closed, the put operation in the capsule is aborted, and the state *closed* is returned. If the capsule is still open, the producer checks if it has written all the entries of the current capsule (line 111). If it has, it returns *finished*. Otherwise, it returns *success*.

For the consumer (line 114), it begins by reading both the producer and consumer indexes, and compares them to determine if there is any unread data. If unread data is present, it proceeds to read the entry, updates the consumer index, and returns the data. However, if the entire capsule has been utilized, it returns *finished* to indicate that all data within the capsule has been consumed.

If there is no unread data, the consumer first checks if the capsule has been closed by the producer, typically due to the producer becoming unregistered. If the capsule is closed, the consumer returns the *closed* state. Next, the consumer evaluates whether it can advance to a new capsule (line 127). If it is unable to advance to a new capsule, it returns the *empty* state. However, if the consumer can advance to a new capsule, it attempts to close the current one (line 129).

If the closing procedure is successful, the consumer returns the *closed* state. In cases where there is a conflict during the closing procedure, the consumer returns the *conflict* state.

Function can_advance. In the scenario of consumers closing a capsule (line 127), there is a concern that if they directly close the capsule, it may consecutively close capsules, causing the producer to be unable to produce any data and resulting in a live-lock. This pattern is also observed in certain FIFO queues [157, 161]. To address this issue, when closing a capsule, a scan is performed from the current capsule to the most recently allocated capsule. During this scan, if any of these capsules contain data, even if they are already owned by other consumers, the current capsule can be safely closed. This approach ensures that each time a capsule is closed, subsequent capsules will have at least one item, thereby preventing the live-lock.

3.7 Integration

CNQ comprises two components: the tube and the capsule. In Sec.3.6, we present the algorithm for the tube based on the interface of three-phase queues. This section introduces various existing three-phase implementations that adhere to the interface, specifically three MPMC FIFO queues: the ring buffer in the Data Plane Development Kit (DPDKRB) [18], the Scalable Circular Queue (SCQD) [161], and the Block-based Bounded Queue (BBQ) [184]. Given that CNQ's performance relies on both the tube and capsule components, we also introduce FAAQ, a novel three-phase queue that demonstrates performance improvements compared to existing solutions, thereby enhancing CNQ's overall performance.

Integration effort. Our primary objective is to empower these queues to offer the three-phase queue interfaces, namely *allocate_at*, *produce*, *reserve_at*, and *consume*, as depicted in Fig. 3.5. To achieve this, we conducted interface remapping, involving the modification of approximately a dozen lines of code for each of these queues.

3.7.1 DPDKRB

The ring buffer in DPDK utilizes an algorithm similar to the one depicted in Fig. 3.2. To incorporate it as the underlying structure for CNQ, we have identified the functions `__rte_ring_move_prod_head`, `__rte_ring_move_cons_head`, and `__rte_ring_update_tail` (with different parameters) in the source code [88] as the four functions corresponding to the aforementioned three-phase queue operations.

3.7.2 SCQD

SCQD is an MPMC FIFO queue renowned for its exceptional performance. It is composed of two index queues, fq and aq, along with a data array. Initially, all data array indexes are stored in the fq index queue. During the enqueue operation, an index is acquired from fq, and the corresponding data is copied to the appropriate location within the data array. Subsequently, the index is placed into the aq index queue. On the other hand, during the dequeue operation, an index is retrieved from the aq index queue. The data at the corresponding position in the data array is then copied out, and finally, the index is returned to the fq index queue. The enqueue and dequeue operations in SCQD are distinctly divided into these three well-defined phases. Consequently, the retrieval and placement of indexes into the fq and aq queues fulfill the four functions of the three-phase queue.

3.7.3 BBQ

To address performance degradation resulting from concurrent read and write operations on shared metadata in traditional queues, BBQ introduces a partitioning mechanism. The queue is divided into multiple sections known as blocks, with each block having its own metadata. During enqueue and dequeue operations, the producer or consumer retrieves the current block it belongs to, and then attempts to allocate or reserve an entry (allocate_entry or reserve_en-

try in BBQ) within that block. If the allocation or reservation is successful, the producer or consumer proceeds to perform the produce or consume operation (commit_entry or consume_entry in BBQ) after copying the data. In the event that the current block is fully utilized, it advances to the next block and retries the allocation or reservation on the new block. These actions of obtaining a block, attempting allocation or reservation, advancing to a new block, and retrying upon failure are encapsulated as an *allocate* or *reserve* function of the three-phase queue. On the other hand, the functions commit_entry and consume_entry in BBQ directly correspond to the *produce* and *consume* functions, respectively, in the context of the three-phase queue, requiring no additional modifications.

3.7.4 FAAQ

BBQ has been optimized to reduce interference between producers and consumers, resulting in excellent performance in single-consumer scenarios. However, it lacks optimization for multi-consumer workloads. In contrast, state-of-the-art approaches have successfully utilized the more efficient FAA technique. BBQ, on the other hand, still relies on MAX or CAS operations that necessitate retries for dequeue operations, leading to performance bottlenecks.

The direct use of FAA, however, is not straightforward due to its lack of atomicity with conditional updating semantics, as observed in state-of-the-art FIFO queues [161, 157, 28]. For instance, if multiple consumers attempt to find available entries for consumption and both use FAA to update the position, it is possible that the position of consumers exceeds that of producers. This situation necessitates a rollback, negatively impacting overall performance [161, 157].

Nevertheless, the block-based feature in BBQ enables us to reduce the probability of rollback when utilizing FAA for dequeue operations, thereby improving performance. Specifically, when a consumer retrieves its block and attempts to read the data, it first checks if the producer has finished writing on it. If the writing is incomplete, the consumer retrieves the data using CAS. However, if

the writing is complete, the consumer can attempt to use FAA because, at this point, the producer's position is at the boundary of the block. Consequently, using FAA to move the consumer's position beyond this boundary does not require rollback. Therefore, when the producer and consumer are not on the same block, the consumer can directly use FAA to enhance performance. It is important to highlight an extreme situation in which the consumer determines that the current block is fully produced. However, before updating its position using FAA, the block has already been completely read by other consumers. Subsequently, the producer writes new data into that block for the next round. It is worth noting that rollback is only necessary in this specific extreme case, and it is not likely to occur for every empty case, as observed in existing works [161, 157].

3.8 Verification

To ensure the correctness of CNQ, we utilize the VSync [165] tool for formal verification of the algorithm. VSync leverages GenMC [135] as its backend, a powerful model checking tool capable of verifying various properties such as memory safety, data race freedom, and loop termination [91]. The verification process involves providing the C code of the algorithm and the client code as input to the tool.

The client code consists of multiple threads, each invoking the algorithm's API functions a specific number of times. GenMC then systematically explores all possible executions of the algorithm triggered by the client code and checks the desired properties. Additionally, users can include assertions in the client code to verify properties that are specific to the algorithm being verified.

For CNQ, the complexity of the algorithm requires careful design of the client code to ensure the triggering of all edge cases. For instance, in the case of capsules, it is necessary to trigger all possible states of the capsule as depicted in §3.4.2. However, due to the significant number of threads and data operations involved, it is often impractical for the model checking tool to

ID	*C1*	*C2*	*C3*	*C4*	*C5*	*C6*
capacity	2	2	2	2	2	2
k	2	2	2	2	2	2
#capsules	1	1	1	2	2	2
#producers	2	1	2	2	1	2
#consumers	1	2	2	1	2	2
#data produced	2,1	3	2,1	2,1	3	2,1
#data consumed	3	2,1	2,1	3	2,1	2,1
#executions	770	165	122k	207k	74k	179m
verification time	6s	3s	628s	1089s	372s	11days

Table 3.1: Statistics of verification client code.

complete the verification within a reasonable time for a single client code. To address this challenge, we adopted a partitioning approach for the client code, as illustrated in Table 3.1, with each partition being responsible for triggering different edge cases.

In all client code, the capacity of the queue and the value of k are set to 2. The client code is divided into two groups: $C1$-$C3$ and $C4$-$C6$. In the first group, there is only one capsule in the queue with a size of 1. The primary objective of this group is to verify the correctness of the capsule algorithm. In contrast, the second group contains the queue with two capsules, each having a size of 1. The client code in this group primarily focuses on verifying the integration.

Each group of clients comprises three different workloads: MPSC, SPMC, and MPMC. In cases where there are multiple producers or consumers, two threads are utilized accordingly. In each client code, a maximum of 3 items are processed. If there are 2 producers (or consumers), each producer (or consumer) enqueues (or dequeues) up to 2 items and 1 item, respectively. In the case of a single producer (or consumer), it can enqueue (or dequeue) a maximum of 3 items. All items inserted into the queue are unique. After enqueuing or dequeuing all items, the corresponding unregister function is called. In certain executions, it is possible for the queue to not be empty after all threads have finished, for example, when the consumer executes before the producer. To handle this situation, once all threads have completed execution, we dequeue

from the queue until it is empty. We then assert that all enqueued data items are dequeued exactly once.

GenMC performs a comprehensive verification by checking all executions triggered by the client code and ensuring that the assertion holds, as well as verifying the aforementioned general properties. The coverage statistics reported by GenMC indicate that all atomic operations have been covered during the verification process. Table 3.1 presents the checking time and the number of executions checked for each client code, ranging from a few hundred states checked in a few seconds to millions of states requiring several days for completion.

CNQ contains memory barriers, and we have optimized the choices of them (i.e., sequentially consistent, acquire, release, or plain memory accesses) and verified the correctness of CNQ under IMM [169] and RC11 [140] weak memory models using the model checker.

3.9 Evaluation

Environment Setup. Experiments were conducted on a server equipped with two Intel Xeon Gold 6266C CPUs [130]. Each CPU has 22 cores, resulting in a total of 44 cores (88 hyperthreads). The server was running Ubuntu 22.04 LTS with Linux kernel 5.15 and GCC 11.2.

Configuration. The data size is fixed at 8 bytes, which is supported by all queues. The initial memory usage for each queue is set to 1MB. In the case of CNQ, the number of capsules is configured to be equal to the number of threads. We evaluate them using three distinct workloads: MPSC, SPMC, and MPMC. In the MPMC workload, the number of producers and consumers is the same. Each experiment is conducted three times to ensure reliability. During the benchmarking process, the threads are assigned to CPU cores starting from those with smaller IDs. In the figures, vertical dashed lines indicate instances where threads cross NUMA nodes or are assigned to hyperthreads. The solid lines represent the average results, while the shaded area in the figures

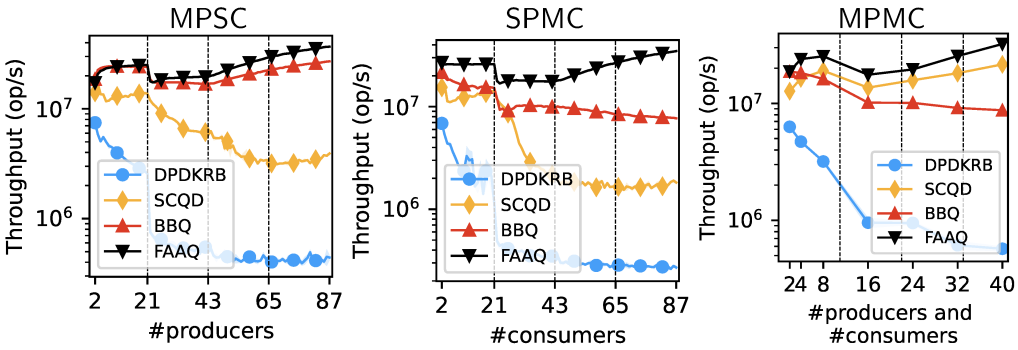


Figure 3.7: FIFO queue throughput.

corresponds to the standard deviation.

3.9.1 CNQ Speedup with Different ks

Throughput of FIFO queues. To showcase FAAQ's enhanced performance compared to existing queues, particularly under the multiple consumer case, as illustrated in Figure 3.7, we present a comparative analysis of the throughput (i.e., the total number of operations per second) of various FIFO queues under three distinct workloads: MPSC, SPMC, and MPMC. Irrespective of the workload type, DPDKRB consistently exhibits the poorest performance. For MPSC and SPMC workloads, BBQ demonstrates superior performance compared to SCQD, while for the MPMC workload, SCQD outperforms BBQ. This discrepancy can be attributed to BBQ's primary emphasis on reducing competition between producers and consumers, which lacks optimization for MPMC scenarios. However, in our optimized version of BBQ, namely FAAQ, we consistently achieve the highest performance across all three workloads. FAAQ outperforms the second-best performing queue by a substantial margin, with performance gains of up to 1.4x, 4.5x, and 1.6x for each respective workload.

Performance improvement by using CNQ. Next, we assess the speedup of CNQ for different values of k when integrated with various MPMC FIFO queues, denoted as CNQ-DPDKRB, CNQ-SCQD, CNQ-BBQ, and CNQ-FAAQ, respectively. We conducted experiments with varying values of k ranging from 2^1 to 2^{14} while ensuring that k is greater than or equal to the number of

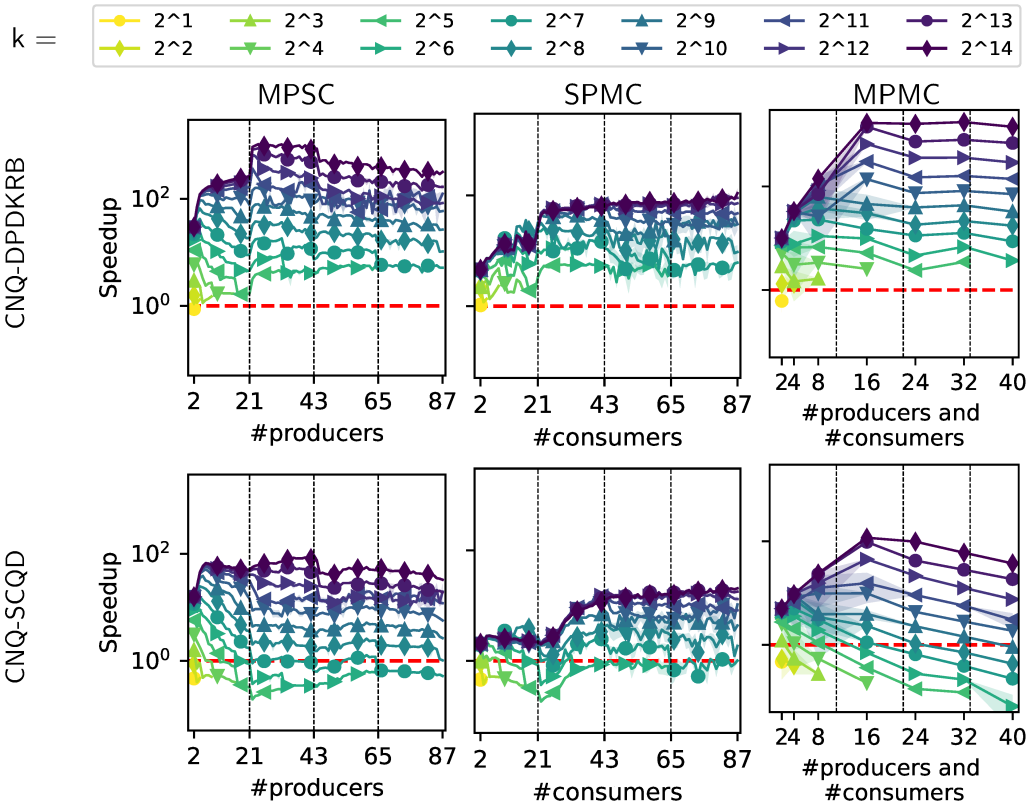


Figure 3.8: Speedup with the integration of CNQ under different ks (1).

producers or consumers. The results are presented in Fig.3.7.

From the perspective of k, larger values lead to more significant performance improvements. For instance, in the case of the MPMC workload, CNQ-FAAQ exhibits performance improvements of up to 3.9x and 21.1x when k is set to 2^8 and 2^{12}, respectively. Conversely, smaller values of k may even result in negative performance speedup. This is due to the capsule size being excessively small, leading to frequent invocations of functions that advancing to new capsules.

Regarding workloads, the MPMC workload demonstrates the most substantial performance improvement, followed by MPSC, with SPMC showing the least enhancement. When considering each queue individually, CNQ-DPDK achieves the highest performance boost among the queues, with improvements of 1055x, 110x, and 1752x times across the three workloads. On the other hand, the performance improvements for CNQ-SCQD are less noticeable with a small number of k. Additionally, both CNQ-BBQ and CNQ-FAAQ exhibit

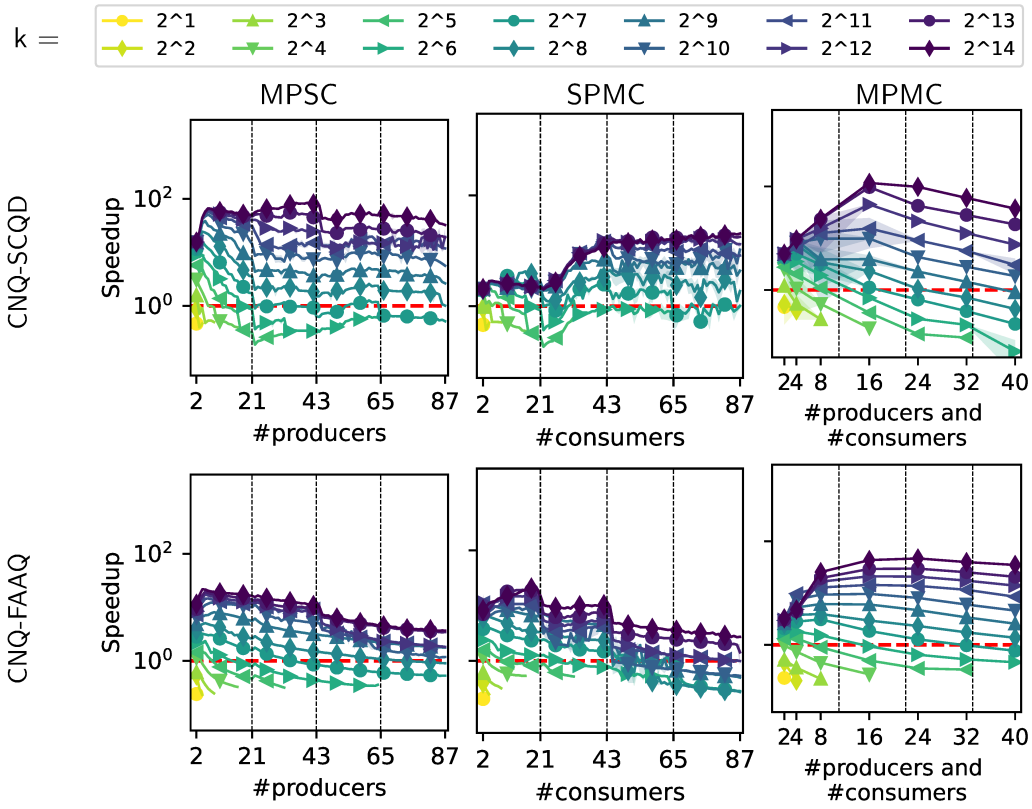


Figure 3.9: Speedup with the integration of CNQ under different ks (2).

similar enhancements in performance.

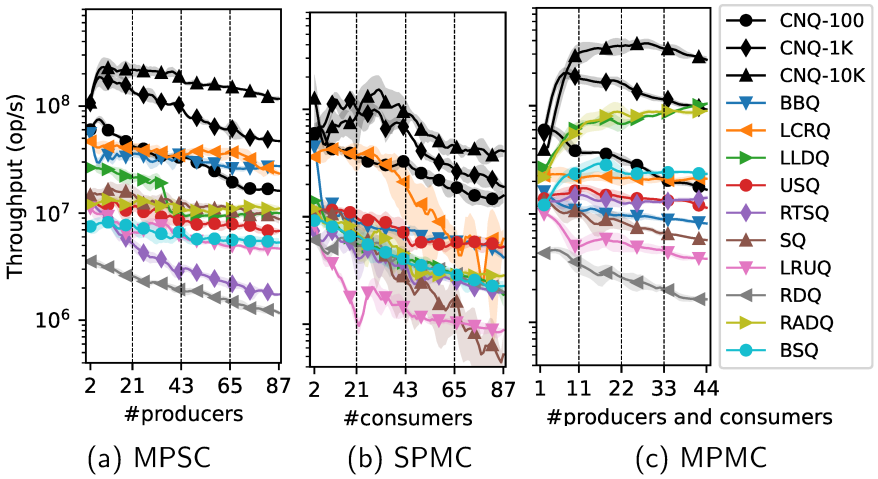
Based on the throughput of the FIFO queues and the performance improvements achieved by CNQ on each queue, we selected FAAQ as the tube for CNQ to conduct comparative experiments against state-of-the-art approaches in the following section (abbreviated as CNQ when there is no ambiguity).

3.9.2 State-of-the-art Comparison

Scal benchmark. Scal [115] is a performance benchmarking framework widely used for evaluating the performance of concurrent data structures. In order to support the upcoming experiments, several extensions were made to the framework: (1) In addition to the existing queues in Scal, we have incorporated BBQ and CNQ-FAAQ into the framework. Regarding CNQ-FAAQ, we conducted experiments using three different values of k: 10^2, 10^3, and 10^4. (2)

Name	Feature	Ordering Semantics
CNQ	nested	k-FIFO
BBQ [184]	block-based	FIFO
LCRQ [157]	list of array queue	FIFO
BSQ [131]	random queue	k-FIFO
USQ [131]	random queue	k-FIFO
RTSQ [113]	random queue	quiescently consistent
RDQ [90]	random queue	quasi-linearizable
SQ [90]	random queue	quasi-linearizable
LLDQ [114]	distributed queue	locally linearizable
LRUQ [116]	distributed queue	k-FIFO
RADQ [116]	distributed queue	random balancing

Table 3.2: FIFO and relaxed queues in the comparison.



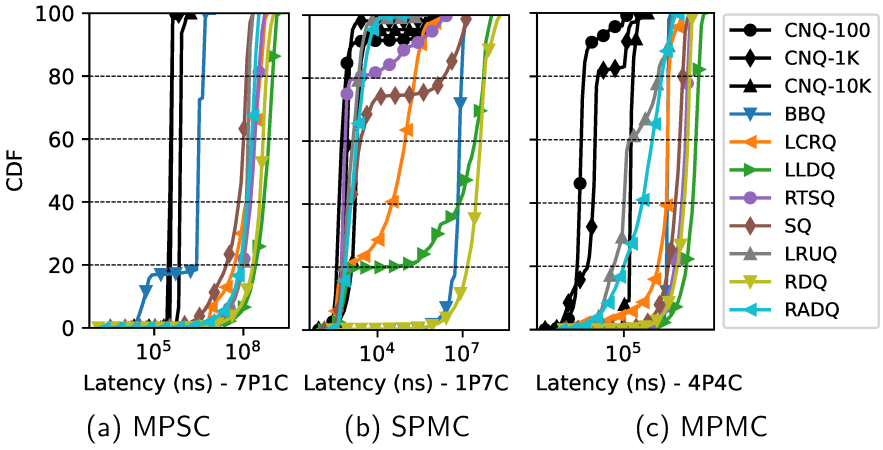
(a) MPSC (b) SPMC (c) MPMC

Figure 3.10: Throughput comparison results.

We expanded the memory allocation module of Scal to gather supplementary statistical data, including peak memory usage. (3) The initial framework solely supported throughput measurements. However, we enhanced it to incorporate data latency measurements, specifically the average waiting time of each data item in the queue.

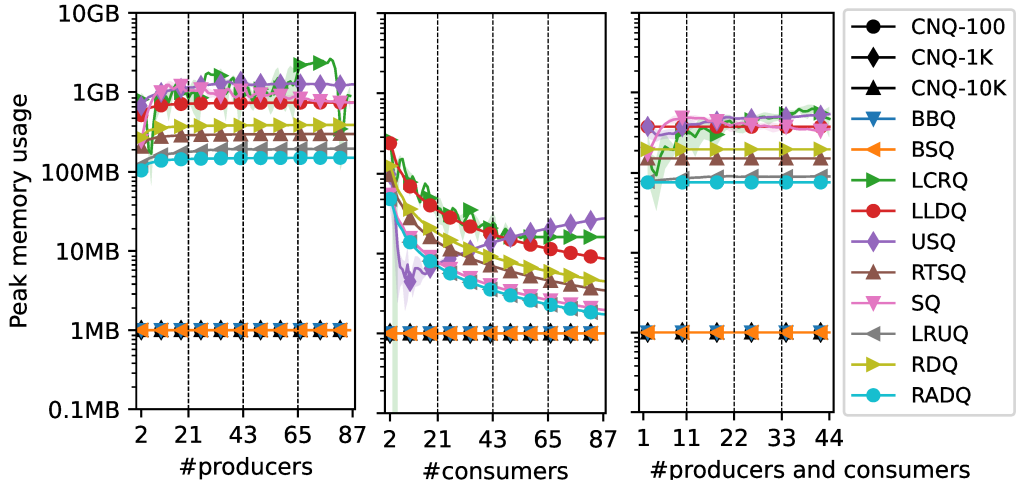
The parameters for other queues, such as k, were set to their default values. The feature and ordering semantics of these queues are outlined in Table 3.2.

Total throughput. Figure 3.10 depicts the throughput results of different queues under different workloads. CNQ demonstrates superior performance compared to other queues across all three workloads, especially with larger



(a) MPSC (b) SPMC (c) MPMC

Figure 3.11: Data latency comparison results (CDF, #thread=8).



(a) MPSC (b) SPMC (c) MPMC

Figure 3.12: Peak memory usage comparison results.

values of k, such as 10^3 or 10^4. Notably, with k=10^4, CNQ achieves a substantial performance enhancement of 6.6x, 14.5x, and 6.0x times compared to the second-best performing queue.

Furthermore, apart from CNQ, state-of-the-art FIFO queues such as BBQ and LCRQ consistently outperform other relaxed queues in the MPSC and SPMC workloads. This can be attributed to their utilization of more scalable hardware instruction, FAA, which contributes to enhanced performance.

Among the other relaxed queues, distributed queues like LLDR and RADQ, which employ weaker ordering semantics, demonstrate optimal performance,

especially in the MPMC workload. This can be attributed to the fact that in balanced workloads, the synchronization overhead associated with the load balancer is minimized.

Data latency. In addition to throughput, we also investigated the data latency of different queues, considering the varying ordering semantics that could delay the dequeue of old items. We excluded BSQ and USQ from the data latency comparison benchmark due to a potential bug that affected the result. Fig. 3.11 presents the cumulative distribution function (CDF) of data latency under different workloads for 8 threads. It is evident that regardless of the workload, CNQ consistently exhibits the best average data latency, surpassing the second-best option by up to 7.9x, 1.3x, and 79.4x times for the three respective workloads. Furthermore, a smaller value of k in CNQ corresponds to a lower average latency, aligning with our expectations. This can be attributed to the k-fifo property, which ensures that the data will be dequeued within k dequeue operations. Consequently, a smaller value of k implies the possibility of earlier dequeuing, resulting in reduced latency. On the other hand, while LLDQ demonstrates excellent throughput performance, it performs poorly in terms of data latency.

The benchmark results reveal a trade-off between the throughput and data latency: a larger k leads to higher throughput but may increase data latency. This is because, in comparison to FIFO queues, the latest item in the queue is not dequeued first but within k dequeue operations. In real-world usage, users have the flexibility to adjust the value of k to align with their specific requirements. For instance, in network processing applications with a fixed latency threshold, such as P99 metrics, users can increase k until the queue meets the minimal requirement desired latency threshold, thereby maximizing throughput. The same principle holds true for fixed throughput scenarios. This adaptability underscores the practicality and versatility of our approach, allowing users to fine-tune the performance based on their specific use cases and performance priorities.

Peak memory usage. Finally, let us further investigate the memory usage of each queue in the aforementioned throughput and latency experiments. For bounded memory usage queues like BBQ, BSQ, and CNQ, no additional memory is allocated, resulting in a peak memory usage value equal to the initial value, i.e., 1MB, indicating minimal memory overhead. However, for other queues, due to considerations of load balancing and ordering semantics, data that does not meet the requirements is temporarily stored in additional allocated memory space. From the results presented in Fig. 3.12, it can be observed that the remaining queues generate a significant amount of additional memory overhead. For instance, in the MPSC and MPMC workloads, each queue requires approximately 100MB to 1GB of additional space. In the case of SPMC, the presence of multiple consumers ensures faster consumption of data compared to the other two workloads, thus reducing the need for storing them (except for USQ, as it preallocates memory based on the number of consumers).

3.10 Conclusion

This chapter introduces CNQ, a formally verified compositional approach that enhances the performance of existing FIFO queues. The performance of CNQ surpasses that of existing queues with relaxed semantics.

Chapter 4

BWoS: A Block-based Work Stealing Queue

4.1 Overview

Work stealing is a widely-used scheduling technique for parallel processing on multicore. Each core owns a queue of tasks and avoids idling by stealing tasks from other queues. Prior work mostly focuses on balancing workload among cores, disregarding whether stealing may adversely impact the owner's performance or hinder synchronization optimizations. Real-world industrial runtimes for parallel processing heavily rely on work-stealing queues for scalability, and such queues can become bottlenecks to their performance.

We present Block-based Work Stealing (BWoS), a novel and pragmatic design that splits per-core queues into multiple blocks. Thieves and owners rarely operate on the same blocks, greatly removing interferences and enabling aggressive optimizations on the owner's synchronization with thieves. Furthermore, BWoS enables a novel probabilistic stealing policy that guarantees thieves steal from longer queues with higher probability. In our evaluation, using BWoS improves performance by up to 1.25x in the Renaissance macrobenchmark when applied to Java G1GC, provides an average 1.26x speedup in JSON processing when applied to Go runtime, and improves maximum throughput of Hyper

HTTP server by 1.12x when applied to Rust Tokio runtime. In microbenchmarks, it provides 8-11x better performance than state-of-the-art designs. We have formally verified and optimized BWoS on weak memory models with a model-checking-based framework.

The remainder of this chapter is organized as follows. In §4.2, we introduce the basic concept of work-stealing, analyze the performance overhead of existing approaches, and elucidate the motivation behind BWoS. In §4.3, we present our block-based approach and outline the high-level design of BWoS. In §4.4, we delve into the detailed implementation of BWoS. In §4.5, we report our results in verifying BWoS on WMMs and relaxing its memory barriers. In §4.6, we conduct experimental comparisons of the performance of BWoS with the state-of-the-art in both micro-benchmarks and macro-benchmarks. In §4.7, we offer a concise examination of related work, and conclude our findings in §4.8.

4.2 Background

Task processing. Tasks vary a lot among benchmarks. Their processing time ranges from a few nanoseconds (*e.g.*, Java G1GC [30]), to microseconds (*e.g.*, RPC [103, 134, 187]), and even to seconds (*e.g.*, HPC tasks [73]). In this paper, we mainly focus on the nanosecond- and microsecond-scale tasks. Ignoring steals, tasks may be processed either:

- in FIFO (first-in-first-out) order, when minimizing processing latency is important (*e.g.*, network connections), or
- in LIFO (last-in-first-out) order, when only the overall execution time matters, as is often the case with multithreaded fork-join programs [105].

We use the term *queue* to refer to the instances of work stealing data structures without implying a specific task ordering.

Victim selection. There are multiple policies for selecting the victim queue to steal from. *Random* [99] chooses one of the remaining queues uniformly at random: it has the least complexity but achieves poor load balancing. Size-

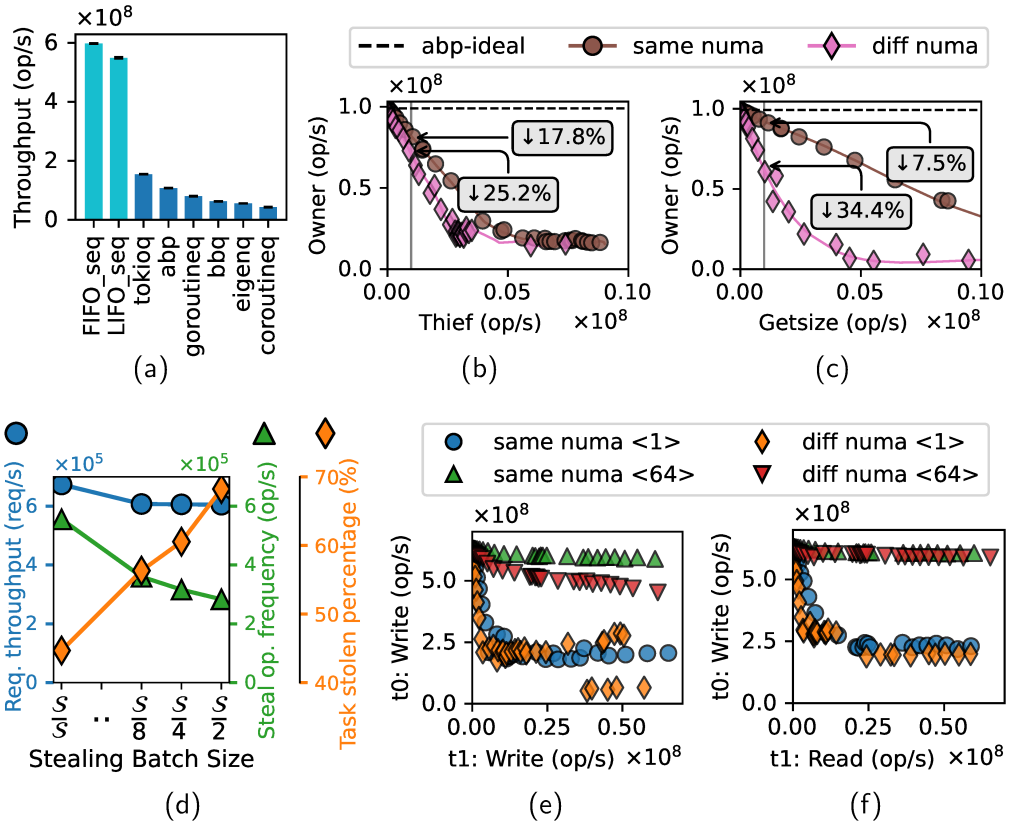


Figure 4.1: Motivating benchmarks: (a) Sequential performance of state-of-the-art work stealing algorithms. (b,c) Performance of the ABP queue owner depending on the frequency of (b) steal and (c) getsize operations. (d) Hyper HTTP server performance with different stealing batch sizes with the original Tokio work stealing queue: S is the victim queue size and $S/2$ refers to the default steal half policy [121]. (e,f) Interference between two threads for two sizes of cacheline sets.

based policies (*e.g.*, *best of two* [155] and *best of many* [127]) scan the queues' size to improve the load balance by stealing from a large queue. The NUMA-aware policy [138] was proposed to optimize the remote communication cost, by tending to steal from the queues in the local cache domain. Batch-based policies (*e.g.*, *steal half* [121] is used in Go and Rust's Tokio runtimes) allow thieves to steal multiple tasks at once to reduce their interference with the owner. Later in this section (§4.2.1.3), we will quantify these overhead sources to guide our queue design.

	Thief: cost of communication		**Victim:** cost of interference		Overhead reduction $1 - \frac{C_s+I_s}{C_d+I_d}$
	same node (C_s)	diff node (C_d)	same node (I_s)	diff node (I_d)	
abp	15ns	141ns	170ns	278ns	56%
ideal			–	–	90%

Table 4.1: Reducing the stealing overhead with a NUMA-aware policy.

4.2.1 Performance Overhead Breakdown

Next, we analyze the state-of-the-art work stealing algorithms to dissect their performance issues, and motivate the design decisions of BWoS. Fig. 4.1 contains our experimental results on an x86 server [130].

4.2.1.1 Cost of Synchronization Operations

As steals may happen at any time, strong atomic primitives are introduced for local queue manipulation. To quantify their cost, we first measure the throughput of the state-of-the-art work stealing algorithms on a sequential setup where an owner puts and gets data from its local queue, without any tasks ever being stolen (§4.6.2). We compare the results with the theoretical performance upper bound: a single-threaded FIFO (FIFO_seq) or LIFO (LIFO_seq) queue implementation [132] without support for steals. Although there is no owner-thief interference, these synchronization operations pose a huge overhead (Fig. 4.1a): throughput of these work stealing algorithms is less than 0.25x for FIFO-based (0.19x for LIFO-based) compared to the upper bound.

4.2.1.2 Interference Cost with Thieves

To estimate how thieves affect the throughput of the owner, we consider an ABP queue benchmark with an owner and one thief, which steals tasks from the queue with various frequencies (one queue and two threads in total). As the "ideal" baseline, we take the single-threaded performance of the ABP queue (*i.e.*, with no steals). To account for any NUMA effects in this measurement,

we use two configurations, running the thief in the same or in different NUMA nodes.

As we can see in Fig. 4.1b, the thief significantly degrades the owner's throughput: *e.g.*, by stealing only 1% of the tasks, the owner's throughput drops by 17.8% when the thief is in the same NUMA node, and by 25.2% when it runs in a different NUMA node. This degradation happens because of the cache interference between the owner and the thief on the shared metadata. We will further explain this in §4.2.2.

4.2.1.3 Overhead due to Victim Selection

There are two main sources of stealing overhead: first, a suboptimal victim selection can lead to workload imbalance triggering more stealing; second, the cost of steal operations.

Size-based policies. Policies like *best of two* [155] or *best of many* [127] read global metadata of multiple queues (their length) to determine the victim. Somewhat surprisingly, as shown in Fig. 4.1c, these reads introduce significant overhead for the owner, especially in the cross-NUMA scenario: even with a *getsize* frequency of only 1%, the owner throughput drops by 34.4%. This is further amplified as *getsize* is called multiple times for a single steal.

Therefore, for size-based policies, although reading more queues' sizes (*e.g.*, *best of many* [127]) can achieve better load balance, it inevitably induces more slowdown to the owners of these queues (§4.6).

NUMA-aware policies. NUMA-aware policies [138] try to reduce the overhead of each steal by prioritizing the stealing from queues in the same NUMA node. We observe that although such NUMA-aware policies can reduce the overhead of steals by 56% in the case of our ABP queue benchmark, they fail to achieve their full potential.

In Table 4.1, we break down the overhead of stealing in the ABP queue into its two main parts: the thief's communication cost and the owner's interference penalty. The former is 141ns when the thief and owner run on different

NUMA nodes (measured by Intel MCA [39]), and reduces to 15ns (consistent with the L3 cache access latency [15]) when they are at the same NUMA node. The victim's interference penalty is 170ns and 278ns for cases of thief and victim running on the same (I_s) and different (I_d) NUMA domains respectively. NUMA-aware policies with existing queues can typically eliminate the first communication overhead, while leaving the second interference overhead not sufficiently optimized.

With long enough queues, steals could ideally happen at a different part of the queue and cause no interference to the victim. This would reduce I_s and I_d to zero, resulting in a 90% improvement due to NUMA-awareness (rather than 56%).

Batch-based policies. Batch-based policies steal more tasks at once with the aim of reducing the frequency of steals. Indeed, in the Hyper HTTP server benchmark (see Fig. 4.1d), choosing larger batch sizes leads to a reduction in the number of steal operations. These larger steal operations, however, make the workload even less balanced (*i.e.* percentage of stolen tasks increases), which results in additional overhead (*e.g.*, task ping-pong), canceling out the overhead reduction due to the fewer steals: the end-to-end throughput remains roughly the same.

4.2.2 Recap to Motivate BWoS

In summary, the owner's performance suffers both from the synchronization cost, and the interference with thieves (due to victim selection and task stealing). This interference occurs because of cache contention on the queue metadata: write-write interference with *steal*, and read-write interference with *get-size* in size-based stealing policies.

To better understand the effects of these types of cache contention, we conduct a simple microbenchmark with two threads: thread t_0 continuously writes to a cacheline, while thread t_1 either reads or writes to a cacheline with a specified frequency (Figs. 4.1e and 4.1f). The cachelines for t_0 and t_1 are

independently and randomly chosen on each iteration out of the cacheline sets of two sizes: 1 or 64.

In both cases, the cache contention on a single cacheline significantly harms the throughput of t_0, regardless of the NUMA domain proximity. Introducing multiple cachelines (64 in this case) reduces the contention and significantly improves the throughput. Therefore, in the design of BWoS we separate the metadata.

4.3 Design

BWoS is based on a conceptually simple idea: the queue's storage is split into a number of blocks, and the global mutable metadata shared between thieves and owner is replaced with the per-block instances.

The structure of BWoS queue facilitates abstracting the operations into *block advancement* that works across blocks, and *fast path* that operates inside of the block chosen by the block advancement (§4.3.1). Moving most of the synchronization from the fast path to the block advancement allows BWoS to fully reap the performance benefit indicated by our previous observation (§4.2.1.1) thus approaching the theoretical upper bound. *get* and *steal* always happen on different blocks. We carefully construct the algorithm such that thieves cannot obstruct the progress of *get*, while *get* can safely *takeover* a block from thieves operating on it without waiting for them. For complexity consideration, we don't prohibit *put* and *steal* in the same block[1], as they can synchronize with the weak barriers without losing performance (§4.6.2).

As metadata is also split per block, thieves and the owner are likely to operate on different blocks and thus update different metadata. As explained in §4.2.2, this reduces the interference between thieves and the owner. For FIFO-based BWoS, block-local metadata allows stealing from the middle of the queue, without enforcing the SPMC queue restriction of always stealing the oldest task, which is not required by the workloads.

[1]Nevertheless, it is guaranteed automatically in LIFO BWoS.

```
1  bool queue<E>::put(E e){
2  again:
3    blk = blk_to_put();
4    switch(blk.put(e))
5    case success():
6      return true;
7    case blk_done(rnd):
8      if (adv_blk_put(blk,rnd))
9        goto again;
10     else return false;
11 }
12 E queue<E>::get(){
13 again:
14   blk = blk_to_get();
15   switch(blk.get())
16   case success(e):
17     return e;
18   case empty:
19     return null;
20   case blk_done(rnd):
21     if (adv_blk_get(blk,rnd))
22       goto again;
23     else return null;
24 }
25  E queue<E>::steal(){
26  again:
27    v, blk = blk_to_steal();
28    switch(blk.steal())
29    case success(e):
30      return e;
31    case empty:
32      return null;
33    case conflict:
34      goto again;
35    case blk_done(rnd):
36      if (adv_blk_steal(v,
37            blk,rnd))
38        goto again;
39      else return null;
40  }
```

Figure 4.2: Pseudocode of *put*, *get*, and *steal* operations.

BWoS can benefit from NUMA-aware policies more than other queues because the reduction in interference for the victim makes both constituents of cross-NUMA-domain stealing overhead negligible (Table 4.1). Furthermore, unlike batch-based policies, stealing policies integrated with BWoS can focus on balancing the workload itself without worrying about the interference from frequently called *steal*.

4.3.1 Bird's-Eye View of the Queue

To better understand the block-based approach, let's consider the *put*, *get*, and *steal* operations of the BWoS queue (Fig. 4.2). For each of these operations, the first step is to select a block to work on (lines 3, 14, and 27). The owner uses the *top* block for put and get for the LIFO BWoS, and gets from the *front* block and puts to the *back* block for the FIFO BWoS. In this case, top, back, and front block pointers are owner-exclusive metadata which is unavailable to the thieves. For *steal*, the choice of the block is more complicated and we will explain it in a later section (§4.3.4).

After selecting the block, operations execute the fast path (lines 4, 15, and 28), which may return one of the three results: (1) The fast path succeeds, returning the value for *get* and *steal*. (2) The fast path fails because there is no data to consume (lines 18 and 31) or because a thief detects a conflict with other thieves or with the owner due to the takeover (line 33). In case of a conflict, the fast path is retried (line 34), otherwise null value is returned. (3) The *margin* (beginning or end) of the current block is reached (lines 7, 20, and 35). In this case, the operation tries to move to the next block by performing the block advancement, and retries if it succeeds, otherwise returns the empty or full queue status.

Splitting the global metadata into block-level instances enables splitting the operations into the fast path and block advancement, which increases the performance by keeping the fast path extremely lightweight. However, the lack of global mutable metadata shared between owner and thieves raises additional challenges, which are mostly delegated to the block advancement—it is now responsible for maintaining complex block-level invariants. We introduce the following invariants:

(1) *put* never overwrites unconsumed data;

(2) *steal* and *get* never read the same data;

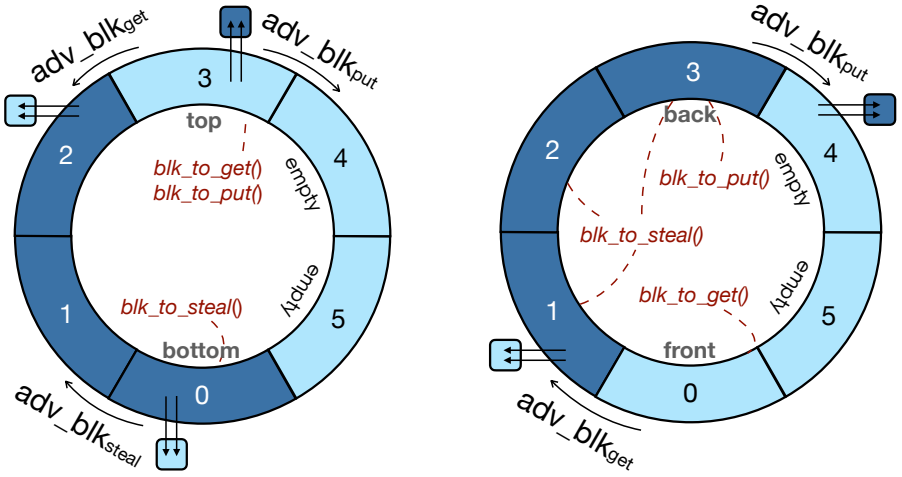
(3) *steal* and *get* never read data that has been read before;

(4) *steal* in progress cannot prevent *get* from reading from a thieves' block.

Before explaining fast path and block advancement implementations, we introduce two key concepts we rely on to ensure that the abovementioned invariants hold: *block-level synchronization* (§4.3.2) and *round control* (§4.3.3).

4.3.2 Block-level Synchronization

Block-level synchronization is the key responsibility of the block advancement and ensures that thieves never steal from the block currently used for get operations. Each block is owned either by the owner or by the thieves. For example, in Fig.4.3, blocks with lighter and darker colors belong to the owner and thieves



(a) LIFO BWoS (b) FIFO BWoS

Figure 4.3: Block-level synchronization in BWoS.

respectively. The owner grants a block to the thieves, or takes a block back from them with block advancement. More specifically, for LIFO BWoS, *get* advances to the *preceding* block (3 to 2) and takes it over from thieves; *put* grants the current one and advances to the *following* block (3 to 4). For FIFO BWoS, *get* (resp. *put*) advances and takes over (resp. grants) the following block.

The grant and takeover procedures are based on the *thief index*—an entry in the block metadata that indicates the stealing location inside the block. Takeover sets this index to the block margin with an atomic exchange, and uses the old value as the threshold between the owner and the ongoing thieves in this block. This ensures that owner is not blocked by thieves when it takes over the block. Moreover, concurrent owner and thieves never read the same data because the threshold between them is set atomically. Similarly, the grant procedure transfers the block to thieves by writing the threshold to the thief index. We will introduce the details in §4.4.2.1.

4.3.3 Round Control

Each block also records *round numbers* of the last data access. When advancing block, the current block's round is copied over to the next block; except in the

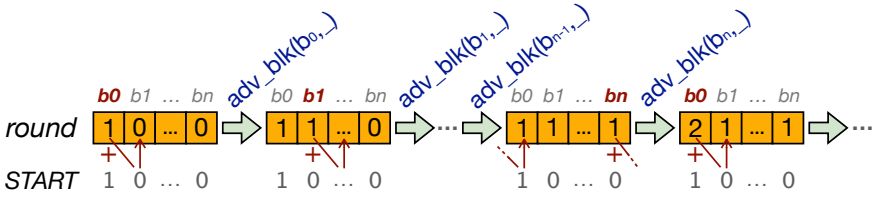


Figure 4.4: Update of round numbers in each block.

case of a wrap-around, where the block number is increased by 1 (Fig. 4.4).

In fact, there are producer, consumer, and thief round numbers in each block. When the producer tries to write round r's data into a block, the consumer and thieves must have finished reading all data with round $r-1$ from that block; so that the producer never overwrites any unread data. Similarly, when the consumer or a thief tries to read round r's data from a block, the producer's round at that block must already be r; this prevents reading any data twice, or reading data that was never written. Details can be found in §4.4.2.2.

4.3.4 Probabilistic Stealing

As discussed in §4.2.1.3, size-based policies can achieve better load balance at the cost of degrading the performance of the owner of each queue. Calculating the size is even harder in our setting because the appropriate metadata is distributed across all blocks. However, BWoS brings an opportunity to have a new size-based, probabilistic stealing policy, which can provide strong load balance without adversely affecting the owner's performance.

We ensure strong load balancing by making the probability of choosing a queue as a victim proportional to its size. We implement this approach with a two-phase algorithm: the P_{select} phase first selects a potential victim randomly, and then the P_{accept} phase decides whether to steal from it with probability $\mathcal{S}/\mathcal{C}$, where $\mathcal{S}$ is the selected queue's size and $\mathcal{C}$ is its capacity; otherwise (with probability $1-\mathcal{S}/\mathcal{C}$) it returns to P_{select} for a new iteration.

Therefore, given a pool of N queues each with the same capacity and a selector in P_{select} that selects each queue with equal probability, P_{accept} can

guarantee that the probability of a thief stealing from a queue is proportional to its size.

To minimize the impact on the owner's performance, instead of measuring $\mathcal{S}$, we estimate $\mathcal{S}/\mathcal{C}$ directly by sampling. The thief chooses a random block from *all* blocks of the queue and checks if it has data *available for stealing*, where the probability of returning true is close to $\mathcal{S}/\mathcal{C}$. As the thief reads only one block's metadata, its interference with the owner is minimal (cf. §4.2.2).

For FIFO BWoS, the above approach can achieve zero-overhead for steals: after the estimation returns true, we can steal from the block used for estimation directly, as block-local metadata enables thieves to steal from any block which has been granted to thieves. We call this instance of applying our probabilistic stealing policy to FIFO BWoS a randomized stealing procedure.

For LIFO BWoS, stealing still happens from the *bottom* block (Fig. 4.3). Thieves advance to the following block when they finish the current one. For FIFO BWoS, thieves do not advance block when randomized stealing is enabled, and fall back to the stealing policy for selection of the new queue and block instead (§4.3.4). In this case, the operation to advance to the next block on stealing (Fig. 4.2 line 36) becomes a no-op.

Moreover, we can further combine the probabilistic stealing policy with a variety of selectors for P_{select} phase (*e.g.*, from NUMA-aware policy), to benefit from both better workload balance and reduced stealing cost. Results show that the hybrid probabilistic NUMA-aware policy brings the best performance to BWoS (§4.6).

4.4 Implementation

4.4.1 Single-Block Operations (Fast Path)

Let's consider how *put*, *get*, and *steal* operations inside the block are implemented (lines 4, 15, and 28 in Fig. 4.2). Because *get* and *steal* always happen on different blocks, we only need to consider two cases of multiple operations

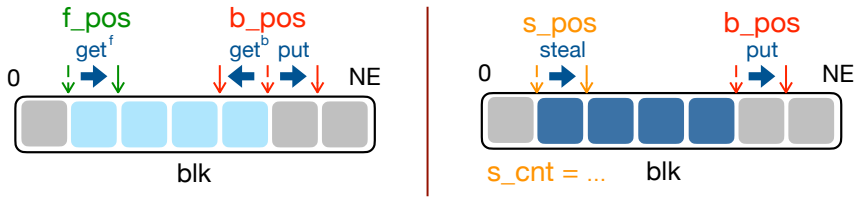


Figure 4.5: Put, get, and steal operations inside the block.

in a block: producer-consumer and producer-thieves (Fig 4.5).

To support these cases, each block has 4 metadata variables: entries which are ready for the consumer in the block are between the *front position* (f_pos) and *back position* (b_pos), while thieves use the *stealing position* (s_pos) and a counter of *finished steals* in the block (s_cnt) for coordinating among themselves and with the producer respectively.

To produce a value, *put* first checks whether it reaches the block margin NE (number of entries), if not, writes the data into the producer position (b_pos), and lets it point to the next entry.

To consume a value, there are two get operations, get^f and get^b, which correspond to the FIFO and LIFO BWoS respectively. *get* checks whether the block margin has been reached, or if the block has run out of data (f_pos has reached b_pos), if not, it reads the data and updates the consumer position variable in the block metadata. The two variants of *get* differ in which position variables and boundaries they use. get^f uses f_pos as consumer position variable, NE as block margin, and b_pos as boundary of valid data. get^b uses b_pos, zero position of the block, and f_pos for the same purposes, respectively.

Thieves follow a similar pattern: *steal* first checks if it has reached the block margin, or if the block has run out of data (s_pos has reached b_pos). Then, it updates s_pos using an atomic compare-and-swap (CAS) to point to the next entry, reads the data, and finally updates s_cnt with an atomic increment. If the CAS fails, *steal* returns *conflict*. (CAS is used because multiple thieves can operate in the same block.)

All of these operations return *block_done* when they reach a block margin. Otherwise, if the block runs out of data, *get* and *steal* return *empty*.

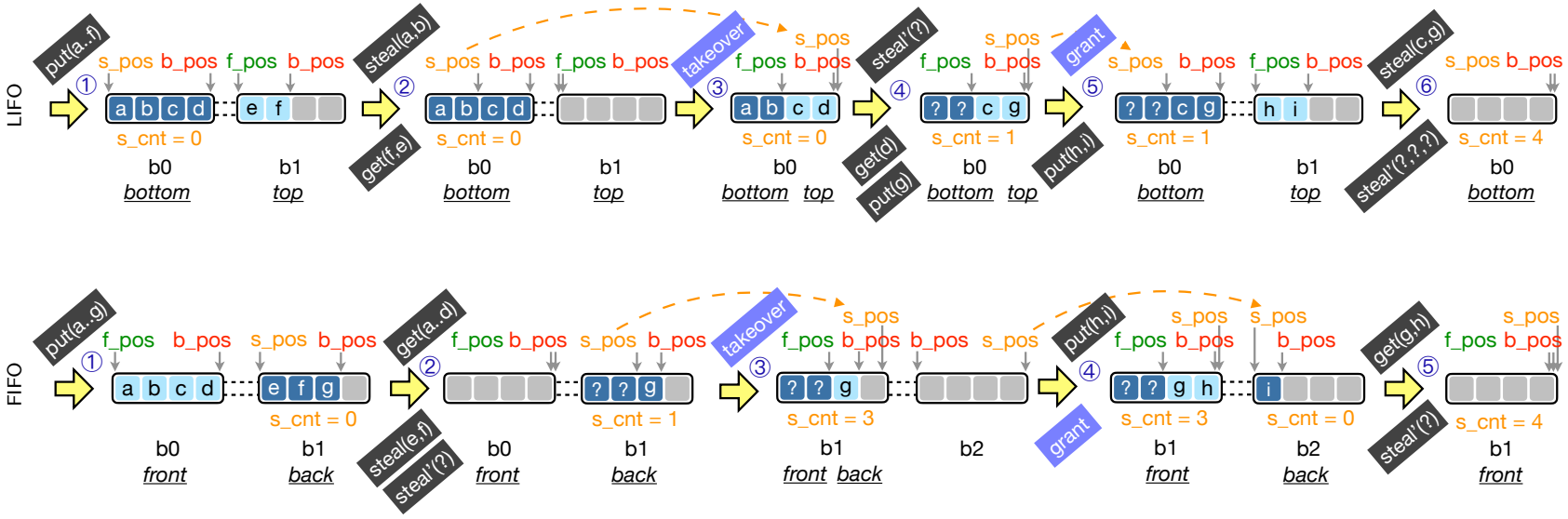


Figure 4.6: Takeover and grant procedures in block advancement.

4.4.2 Block Advancement

In case a block margin is reached, *put*, *get*, and *steal* move to the next block: They first check whether advancing is permitted by the round control, and if so, they call takeover (by *get*) or grant (by *put*) procedures, and reset block-level metadata.

4.4.2.1 Takeover and Grant Procedures

We explain the takeover and grant procedures using a queue with 4-entry blocks as an example (Fig. 4.6).

LIFO. Let us assume that 6 elements (a-f) were put into the queue. Thus, the owner is in the block b_1; b_pos in b_0 and b_1 becomes 4 and 2 respectively, while f_pos and s_pos remain at the initial value (0) (state ①). Then, two actions happen concurrently: two thieves try to steal entries, updating s_pos in b_0 to 2, and start to copy out the data (steal on Fig. 4.6), while the owner gets 3 values, consuming f, e (state ②), and advancing to b_0, thus starting the takeover. To perform the takeover, the owner atomically exchanges s_pos with the block margin (4), and then sets f_pos to the previous s_pos value (2) (state ③). After the takeover, the owner gets d and puts g. Meanwhile, one ongoing *steal* completes (steal' on Fig. 4.6), increasing s_cnt by 1 (state ④). It does not matter which of the two completes first. When the owner puts new items h and i, it grants b_0 to thieves and advances to b_1. To perform the grant, it sets s_pos to the f_pos value (2), indicating to thieves that the block is available (state ⑤). After thieves steal all entries in b_0, s_cnt reaches the block margin (state ⑥). Thus, b_0 can be reused in the next round.

FIFO. First, the producer puts 7 elements (a-g) into the queue. The producer and the consumer are in b_1 and b_0 respectively, and thieves can steal from b_1 (state ①). Then, the consumer gets all elements in b_0, and advances to b_1 (state ②). This requires taking over b_1 from thieves: for this purpose, it updates s_pos and f_pos in the same way as the LIFO BWoS, but also adds the difference between the new f_pos (2) and the block margin (i.e. length of the

block) to s_cnt (state ③). This way, when all thieves finish their operation in b_1, its s_cnt will be equal to the block margin. After that, the producer puts a new item h, and advances to b_2 granting it to thieves (state ④). Finally, both thieves and the consumer have read all entries from b_1, its f_pos and s_cnt are equal to the block margin (state ⑤). The producer uses this condition to check if the block can be reused for producing new values into it.

4.4.2.2 Round Control and Reset Procedure

To implement round control (§4.3.3), the position variables in block metadata (f_pos, b_pos, s_pos, s_cnt) contain both the index or counter (idx field) as described in §4.4.2.1 and the round number (rnd field). We fit both components into a 64-bit variable that can be updated atomically.

Consider, for example, the put operation of FIFO BWoS (Fig. 4.7). In *put*, when the producer idx reaches the block margin NE of the block blk (step ①), the new round x of the next block nblk is calculated as described in §4.3.3 (step ②). When advancing to the block nblk with the producer round x, the producer checks that the consumer and the thieves have finished reading all data from the previous round in nblk by checking if their idx fields are equal to NE and their rnd fields are $x - 1$ (step ③). When the check succeeds, the new value with the index 0 and the round x will be written into the producer position variable (step ④), thus *resetting* the block for the next round producing. Otherwise, a "queue full" condition is reported.

The get operation of the FIFO BWoS is similar. To decide whether *get* can use a next block, it checks whether the block's next consumer's round is equal to the producer round (step ③), and resets the round and index fields if the check succeeds.

Each operation resets only a subset of position variables (b_pos, f_pos, s_pos, s_cnt). We carefully select which variables each operation resets so that takeover and grant procedures by the owner have no write conflict with the reset done by thieves.

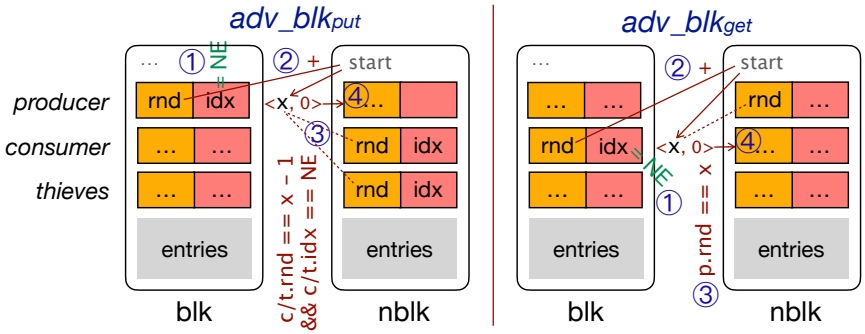


Figure 4.7: Round control in FIFO BWoS.

4.5 Verification and Optimization

The complexity of the BWoS algorithm necessitates the use of formal verification techniques to ensure that there are no lurking design or implementation bugs, and to optimize the use on WMMs. One can easily imagine several tricky cases with block advancements. For example, for LIFO BWoS, when the owner calls *puts* and *gets* and advances to the next block, it may easily trigger ABA [123] bugs during the round control and takeover.

Unlike simpler algorithms like ABP [141], it is virtually impossible to justify the correctness of an optimal memory barrier placement by inspection. Luckily, model checking tools [165, 135, 147, 111] are widely used to check the correctness of concurrent algorithms and optimize the memory barrier under WMMs automatically, improving both performance and developer confidence. For example, the Tokio library uses the model checker Loom [55, 164], which has helped them find more than 10 bugs [57].

4.5.1 Verification Client

A model checker takes as input a small *verification client* program that invokes queue operations. It verifies that all possible executions of the input program satisfy some generic correctness properties, such as memory safety and termination [91], as well as any algorithm-specific properties that are included in the verification client as *assertions*. Whenever verification fails, the model checker

```
1  class stat {
2   u64 sum = 0, buf = 1;
3   void put(queue<u64> q){
4      if (q.put(buf))
5        sum += buf;
6        buf <<= 1;
7   }
8   bool get(queue<u64> q){
9      data = q.get(buf);
10     if (data != null) {
11       sum += data;
12       return true;
13     }
14     return false;
15  }
16  void steal(queue<u64> q){
17     data = q.steal(buf);
18     if (data != null)
19       sum += data;
20  }
21 }
22 stat f, b, s1, s2;
23 queue<u64> q; // 2 * 2
24 T0: b.put(q)*3; f.get(q)*2;
25      b.put(q)*4; f.get(q)*3;
26      b.put(q)*5; f.get(q)*4;
27 T1: s1.steal(q);
28 T2: s2.steal(q)*2;
29 T3: while (f.get(q));
30     assert (b.sum == f.sum +
31                s1.sum + s2.sum);
32 (T0 || T1 || T2) ; T3
```

Figure 4.8: Verification and optimization client code.

returns a concrete erroneous execution as a counterexample.

To be able to generalize the verification result beyond the specific client program verified, the client program must trigger all possible contending scenarios and cover all desired properties. Because of the symmetry of BWoS (each owner operates on its own queue and steals from others), it suffices to verify the use of one queue owned by one thread and contended by several thief threads.

Verified properties. We have verified the following properties with the GenMC model checker [136, 135]:

- Memory safety: The program does not access uninitialized, unallocated or deallocated memory.
- Data race freedom: there are no data races on variables that are marked as non-atomic.
- Consistency: Each element written by the producer is read only once by either the consumer or thieves. No data corruption or loss occurs.
- Loop termination: Every unbounded spinloop and bounded fail-retry-loop in the program will eventually terminate even under weak memory models.

All possible executions, including those that occur due to weak memory reordering under the IMM [169] and RC11 [140] memory models, have been explored, and the aforementioned properties hold for each of them. With GenMC we were

	VERI/OPT time	memory barriers				#executions explored
		#SEQ	#ACQ	#REL	#RLX	
LIFO BWoS	62 $min.$	0	2	2	14	1.39 M
FIFO BWoS	53 $min.$	0	3	3	16	1.43 M
ABP	16 $min.$	4	3	1	7	2.05 M

Table 4.2: Statistics of the verification and optimization.

able to verify safety properties and termination of loops, but not the properties of individual operations.

Contending scenarios. As in any model checking verification, our models have a limited size within which the above properties hold. The client code for verifying and optimizing *put*, *get*, *steal* operations of BWoS is shown in Fig. 4.8. We configure the queue to have two blocks, each with the capacity of two entries (line 23). It is thus sufficient to put 5 entries to trigger the queue wraparound. We then launch 3 threads that run in parallel: The owner thread T0 has 3 rounds of *put* and *get* (lines 24-26) with different numbers of entries, trying to trigger block advancement for both producers and consumers in each round. Thief threads T1 and T2 steal one and two entries respectively, and thus together with T0, they trigger the queue empty condition, takeover, grant, and reset procedures, as well as conflicts between thieves.

Assertion and properties. After threads T0–T2 exit, thread T3 gets all remaining entries, and asserts that the sum of *put* elements is equal to the sum of elements read via *get* and *steal* (lines 30-31). Notice that the elements are generated as powers of two (line 6), therefore this assertion ensures that *each element written by the producer has been read only once*.

4.5.2 Results

We have optimized and verified the C code of LIFO and FIFO BWoS with the VSync framework and the GenMC model checker. We have also verified the ABP queue using our verification client as a baseline. The statistics are shown in

Table 4.2, broken down by memory barrier type: sequentially consistent (SEQ), acquire (ACQ), release (REL), and relaxed (RLX, i.e. plain memory accesses).

For BWoS, barrier optimization and verification finished in about an hour on a 6-core workstation [108], with over 1 million execution explorations. For ABP, the checking finishes in 16 minutes. More executions are explored for ABP since thieves and owner synchronize for every operation, which brings more interleaving cases.

Verification confidence. By adding one thread and discovering that no further barriers were required, we conclude that further increasing the thread count is unlikely to discover some missing barrier. Hence, we can avoid the state space explosion that happens with larger thread counts. On the other hand, discovering that an existing barrier had to be stronger would have forced us to review the algorithm in general.

Experience. Model checking proved itself to be invaluable during BWoS's development. For example, an early version of LIFO BWoS had a bug where thieves would reset the s_pos variable when advancing to their following block (blk). In the case when the owner is advancing to its preceding block which also happens to be blk, it would update s_pos in the takeover procedure, which conflicts with the thieves' reset procedure, resulting in data loss. This data loss was detected by GenMC with the verification client assertion (lines 30-31). We have fixed it by delegating the thieves' s_pos reset procedure to the owner, thus removing this conflict.

Optimization. For BWoS, most concurrent accesses are converted to relaxed barriers, with the few remaining cases being release or acquire barriers. For the owner's fast path that determines the performance, we have only one release barrier in the FIFO BWoS. In contrast, the highly optimized ABP [141] contains many barriers. In particular, owner operations contain 2 sequentially consistent, 1 acquire, and 1 release barriers, which significantly degrade its performance.

We note that these optimization results are optimal: relaxing any of these barriers produces a counterexample. To further increase our confidence in the

verification result, we added another thief thread stealing one entry, and checked the optimized BWoS with GenMC. BWoS passes the check in 3 days with around 200 million execution explorations.

Barrier analysis. LIFO BWoS does not contain any barriers in the fast path because the owner and the thieves do not synchronize within the same block. An acquire-release pair is related to s_pos in the owner's slow path and thieves' fast path that ensures the correctness of the takeover procedure. Another acquire-release pair is related to s_cnt which ensures the owner doesn't overwrite ongoing reading when it catches up with a thieves' block (wraparound case). For FIFO BWoS, besides the above barriers, since producer and thieves need to synchronize within a block, an additional acquire-release pair in their fast path is required.

4.6 Evaluation

Experimental setup. We perform all experiments on two x86 machines connected via 10Gbps Ethernet link, each with 88 hyperthreads (x86-88T) [130], and one Arm machine with 96 cores (arm-96T) [129]. The operating system is Ubuntu 20.04.4 LTS with Linux kernel version 5.7.0.

4.6.1 Block Size and Memory Overhead

In comparison with other queues, BWoS has extra parameters that the user needs to chose when initializing a data structure, namely the block size and the number of blocks. In our experience with both micro- and macro-benchmarks, the system's throughput remains mostly contants regardless of the block size or the number of blocks as long as they are above certain minimal values: 8 or more blocks in the queue and 64 or more elements in the block, both for our x86-88Tand arm-96Tmachines.

The reason for this insensitivity to block size change is twofold: first, since a single thread is responsible for advancing the blocks of its own queue, the

block size does not introduce any contention-related overhead. Larger block sizes cause the queue owner to advance the block less often, but after a certain block size, the overhead of advancing the block becomes negligible. Second, since BWoS forbids the owner and thieves consuming items in the same block with block-level synchronization, the contention of them on a queue is largely independent from the number of blocks. These insights guide the block size selection for our benchmarks: we set the number of blocks to 8 and calculated the block size based on the queue capacity.

Therefore, selecting an appropriate block size is straightforward. Further fine-tuning of these parameters may be beneficial for extreme scenarios where memory-size constraints are present or the overly large block size becomes detrimental to stealing (§4.8).

BWoS contains three pointers for each queue, and four atomic variables, two pointers, and one boolean variable for each block as its metadata. The actual memory usage also includes cache padding added to prevent false sharing. The memory overhead from this metadata is static and thus negligible for most use-cases.

4.6.2 Microbenchmarks

To verify our claims, we have designed a microbenchmark which supports both LIFO and FIFO work stealing and compared BWoS with the state-of-the-art algorithms: an off-the-shelf ABP [95] implementation from Taskflow v3.4.0 [73] with barrier optimization [141] (abp), the block-based bounded queue [184] (bbq), work stealing queues from Tokio v1.17.0 [76] (tokioq), Go's runtime v1.18 [74] (goroutineq), Kotlin coroutines v1.6.4 [49] (coroutineq), and Eigen v3.3 [86] (eigenq).

Each queue has a capacity of 8k entries, with 8-byte data items; BWoS is configured to have 8 blocks. We perform the following three experiments:

• Single queue without stealing (§4.6.2.1): The owner thread executes the workload in a loop: it first puts until the queue is full and then gets until the

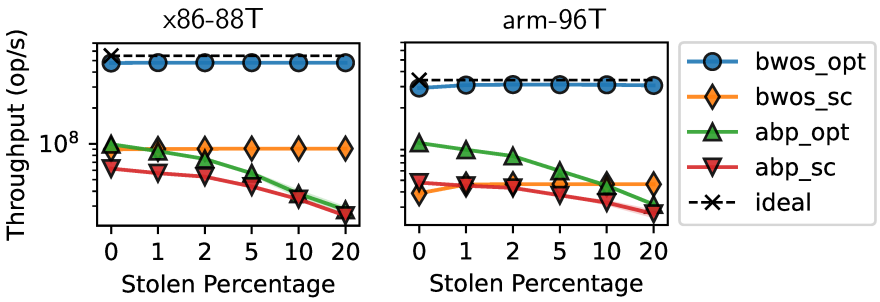


Figure 4.9: Throughput of LIFO BWoS and ABP with (opt) or without (sc) memory barrier optimization on x86-88Tand arm-96Trunning with different stolen percentages.

queue is empty.

• Single queue with stealing (§4.6.2.2): An additional thief thread calls *steal* operations on the queue in a loop. We adjust the *put*/*get* ratio, and the idle time between each *steal* to perform the experiment at varying stolen percentages[2].

• Pool consisting of 8 queues (§4.6.2.3): 8 threads perform the following operations in a loop: put items to its queue until it is full, then get until it is empty, and then attempt to steal $k * \mathcal{C}$ items from the pool, where k is the *balancing factor* (in percent), and $\mathcal{C}$ is the queue capacity. The threads are distributed equally between two NUMA domains, and within each NUMA domain between two L3 cache groups [106].

In each experiment, we measure the total throughput: the sum of *put*, *get*, and *steal* operation throughputs (ops/sec).

4.6.2.1 Queue without Stealing

Overall performance. Figures 4.9 and 4.10 for stolen percentage equal to 0 show the performance of the queue without stealing. BWoS outperforms other algorithms by a significant margin. For example, LIFO BWoS (bwos_opt) has 4.55x higher throughput than ABP (abp_opt) on x86-88T, and FIFO BWoS written in C/C++, Rust, Go, and Kotlin outperform bbq in C, eigenq in C++,

[2]The thief thread is located in the same L3 cache group as the owner; the results are similar when putting the thief thread elsewhere.

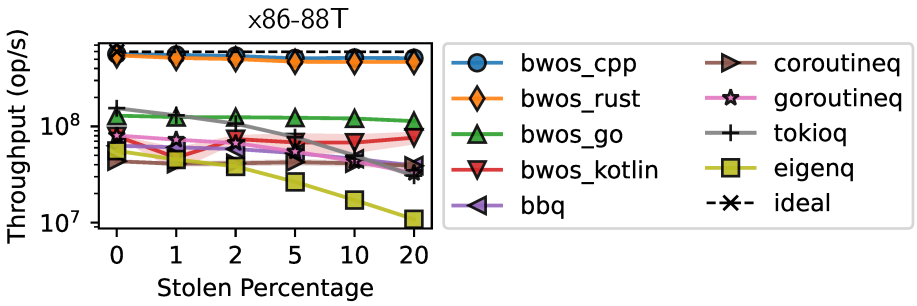


Figure 4.10: Throughput of FIFO BWoS and other state-of-the-art FIFO work stealing algorithms.

tokioq in Rust, goroutineq in Go, coroutineq in Kotlin by 8.9x, 10.15x, 3.55x, 1.61x, and 1.82x accordingly.

Impact of the memory barrier optimization. abp and LIFO BWoS get 1.65x and 5.39x speedup on x86-88T, and 2.03x and 3.38x speedup on arm-96Trespectively due to the memory barrier optimization. We observe similar results for FIFO work stealing algorithms[3]. The much greater speedup of BWoS compared to ABP is possible in particular due to the separation of fast path and block advancement, where most of the barriers in the fast path become relaxed.

Effectiveness of the block-level synchronization. Results show that on x86-88TLIFO and FIFO BWoS are only 10.7% and 5.4% slower than ideal, respectively. On arm-96T the results are similar. Thus, block-level synchronization allows BWoS to approach the theoretical upper bound by removing the consumer-thief synchronization from the fast path.

4.6.2.2 Queue with Stealing

Overall performance. As the stolen percentage increases, BWoS continues to outperform other work-stealing algorithms. For example, with 10% stolen percentage, LIFO BWoS outperforms abp by 12.59x, while FIFO BWoS outperforms bbq, eigenq, tokioq, goroutineq, coroutineq by 11.2x, 30.1x, 9.41x,

[3]Notice here bwos_go does not have barrier optimization because Go does not expose an interface for relaxed atomics. However, for the macrobenchmarks we apply the barrier optimization by using the Go internal atomic library.

2.78x, and 1.64x respectively.

Effectiveness of the block-based approach. Unlike other algorithms, BWoS suffers only a minor performance drop as the stolen percentage increases. For example, for 20% stolen percentage, the throughput of LIFO and FIFO BWoS drops only by 0.53% and 9.35%, while for abp_opt, tokioq and goroutineq it degrades by 71.9%, 80.2%, and 59.3% respectively. Note that the BBQ concurrent FIFO queue [184], which is also a block-based design, does not reach performance comparable to BWoS, stressing the importance of our design decisions for the work stealing workloads.

4.6.2.3 Pool with Different Stealing Policies

Stealing policies. We perform this experiment with 6 stealing policies, namely the random choice policy (rand), a policy that chooses the victim based on a static configuration (seq), a policy that chooses the last selected one as the victim [182] (last), best of two (best_of_two), best of many (best_of_many), and NUMA-aware policy (numa). For best_of_many we choose best of half (i.e. best of four).

Overall performance. In this experiment, we compare BWoS only with the second-best algorithm from the previous experiments: abp and tokioq for LIFO and FIFO work stealing respectively. Fig. 4.11 shows that BWoS performs consistently better than other algorithms. When the balancing factor is 0%, BWoS outperforms abp by 4.69x and tokioq by 2.68x. As the balancing factor increases, the throughput of BWoS variants is 7.90x higher than of abp and 6.45x higher than of tokioq.

Impact of the NUMA-aware policy. LIFO and FIFO BWoS with numa policy outperform BWoS with other policies by at most 2.21x and 1.73x respectively. For other work stealing algorithms, best_of_two brings the best performance. Thus, BWoS benefits from numa policy while other algorithms do not. On the other hand, in many cases best_of_many brings the worst performance, proving that interference with the owner can outweigh its improvements to the load

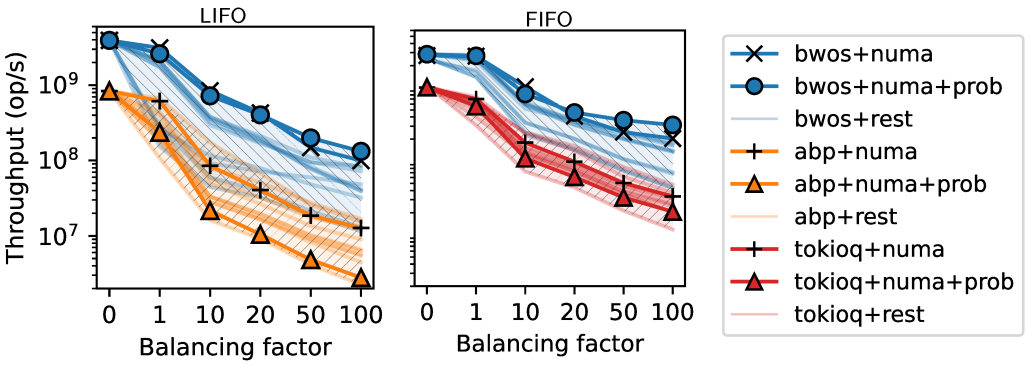


Figure 4.11: Throughput of the pool (8 queues) with different stealing policies and different balancing factors on x86-88T. rest refers to all non-numa policies with and without probabilistic stealing.

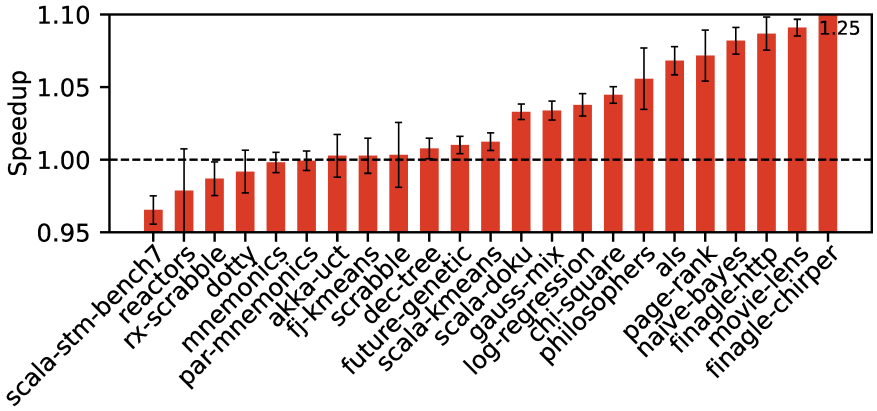


Figure 4.12: Speedup of 23 benchmarks from Renaissance benchmark suite on x86-88T.

balance.

Effectiveness of the probabilistic stealing. BWoS can additionally benefit from the probabilistic stealing. When the balancing factor is 100%, numa with probabilistic stealing (bwos+numa+prob) brings 1.34x, 1.53x performance improvement on average to LIFO and FIFO BWoS.

4.6.3 Macrobenchmarks

4.6.3.1 Java G1GC

We replace the task queue [46] in Java 19 HotSpot [75] with LIFO BWoS, and run the Renaissance benchmark suite v0.14.0 [67], which consists of 25

modern, real-world, and concurrent benchmarks [170] designed for testing and optimizing garbage collectors. Two database benchmarks are omitted since they don't support JDK 19. JVM enables `-XX:+DisableExplicitGC` [126, 63] and `-XX:+UseG1GC` flags when running the benchmark. All other parameters (*e.g.*, number of GC threads, VM memory limit) are default. We run 10 iterations for each benchmark with the modified and the original JVM, and measure the end-to-end program run time via the Renaissance testing framework.

Figure 4.12 shows the speedup of all 23 benchmarks on x86-88T. When BWoS is enabled, 17 of them get performance improvement. The average speedup of all benchmarks is 3.55% and the maximum speedup is 25.3%. The applications that benefit more from concurrent GC also get greater speedup from BWoS. Results on arm-96Tare similar where the average speedup is 5.20%, 18 benchmarks are improved and the maximum speedup is 17.2%.

On the other hand, several Renaissance benchmarks did not get any performance improvement from using BWoS. We have investigated this issue by running JVM with flags `-Xlog:gc+cpu` and `-Xlog:gc+heap+exit` to collect GC-related statistics. These experiments have shown that applications that trigger GC often demonstrate improvement from BWoS, while applications that don't trigger GC or triggered it only rarely (e.g. at JVM exit) see no speedup. For the benchmarks which never or seldomly trigger the GC, the slowdown is most likely due to the longer queue initialization.

4.6.3.2 Rust Tokio Runtime

We replace the run queue [77] in Tokio v1.17.0 [76] with FIFO BWoS, and run Hyper HTTP server v0.14.18 [38] and Tonic gRPC server v0.6.2 [78] with the modified runtime. Tokio runtime (also Go runtime) provides a batch stealing interface. Based on observations from benchmarks similar to Fig. 4.1d, we configured the thief of BWoS to steal all available entries from its block at once. Benchmarks are performed on two x86-88Tmachines, one running the server, the other running the HTTP benchmarking tool wrk v4.2.0 [84] or the

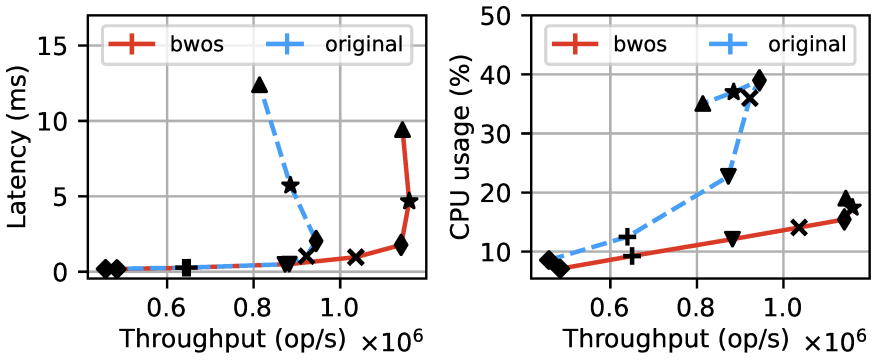


Figure 4.13: Throughput and latency results of Hyper HTTP server with BWoS and the original algorithm.

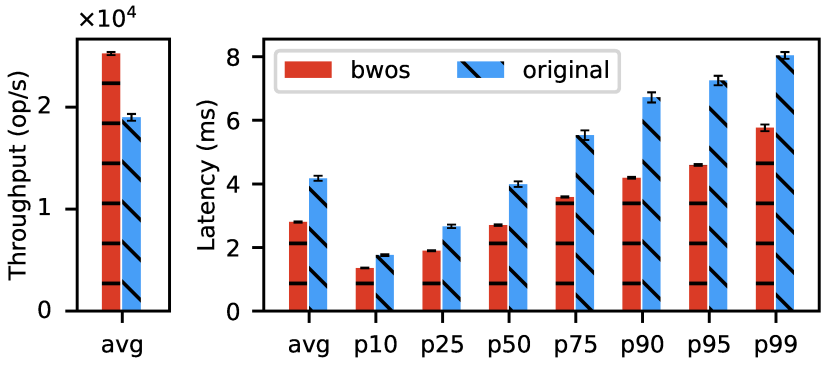


Figure 4.14: Throughput and latency of Tonic gRPC server with BWoS and the original algorithm.

gRPC benchmarking and load testing tool ghz v0.017 [31]. All parameters of Hyper and Tonic are default. Each benchmark runs 100 seconds and has 10 iterations. The latency and throughput are measured by wrk or ghz, while the CPU utilization of the server is collected through the Python psutil library [66]. wrk and ghz run the echo workload and SayHello protocol respectively and are configured to utilize all hyperthreads of their machine.

Figure 4.13 shows the throughput-latency and throughput-CPU utilization results of Hyper with different connection numbers (100, 200, 500, $1k$, $2k$, $5k$, and $10k$). Before the system is overloaded, BWoS provides 1.14×10^6 op/s throughput while dropping 60.4% CPU usage with similar latency, the original algorithm provides only 9.44×10^5 op/s throughput. With 1k connections, BWoS increases throughput by 12.3% with 6.74% lower latency and 60.9%

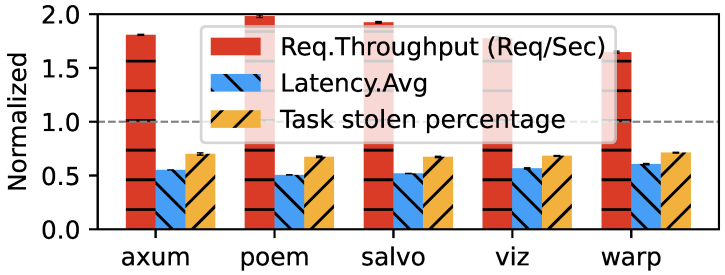


Figure 4.15: Request throughput, average latency, and task stolen percentage comparison results of 5 Rust web frameworks of BWoS (normalized to the original algorithm results) with rust-web-benchmarks workload on x86-88T.

lower CPU utilization.

Figure 4.14 shows the throughput and latency results of Tonic. Using BWoS increases throughput by 32.9%, with 32.8% lower average latency and 36.6% lower P95 latency.

To prove the generality of BWoS when applied to web frameworks, we also benchmark another 5 popular Rust web frameworks [10, 64, 70, 81, 82] that used Tokio runtime with rust-web-benchmarks [11] workload on x86-88T(Fig. 4.15). Results show that BWoS increases the throughput by 82.7% while dropping 45.1% of average latency. In addition, the task stolen percentage drops from 69.0% to 49.2%. We have made our implementation for the Tokio runtime available to the open-source community [4].

4.6.3.3 Go Runtime

We replace the runqueue [34] in the Go programming language [74] v1.18.0 runtime with BWoS and benchmark 9 JSON libraries [47, 25, 32, 2, 36, 24, 27, 26, 37]. The benchmark suite [33] comes from the go-json library and runs 3 iterations with default parameters. We record the latency of each operation (*e.g.*, encoding/decoding small/medium/large JSON objects) reported by the benchmark suite, and calculate the speedup.

As shown in Fig. 4.16, when BWoS is enabled, operations get 25.8% average performance improvement on x86-88T. arm-96Tproduces similar results with 28.2% speedup on average. In general, encoding operations have better speedup

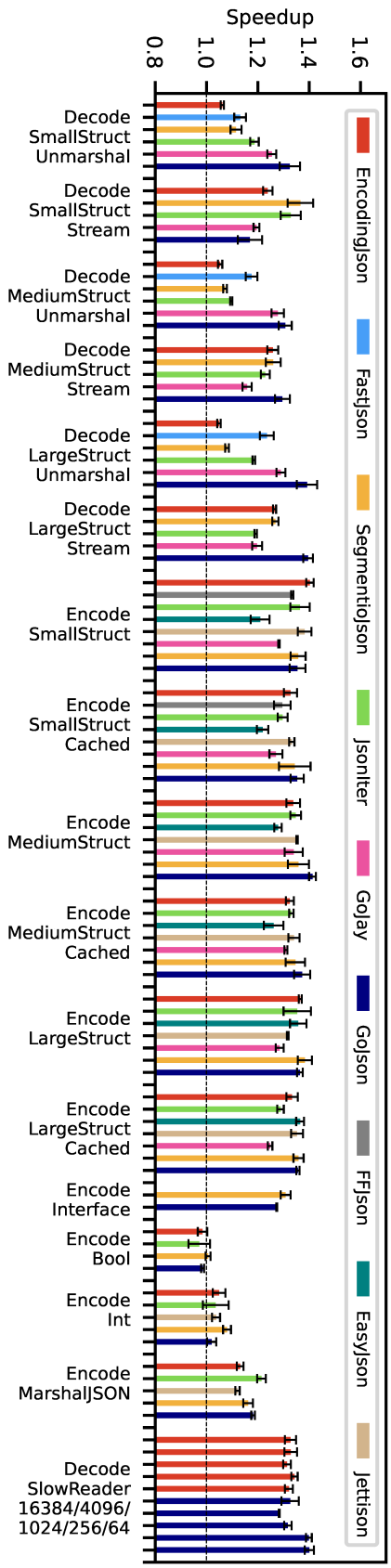


Figure 4.16: Speedup in the go-json library benchmark on x86-88T from using BWoS.

compared to decoding operations. We observe no improvement for encoding booleans and integers.

4.7 Related Work

Block-based queues. Wang *et al.* proposed a block-based bounded queue [184] (BBQ) that splits the buffer into multiple blocks, thus reducing the producer-consumer interference. BWoS differs from BBQ in the following ways: (1) although BBQ also applies metadata separation, the producer-consumer interference it reduces is not an issue for work stealing as these always execute on the same core. By introducing block-level synchronization, steal-from-middle property, and randomized stealing, FIFO BWoS outperforms BBQ by a large margin (§4.6). (2) For the round control in BWoS, the new round of a block is determined only by the round of its adjacent block instead of relying on global metadata, as the *version* mechanism in BBQ does. This design simplifies the round updating and reduces its overhead.

Owner-thief interference and synchronization costs. Attiya *et al.* proved that work stealing in general requires strong synchronization between the owner and thieves [96]. BWoS overcomes this issue by delegating this synchronization to the block advancement, thus removing it from the fast path. Acar *et al.* used a sequential deque with message passing to remove the owner's barrier overhead [89]. However, this design relies on explicit owner-thief communication, thus the steal operation cannot run to completion in parallel with the owner's operations. Dijk *et al.* proposed a deque-based LIFO work-stealing algorithm which splits the deque into owner and thief parts, thus reducing the owner's memory fences when they do not reach the queue split point [110]. However, the entries read by thieves cannot be reused until the whole deque is empty. Horie *et al.* proposed a similar idea, where each owner has a public queue that is accessible from other threads and a private queue that is only accessible by itself [126]. However, it requires more effort to deal with load balancing, *e.g.*,

introducing global statistics metadata which causes more cache misses for the owner. In contrast, BWoS reduces the interference using techniques of block-level synchronization, and probabilistic and randomized stealing. Morrison *et al.* introduced work stealing algorithms which rely on the bounded TSO microarchitectural model, which x86 and SPARC CPUs were shown to possess [158]. Michael *et al.* reduced the thief-owner synchronization by allowing them to read the same task [153], which requires reengineering of tasks to be idempotent. BWoS exhibits correct and efficient execution on a wide range of CPU architectures without any additional requirements.

Stealing policies. Yang *et al.* gave a survey of scheduling parallel computations by work stealing [186]. Kumar *et al.* benchmarked and analyzed variations of stealing policies [137]. Mitzenmacher proposed to give the thief two choices for selecting the victim to have a better load balancing [155]. Most of the analyzed policies are size-based, and thus aim to reach the same goal as our probabilistic stealing policy—namely, better load balance. Hendler *et al.* allow thieves to steal half of the items in a given queue once to reduce interference [121]. BWoS supports batched stealing, but the maximum amount of data that can be stolen atomically is a block. However, the stealing policy can be configured to steal more than one block. Kumar *et al.* proposed a NUMA-aware policy for work stealing [138]. This policy is fully orthogonal to BWoS and can be combined with its probabilistic stealing policy.

Formally verified work stealing. Lê *et al.* [141] manually verified and optimized the memory barriers of Chase-Lev dequeue [101] on WMMs. Unlike the verification of BWoS which relies on model checking, manual verification is a high-effort undertaking. In the context of concurrent queues, Meta's FollyQ was verified using interactive theorem prover [183]. While this approach provides the highest levels of confidence in the design, it works only with sequentially consistent memory model, and is also a high-effort endeavor. Recently, GenMC authors have verified the ABP queue as part of evaluation of their model checker [135]. The authors of BBQ have relied on VSync to simultaneously ver-

ify and optimize the barrier for weak memory models [184]. BWoS also uses VSync for this purpose, but instead of many hand-crafted tests, which exercise the individual corner cases in BBQ, we create one comprehensive client that covers several corner cases and their interactions at once. We further verify the optimization results by adding one more thief into the verification client and checking it with GenMC.

4.8 Summary

In this chapter, we present BWoS, a formally verified block-based work stealing solution. By introducing the block queue structure, we have eliminated the costly barriers from the fast path and significantly reduced the contention between the queue owner and the thieves. Our evaluation shows that BWoS outperforms other work-stealing queues by an order of magnitude in microbenchmarks, and improves the performance of commonly used industrial systems by up to 15–60% on average depending on the workload.

Chapter 5

Conclusion and Future Work

In conclusion, we present two key insights gleaned from our research.

The benefit of block-Based data structures is multifaceted. Firstly, by substituting global metadata with block-level metadata, we can eliminate interference among different threads operating on various parts of the queue. This includes scenarios such as producer-consumer interference in BBQ, producers (or consumers) interference in CNQ, and owner-thief interference in BWoS. Secondly, the introduction of local metadata enables the independent management of each block, facilitating the incorporation of additional features. For instance, in BWoS, we implement block-level synchronization and steal-from-the-middle capabilities. This approach also opens possibilities for holistic optimization of the data structure's use, as we demonstrate with our probabilistic stealing policy in BWoS. Lastly, as illustrated in CNQ, the block-based design offers modular verification, allowing for separate verification steps for individual blocks and their composition.

Verified software can outperform unverified software. The more hardware details and tweaks are reflected in the software, the more complex and opaque that piece of code becomes. The interaction of this complexity with concurrency and weak memory consistency presents a major challenge. We contend that practical verification tools (i.e., tools applied to increase confidence in correctness) are a key enabler in the development of efficient, and inevitably

complex, concurrent software.

Future Work. There are several directions for further work: We plan to further explore the performance trade-offs for BWoS: if the number of outstanding work items is smaller than the block size, BWoS can prevent stealing and thus limit the achieved parallelism. Furthermore, if the queue capacity has to be very small (due to space requirements), it may be necessary to reduce the block size and thus incur more block advancement that leads to a performance drop. These situations would benefit from more exploration in the system design. Additionally, we plan to extend our investigation of CNQ across various contexts, such as tracing systems [56, 29].

Bibliography

[1] 42 Years of Microprocessor Trend Data. https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/.

[2] A high-performance 100% compatible drop-in replacement of "encoding/json". https://github.com/json-iterator/go.

[3] ABA in local queue. https://github.com/tokio-rs/tokio/issues/5041.

[4] Add BWoS-queue backend to tokio. https://github.com/tokio-rs/tokio/pull/5283.

[5] AMD EPYC 9754 Processor. https://www.amd.com/en/products/cpu/amd-epyc-9754.

[6] Apache Commons. http://commons.apache.org/.

[7] API NetWorks Accelerates Use of HyperTransport Technology With Launch of Industry's First HyperTransport Technology-to-PCI Bridge Chip. https://web.archive.org/web/20061010070210/http://www.hypertransport.org/consortium/cons_pressrelease.cfm?RecordID=62.

[8] Arm A64 Instruction Set Architecture. https://developer.arm.com/documentation/ddi0596/2021-09.

[9] Arm architecture reference manual armv8, for a-profile architecture. https://developer.arm.com/documentation/ddi0553/latest.

[10] axum: Ergonomic and modular web framework built with Tokio, Tower, and Hyper. https://github.com/tokio-rs/axum.

[11] Benchmarking web frameworks written in rust with rewrk tool. https://github.com/programatik29/rust-web-benchmarks.

[12] Boost C++ Libraries. https://www.boost.org/.

[13] BPF ring buffer. https://www.kernel.org/doc/html/latest/bpf/ringbuf.html.

[14] C++ Atomic operations library. https://en.cppreference.com/w/cpp/atomic/atomic.

[15] Cascade Lake - Microarchitectures - Intel. https://en.wikichip.org/wiki/intel/microarchitectures/cascade_lake.

[16] Cisco Layer Two Forwarding (Protocol) "L2F". https://datatracker.ietf.org/doc/html/rfc2341.

[17] Client-Server Multi-process Example. https://doc.dpdk.org/guides/sample_app_ug/multi_process.html.

[18] Data Plane Development Kit. https://www.dpdk.org/.

[19] Data Plane Development Kit Test Suite. https://doc.dpdk.org/dts/gsg/.

[20] dpdk/drivers/net/ring. https://github.com/DPDK/dpdk/tree/main/drivers/net/ring.

[21] dpdk/lib/eventdev. https://github.com/DPDK/dpdk/tree/main/lib/eventdev.

[22] Dual Core Era Begins, PC Makers Start Selling Intel-Based PCs. https://www.intel.com/pressroom/archive/releases/2005/20050418comp.htm.

[23] Efficient IO with io_uring. https://kernel.dk/io_uring.pdf.

[24] Fast JSON encoder/decoder compatible with encoding/json for Go. https://github.com/goccy/go-json.

[25] Fast JSON parser and validator for Go. https://github.com/valyala/fastjson.

[26] Fast JSON serializer for golang. https://github.com/mailru/easyjson.

[27] faster JSON serialization for Go. https://github.com/pquerna/ffjson.

[28] Folly: Facebook Open-source Library. https://github.com/facebook/folly.

[29] ftrace - Function Tracer. https://www.kernel.org/doc/html/v4.17/trace/ftrace.html.

[30] Garbage First Garbage Collector Tuning. https://www.oracle.com/technical-resources/articles/java/g1gc.html.

[31] ghz: gRPC benchmarking and load testing tool. https://github.com/bojand/ghz.

[32] Go package containing implementations of efficient encoding, decoding, and validation APIs. https://github.com/segmentio/encoding.

[33] GoJson benchmarks. https://github.com/goccy/go-json/tree/master/benchmarks.

[34] golang run-queue. https://github.com/golang/go/blob/master/src/runtime/proc.go.

[35] Guava: Google Core Libraries for Java. https://github.com/google/guava.

[36] high performance JSON encoder/decoder with stream API for Golang. https://github.com/francoispqt/gojay.

[37] Highly configurable, fast JSON encoder for Go. https://github.com/wI2L/jettison.

[38] Hyper: An HTTP library for Rust. https://github.com/hyperium/hyper.

[39] Intel Memory Latency Checker v3.9a. https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html.

[40] Intel oneAPI Threading Building Blocks. https://www.intel.com/content/www/us/en/developer/tools/oneapi/onetbb.html.

[41] Intel QuickPath Interconnect. https://www.intel.com/content/www/us/en/io/quickpath-technology/quickpath-technology-general.html.

[42] io_uring source code. https://elixir.bootlin.com/linux/v5.14-rc6/source/fs/io_uring.c.

[43] io_uring_enter - initiate and/or complete asynchronous I/O. https://unixism.net/loti/ref-iouring/io_uring_enter.html.

[44] Java Development Kit. https://jdk.java.net/.

[45] Java Native Interface Specification. https://docs.oracle.com/en/java/javase/11/docs/specs/jni/intro.html.

[46] JDK task queue. https://github.com/openjdk/jdk/blob/master/src/hotspot/share/gc/shared/taskqueue.hpp.

[47] json package - encoding/json. https://pkg.go.dev/encoding/json.

[48] Kunpeng 920 Chipset. https://www.hisilicon.com/en/products/Kunpeng/Huawei-Kunpeng/Huawei-Kunpeng-920.

[49] Library support for Kotlin coroutines. https://github.com/Kotlin/kotlinx.coroutines.

[50] liburing. https://github.com/axboe/liburing.

[51] Linux Kernel Circular Buffers. https://www.kernel.org/doc/html/latest/core-api/circular-buffers.html.

[52] Linux.org. https://www.linux.org.

[53] LMAX Disruptor: A High Performance Inter-Thread Messaging Library. https://github.com/LMAX-Exchange/disruptor.

[54] Lockless Ring Buffer Design. https://www.kernel.org/doc/Documentation/trace/ring-buffer-design.txt.

[55] Loom: Permutation testing for concurrent code. https://docs.rs/crate/loom/0.2.4.

[56] LTTng: an open source tracing framework for Linux. https://lttng.org/.

[57] Making the Tokio scheduler 10x faster. https://tokio.rs/blog/2019-10-scheduler.

[58] MESI and MOESI protocols. https://developer.arm.com/documentation/den0013/latest/Multi-core-processors/Cache-coherency/MESI-and-MOESI-protocols.

[59] Microsoft Windows. https://www.microsoft.com/en-us/windows.

[60] MPMC Queue. https://github.com/facebook/folly/blob/main/folly/MPMCQueue.h.

[61] MySQL. https://www.mysql.com.

[62] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page.

[63] Performance Tuning Guide. https://docs.oracle.com/cd/E19159-01/819-3681/abeih/index.html.

[64] Poem Framework: A full-featured and easy-to-use web framework with the Rust programming language. https://github.com/poem-web/poem.

[65] Producer/consumer synchronization modes. https://doc.dpdk.org/guides/prog_guide/ring_lib.html#producer-consumer-synchronization-modes.

[66] psutil - PyPI. https://pypi.org/project/.

[67] Renaissance Suite. https://renaissance.dev/.

[68] RISC-V. https://riscv.org/.

[69] Rust Programming Language. https://www.rust-lang.org/.

[70] Salvo: A powerful and simplest web server framework in Rust world. https://github.com/salvo-rs/salvo.

[71] std::mutex. https://en.cppreference.com/w/cpp/thread/mutex.

[72] Storage Performance Development Kit. https://spdk.io.

[73] Taskflow: A General-purpose Parallel and Heterogeneous Task Programming System. https://github.com/taskflow/taskflow.

[74] The Go programming language. https://go.dev/.

[75] The HotSpot Group. http://openjdk.java.net/groups/hotspot.

[76] Tokio: A runtime for writing reliable asynchronous applications with Rust. https://github.com/tokio-rs/tokio.

[77] Tokio run-queue. https://github.com/tokio-rs/tokio/blob/master/tokio/src/runtime/scheduler/multi_thread/queue.rs.

[78] Tonic: A native gRPC client & server implementation with async/await support. https://github.com/hyperium/tonic.

[79] TRex: Realistic Traffic Generator. https://trex-tgn.cisco.com/.

[80] Unread entries potentially lost in buf_ring after ABA condition. https://bugs.freebsd.org/bugzilla/show_bug.cgi?id=246475.

[81] Viz: Fast, flexible, lightweight web framework for Rust. https://github.com/viz-rs/viz.

[82] warp: A super-easy, composable, web server framework for warp speeds. https://github.com/seanmonstar/warp.

[83] What is NUMA? https://www.kernel.org/doc/html/v4.18/vm/numa.html.

[84] wrk: Modern HTTP benchmarking tool - GitHub. https://github.com/wg/wrk.

[85] DPDK Ring Library. https://doc.dpdk.org/guides/prog_guide/ring_lib.html, 2023.

[86] Eigen: a C++ template library for linear algebra. https://eigen.tuxfamily.org/, 2023.

[87] Intel 64 and IA-32 Architectures Software Developer's Manual. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html, 2023.

[88] Source code of DPDK ring buffer. https://github.com/DPDK/dpdk/blob/releases/lib/ring/rte_ring_elem_pvt.h, 2023.

[89] Umut A Acar, Arthur Charguéraud, and Mike Rainey. Scheduling parallel programs by work stealing with private deques. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 219–228, 2013.

[90] Yehuda Afek, Guy Korland, and Eitan Yanovsky. Quasi-linearizability: Relaxed consistency for improved concurrency. In *International Conference on Principles of Distributed Systems*, pages 395–410. Springer, 2010.

[91] Bowen Alpern and Fred B Schneider. Defining liveness. *Information processing letters*, 21(4):181–185, 1985.

[92] Amazon Web Services. AWS Graviton Processor – Enabling the best price performance in Amazon EC2, 2020. `https://mysqlonarm.github.io/ARM-LSE-and-MySQL/`.

[93] Thomas E Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, 1990.

[94] Sergei Arnautov, Pascal Felber, Christof Fetzer, and Bohdan Trach. FFQ: A fast single-producer/multiple-consumer concurrent FIFO queue. In *2017 IEEE International Parallel and Distributed Processing Symposium, (IPDPS 2017)*, pages 907–916, 2017.

[95] Nimar S Arora, Robert D Blumofe, and C Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of computing systems*, 34(2):115–144, 2001.

[96] Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin T. Vechev. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 487–498. ACM, 2011.

[97] Suparna Bhattacharya, Steven Pratt, Badari Pulavarty, and Janet Morgan. Asynchronous i/o support in linux 2.5. In *Proceedings of the Linux Symposium*, pages 371–386, 2003.

[98] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *ACM SigPlan Notices*, 30(8):207–216, 1995.

[99] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.

[100] Daniel Cederman and Philippas Tsigas. On dynamic load balancing on graphics processors. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 57–64, 2008.

[101] David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 21–28, 2005.

[102] Sanjay Chatterjee, Max Grossman, Alina Sbîrlea, and Vivek Sarkar. Dynamic task parallelism with a GPU work-stealing runtime system. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 203–217. Springer, 2011.

[103] Inho Cho, Ahmed Saeed, Joshua Fried, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. Overload Control for μs-scale RPCs with Breakwater. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 299–314, 2020.

[104] Guojing Cong, Sreedhar Kodali, Sriram Krishnamoorthy, Doug Lea, Vijay Saraswat, and Tong Wen. Solving large, irregular graph problems using adaptive work-stealing. In *2008 37th International Conference on Parallel Processing*, pages 536–545. IEEE, 2008.

[105] Melvin E Conway. A multiprocessor system design. In *Proceedings of the November 12-14, 1963, fall joint computer conference*, pages 139–146, 1963.

[106] Rafael Lourenco de Lima Chehab, Antonio Paolillo, Diogo Behrens, Ming Fu, Hermann Härtig, and Haibo Chen. CLOF: A compositional lock framework for multi-level NUMA systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 851–865, 2021.

[107] Damian Dechev, Peter Pirkelbauer, and Bjarne Stroustrup. Understanding and effectively preventing the ABA problem in descriptor-based lock-free designs. In *2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 185–192. IEEE, 2010.

[108] Dell. Precision 5820 Tower Spec. `https://i.dell.com/sites/csdocuments/Shared-Content_data-Sheets_Documents/en/us/Precision-5820-Tower-Spec-Sheet.pdf`.

[109] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. Garbage-first garbage collection. In *Proceedings of the 4th international symposium on Memory management*, pages 37–48, 2004.

[110] Tom van Dijk and Jaco C Pol. Lace: non-blocking split deque for work-stealing. In *European Conference on Parallel Processing*, pages 206–217. Springer, 2014.

[111] Natalia Gavrilenko, Hernán Ponce-de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. BMC for weak memory models: Relation analysis for compact SMT encodings. In *International Conference on Computer Aided Verification*, pages 355–365. Springer, 2019.

[112] John L Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[113] Andreas Haas. *Fast concurrent data structures through timestamping*. PhD thesis, PhD thesis, University of Salzburg, Salzburg, Austria, 2015.

[114] Andreas Haas, Thomas A Henzinger, Andreas Holzer, Christoph Kirsch, Michael Lippautz, Hannes Payer, Ali Sezgin, Ana Sokolova, and Helmut Veith. Local linearizability for concurrent container-type data structures. *Leibniz International Proceedings in Informatics*, 59, 2016.

[115] Andreas Haas, Thomas Hütter, Christoph M Kirsch, Michael Lippautz, Mario Preishuber, and Ana Sokolova. Scal: A benchmarking suite for concurrent data structures. In *International Conference on Networked Systems*, pages 1–14. Springer, 2015.

[116] Andreas Haas, Michael Lippautz, Thomas A Henzinger, Hannes Payer, Ana Sokolova, Christoph M Kirsch, and Ali Sezgin. Distributed queues in shared memory: multicore performance and scalability through quantitative relaxation. In *Proceedings of the ACM International Conference on Computing Frontiers*, pages 1–9, 2013.

[117] Robert H Halstead Jr. Implementation of Multilisp: Lisp on a multiprocessor. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 9–17, 1984.

[118] Tim Harris and Stefan Kaestle. Callisto-RTS: Fine-grain parallel loops. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 45–56, 2015.

[119] Tim Harris, Martin Maas, and Virendra J Marathe. Callisto: Co-scheduling parallel runtime systems. In *Proceedings of the Ninth European Conference on Computer Systems*, pages 1–14, 2014.

[120] Timothy L Harris. A pragmatic implementation of non-blocking linked-lists. In *International Symposium on Distributed Computing*, pages 300–314. Springer, 2001.

[121] Danny Hendler and Nir Shavit. Non-blocking steal-half work queues. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 280–289, 2002.

[122] John Hennessy and David Patterson. A new golden age for computer architecture: domain-specific hardware/software co-design, enhanced. In *ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018.

[123] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, USA, 2011.

[124] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[125] Mark D Hill and Michael R Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008.

[126] Michihiro Horie, Hiroshi Horii, Kazunori Ogata, and Tamiya Onodera. Balanced double queues for GC work-stealing on weak memory models. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on Memory Management*, pages 109–119, 2018.

[127] Michihiro Horie, Kazunori Ogata, Mikio Takeuchi, and Hiroshi Horii. Scaling up parallel GC work-stealing in many-core environments. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management*, pages 27–40, 2019.

[128] David Hovemeyer, William Pugh, and Jaime Spacco. Atomic instructions in java. In *European Conference on Object-Oriented Programming*, pages 133–154. Springer, 2002.

[129] Huawei. 2280 Balanced Model - Huawei Enterprise. `https://e.huawei.com/uk/products/servers/taishan-server/taishan-2280-v2`.

[130] Huawei. FusionServer Pro 2288X V5 Rack Server. `https://support-it.huawei.com/server-3d/res/server/2288xv5/index.html?lang=en`.

[131] Christoph M Kirsch, Michael Lippautz, and Hannes Payer. Fast and scalable, lock-free k-fifo queues. In *International Conference on Parallel Computing Technologies*, pages 208–223. Springer, 2013.

[132] Donald Ervin Knuth. *The Art of Computer Programming*, volume 3. Pearson Education, 1997.

[133] Alex Kogan and Erez Petrank. A methodology for creating fast wait-free data structures. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 141–150. ACM, 2012.

[134] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2P2: Making RPCs first-class datacenter citizens. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 863–880, 2019.

[135] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. Model checking for weakly consistent libraries. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 96–110, New York, NY, USA, 2019. Association for Computing Machinery.

[136] Michalis Kokologiannakis and Viktor Vafeiadis. GENMC: A model checker for weak memory models. In *International Conference on Computer Aided Verification*, pages 427–440. Springer, 2021.

[137] Saurav Kumar and Aryabartta Sahu. Benchmarking and analysis of variations of work stealing scheduler on clustered system. In *2014 15th International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 28–35. IEEE, 2014.

[138] Vivek Kumar. PufferFish: NUMA-Aware Work-stealing Library using Elastic Tasks. In *2020 IEEE 27th International Conference on High Perfor-*

mance Computing, Data, and Analytics (HiPC), pages 251–260. IEEE, 2020.

[139] Ori Lahav, Egor Namakonov, Jonas Oberhauser, Anton Podkopaev, and Viktor Vafeiadis. Making weak memory models fair. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021.

[140] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in C/C++11. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 618–632. ACM, 2017.

[141] Nhat Minh Lê, Antoniu Pop, Albert Cohen, and Francesco Zappa Nardelli. Correct and efficient work-stealing for weak memory models. In *18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2013*, pages 69–79. Association for Computing Machinery, 2013.

[142] Doug Lea. A Java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, pages 36–43, 2000.

[143] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The design of a task parallel library. *Acm Sigplan Notices*, 44(10):227–242, 2009.

[144] Jing Li, Son Dinh, Kevin Kieselbach, Kunal Agrawal, Christopher Gill, and Chenyang Lu. Randomized work stealing for large scale soft real-time systems. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 203–214. IEEE, 2016.

[145] Chi Liu, Ping Song, Yi Liu, and Qinfen Hao. Efficient work-stealing with blocking deques. In *2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC, CSS, ICESS)*, pages 149–152. IEEE, 2014.

[146] Nian Liu, Binyu Zang, and Haibo Chen. No Barrier in the Road: A Comprehensive Study and Optimization of ARM Barriers. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '20, pages 348–361, New York, NY, USA, 2020. Association for Computing Machinery.

[147] Jacob R Lorch, Yixuan Chen, Manos Kapritsos, Bryan Parno, Shaz Qadeer, Upamanyu Sharma, James R Wilcox, and Xueyuan Zhao. Armada: Low-effort verification of high-performance concurrent programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 197–210, 2020.

[148] Millán A Martínez, Basilio B Fraguela, and José C Cabaleiro. A parallel skeleton for divide-and-conquer unbalanced and deep problems. *International Journal of Parallel Programming*, 49(6):820–845, 2021.

[149] Sarah McClure, Amy Ousterhout, Scott Shenker, and Sylvia Ratnasamy. Efficient scheduling policies for microsecond-scale tasks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1–18, 2022.

[150] Paul E McKenney, Jonathan Appavoo, Andi Kleen, Orran Krieger, Rusty Russell, Dipankar Sarma, and Maneesh Soni. Read-copy update. In *AUUG Conference Proceedings*, volume 175. AUUG, Inc., 2001.

[151] Maged M Michael and Michael L Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275, 1996.

[152] Maged M Michael and Michael L Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275, 1996.

[153] Maged M Michael, Martin T Vechev, and Vijay A Saraswat. Idempotent work stealing. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 45–54, 2009.

[154] Gal Milman, Alex Kogan, Yossi Lev, Victor Luchangco, and Erez Petrank. Bq: A lock-free queue with batching. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, pages 99–109, 2018.

[155] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

[156] Gordon E Moore. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp. 114 ff. *IEEE solid-state circuits society newsletter*, 11(3):33–35, 2006.

[157] Adam Morrison and Yehuda Afek. Fast concurrent queues for x86 processors. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 103–112, New York, NY, USA, 2013. Association for Computing Machinery.

[158] Adam Morrison and Yehuda Afek. Fence-free work stealing on bounded TSO processors. In Rajeev Balasubramonian, Al Davis, and Sarita V. Adve, editors, *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2014, Salt Lake City, UT, USA, March 1-5, 2014*, pages 413–426. ACM, 2014.

[159] Hossein Naderibeni and Eric Ruppert. A wait-free queue with polylogarithmic step complexity. In *Proceedings of the 2023 ACM Symposium on Principles of Distributed Computing*, pages 124–134, 2023.

[160] Ruslan Nikolaev. A Scalable, Portable, and Memory-Efficient Lock-Free FIFO Queue . `https://github.com/rusnikola/lfqueue`.

[161] Ruslan Nikolaev. A scalable, portable, and memory-efficient lock-free fifo queue. In *33rd International Symposium on Distributed Computing (DISC 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[162] Ruslan Nikolaev and Binoy Ravindran. Wcq: A fast wait-free queue with bounded memory usage. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '22, page 461–462, New York, NY, USA, 2022. Association for Computing Machinery.

[163] Brian Norris and Brian Demsky. CDSchecker: checking concurrent data structures written with C/C++ atomics. In Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes, editors, *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 131–150. ACM, 2013.

[164] Brian Norris and Brian Demsky. CDSchecker: checking concurrent data structures written with C/C++ atomics. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pages 131–150, 2013.

[165] Jonas Oberhauser, Rafael Lourenco de Lima Chehab, Diogo Behrens, Ming Fu, Antonio Paolillo, Lilith Oberhauser, Koustubha Bhat, Yuzhong Wen, Haibo Chen, Jaeho Kim, and Viktor Vafeiadis. Vsync: Push-button verification and optimization for synchronization primitives on weak memory models. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, pages 530–545, New York, NY, USA, 2021. Association for Computing Machinery.

[166] Or Ostrovsky and Adam Morrison. Scaling concurrent queues by using htm to profit from failed atomic operations. In *Proceedings of the 25th*

ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 89–101, 2020.

[167] Kaiming Ouyang, Min Si, Atsushi Hori, Zizhong Chen, and Pavan Balaji. CAB-MPI: Exploring interprocess work-stealing towards balanced MPI communication. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2020.

[168] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-tso. In *Theorem Proving in Higher Order Logics: 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings 22*, pages 391–407. Springer, 2009.

[169] Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. Bridging the gap between programming languages and hardware weak memory models. *Proc. ACM Program. Lang.*, 3(POPL):69:1–69:31, 2019.

[170] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tuma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, et al. Renaissance: benchmarking suite for parallel applications on the JVM. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 31–47, 2019.

[171] Junjie Qian, Witawas Srisa-an, Du Li, Hong Jiang, Sharad Seth, and Yaodong Yang. Smartstealing: Analysis and optimization of work stealing in parallel garbage collection for Java VM. In *Proceedings of the Principles and Practices of Programming on The Java Platform*, pages 170–181. 2015.

[172] Florian Schmaus, Nicolas Pfeiffer, Timo Hönig, Jörg Nolte, and Wolfgang Schröder-Preikschat. Nowa: A wait-free continuation-stealing concurrency platform. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 360–371. IEEE, 2021.

[173] Hermann Schweizer, Maciej Besta, and Torsten Hoefler. Evaluating the cost of atomic operations on modern architectures. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 445–456. IEEE, 2015.

[174] Nicholas A Solter and Scott J Kleper. *Professional C++*. John Wiley & Sons, 2005.

[175] Markus Steinberger, Bernhard Kainz, Bernhard Kerbl, Stefan Hauswiesner, Michael Kenzel, and Dieter Schmalstieg. Softshell: dynamic scheduling on GPUs. *ACM Transactions on Graphics (TOG)*, 31(6):1–11, 2012.

[176] Warut Suksompong, Charles E Leiserson, and Tao B Schardl. On the efficiency of localized work stealing. *Information Processing Letters*, 116(2):100–106, 2016.

[177] Kun Suo, Jia Rao, Hong Jiang, and Witawas Srisa-an. Characterizing and optimizing hotspot parallel garbage collection on multicore systems. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–15, 2018.

[178] Boris Teabe, Vlad Nitu, Alain Tchana, and Daniel Hagimont. The lock holder and the lock waiter pre-emption problems: Nip them in the bud using informed spinlocks (i-spinlock). In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 286–297, 2017.

[179] Thomas N Theis and H-S Philip Wong. The end of moore's law: A new beginning for information technology. *Computing in science & engineering*, 19(2):41–50, 2017.

[180] Julio Toss. Work stealing inside GPUs. 2011.

[181] Stanley Tzeng, Anjul Patney, and John D Owens. Task management for irregular-parallel workloads on the GPU. 2010.

[182] Bill Venners. The java virtual machine. *Java and the Java virtual machine: definition, verification, validation*, 1998.

[183] Simon Friis Vindum, Dan Frumin, and Lars Birkedal. Mechanized verification of a fine-grained concurrent queue from meta's folly library. 2022.

[184] Jiawei Wang, Diogo Behrens, Ming Fu, Lilith Oberhauser, Jonas Oberhauser, Jitang Lei, Geng Chen, Hermann Härtig, and Haibo Chen. BBQ: A block-based bounded queue for exchanging data and profiling. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 249–262, 2022.

[185] Chaoran Yang and John Mellor-Crummey. A wait-free queue as fast as fetch-and-add. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–13, 2016.

[186] Jixiang Yang and Qingbi He. Scheduling parallel computations by work stealing: A survey. *International Journal of Parallel Programming*, 46(2):173–197, 2018.

[187] Yiwen Zhang, Gautam Kumar, Nandita Dukkipati, Xian Wu, Priyaranjan Jha, Mosharaf Chowdhury, and Amin Vahdat. Aequitas: admission control for performance-critical RPCs in datacenters. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 1–18, 2022.