UNIVERSITY OF BAMBERG

International Software Systems Science Degree Program in the
Faculty of Information Systems and Applied Computer Sciences

Masters's Thesis

Analysis of Novel Approaches for Work-Stealing Scheduling

BY

Shibesh Duwadi

Registration Number: 2129835

Supervisors:
Florian Schmaus, M.Sc.
Prof. Dr. Michael Engel

Practical Computer Science, esp. Systems Programming

May 2025

Links to this document:
https://github.com/Shibeshd/masters-thesis-report (most recent version)

Abstract

The inception of multicore processors has emphasized the importance of parallelism in enhancing computing performance. Work-stealing has emerged as the prevalent load-balancing strategy in task-parallel state-of-the-art runtimes due to its demonstrated efficiency. In work-stealing, idle processor cores, called thieves, dynamically "steal" tasks from busier workers, thereby maximizing processor core utilization. While classical work-stealing algorithms and the research surrounding them have primarily focused on workload distribution, they often overlook the impact of contention and synchronization overhead in highly concurrent environments. Despite their widespread adoption in state-of-the-art runtime systems, these classical queue implementations can introduce severe bottlenecks under specific operational conditions.

The recently proposed Block Based Work-Stealing (BWoS) architecture aims to address the limitations inherent in classical algorithms through several innovative performance-enhancing techniques. The fundamental principle of this approach involves partitioning a queue into multiple blocks with explicit ownership semantics—ensuring that only an owner or thief operates on a block at any given time—theoretically reducing interference and synchronization costs between worker and thief threads. In this thesis, we provide some background on the design architecture of a few classical lock-free work-stealing queue algorithms, compare their designs with BWoS and present a C++ implementation of both FIFO and LIFO variants of the BWoS queue based on design specifications. We note that the algorithm's heightened complexity posed significant implementation challenges due to inherent issues with the block-based design, particularly in concurrent execution. Finally, we conduct a comprehensive performance comparison between BWoS and other lock-free and locked queue implementation. Through controlled microbenchmarks and diverse real-world applications, our analysis reveals that work-stealing performance is highly context-dependent, with significant variations across task granularity and workload characteristics. These empirical findings emphasize the necessity of practical evaluation over purely theoretical analysis. Although BWoS demonstrates notable advantages in specific scenarios, our results indicate that it may not be universally optimal for general use cases and requires precise parameter tuning to achieve peak performance.

Acknowledgements

First and foremost, I would like to thank my supervisor, Florian Schmaus, for his exceptional guidance and support throughout the course of this thesis. His unending enthusiasm for explaining complex technical topics, no matter how intricate, has truly been a blessing. His course, *SYSNAP-Proj-M*, was a pivotal moment for me in selecting this thesis topic, sparking my interest in concurrency and parallel processing. I am especially thankful for his invaluable help with the implementation and his willingness to engage in in-depth discussions.

My sincere thanks also go to Prof. Dr. Michael Engel, the head of the Systems Programming chair, and my colleagues at the chair. The collaborative spirit, insightful discussions, and the positive and welcoming atmosphere they fostered have been an essential part of this experience.

I would like to thank my girlfriend, Pallavi, whose constant support and encouragement have meant the world to me. She has been my rock during difficult times, cheering me on when things didn't go as planned, and giving me the strength to push through countless sleepless nights. Her support and belief in me were instrumental in bringing this thesis to completion.

Last but not least, I am grateful to my parents for their unwavering support and encouragement throughout my academic journey. Their sacrifices have enabled me to pursue higher education far from home, in a country and culture very different from our own. Without their love and support, none of this would have been possible.

Contents

List of Figures

List of Tables

1 Introduction

The conception of multicore processors has made parallelism the cornerstone of performance enhancement in modern computing systems. As processor architecture evolves, the gap in performance between sequential and well-parallelized code continues to widen, a trend that is expected to persist in future generations [Sat+12]. This divergence is primarily due to the limited resource utilization of single-threaded applications, which leverage only a fraction of a processor's capabilities. In contrast, parallel programming models aim to exploit the full spectrum of computational resources, a goal that becomes increasingly critical as researchers aim to find the perfect balance between clock speeds and power consumption.

Achieving this goal necessitates a shift in programming paradigms i.e. from managing threads and locks to employing finer-grained units of execution such as tasks or fibers. While threads remain important, they are hard to use and are affected by *thread blocking anomaly* i.e. the loss of parallelism when threads execute a blocking system call [Sch+21a]. To lower the barrier to parallel programming, modern programming languages provide inbuilt constructs to express concurrency that, when combined with robust runtime systems, facilitate *dynamic task parallelism* through concurrency platforms. These platforms abstract the complexities of thread management, scheduling, and load balancing, thereby alleviating the programmer's burden by allowing them to focus on logical tasks which ultimately results in more robust applications. However, the effectiveness of such abstractions is inherently linked to the underlying implementation. Task-parallel runtime systems need to adeptly handle larger volumes of fine-grained tasks while keeping overhead low; failure to do so can negate the benefits of task-based parallelism, especially as core counts increase.

A prevalent strategy employed by modern concurrency platforms for load balancing is *work-stealing*. This approach typically relies on concurrent data structures like double-ended queues (deques), which facilitate access on both ends. In this paradigm, each core or kernel-level thread maintains its own worker queue, scheduling and executing tasks independently. Idle workers, upon depleting their local queues, assume the role of "thieves," attempting to steal tasks from the queues of busier workers to maintain balanced workloads. Synchronization mechanisms, such as exclusive locks or atomic operations in a lock-free manner, are employed to coordinate access, introducing overhead that becomes significant, particularly in systems with a high core count or when dealing with small, rapidly executable tasks, where synchronization costs can dominate execution time [Wei19]. Additionally, issues like false cache line sharing—where multiple processors inadvertently contend for the same cache lines—can degrade performance. The granularity of tasks plays a crucial role; overly fine-grained tasks may lead to excessive overhead, while coarse-grained tasks might result in suboptimal load balancing. Furthermore, implementing efficient work-stealing algorithms can be complex, especially when aiming to minimize contention

and ensure fairness among threads. These challenges underscore the importance of carefully designing and tuning work-stealing mechanisms to suit specific application requirements and system architecture.

Researchers have proposed various algorithms to tackle these challenges, with the aim to enhance performance and scalability. Real world runtimes for high performance computing rely heavily on work-stealing queues for scalability and performance [07; Flo25; FLR98; Sch+21b]. Moreover, managed languages like Java and Go utilize work-stealing for automatic memory management through garbage collection processes [Det+04; Hel23; Hor+18]. Hence, analyzing the many flavors of work-stealing scheduling is crucial because different implementations can significantly impact system efficiency, scalability and performance. Factors such as *task granularity*, *system architecture*, and *workload characteristics* influence the effectiveness of a particular strategy. This thesis aims to evaluate and compare several work-stealing algorithms against the aforementioned factors and identify their strengths and weaknesses. The goal is to inform optimized scheduling decisions tailored to specific applications and system configurations.

2 Background

Over the past several decades, companies like Intel and AMD have shifted their focus from increasing clock speeds to enhancing performance through multicore processors and simultaneous multithreading (SMT) [Sut05]. Hyperthreading was first introduced in Xeon server processors in February 2002 and later extended to consumer-oriented Pentium 4 desktop processors in November 2002 [X-b14]. This shift marked a fundamental turning point in software development, as most applications at the time were designed to utilize only a single processor thread, preventing them from fully leveraging the advancements in newer processor architectures.

As a result, to fully exploit the capabilities of modern multicore processors, programmers had to explicitly express concurrency in their code, enabling applications to efficiently utilize multiple processing threads and achieve significant performance gains. In modern day, the growing dependence on multi-core and many-core systems to meet the high-performance demands of complex applications is becoming increasingly evident. As the number of processing elements in computer architectures continues to rise, scalability via efficient load balancing has become a critical factor in ensuring efficient performance. Scalability is often achieved through fully-strict fork/join concurrency [Sch+21b], a commonly adopted model in popular concurrency platforms like Intel's *TBB* (Thread Building Blocks) [07] and *Cilk* [FLR98].

2.1 Concurrency and Parallelism

Concurrency and parallelism, while related, are distinct ideas. However, the distinction between these concepts is often overlooked, as both are commonly regarded simply as tools for boosting performance. **Concurrency** involves the execution of multiple threads whose lifetimes overlap in time, though not necessarily running simultaneously. For example, assume two threads T_A and T_B as shown in Figure 2.1. If thread T_A finishes executing before thread T_B begins, the threads are considered to have executed sequentially. However, if T_B starts running before T_A has completed executing, then thread T_A is suspended and processor resources are allocated to thread T_B via a context switch. At this stage, logically speaking, both the threads are running at the same time and neither has completed execution. Here, we can say that T_A and T_B happen concurrently.

Now if we take the same example and bind each thread to different cores of a processor, then we can execute both of those threads simultaneously without any context switching. This is known as **parallelism**. Parallelism, by definition, occurs when two or more independent threads execute at the same time. From this example, we can deduce that concurrency facilitates parallelism by describing the structure of the program. The same program can be concurrent when running on a single core and parallel when running on multiple cores.

While multiple threads are commonly used to achieve parallel execution, parallelism itself doesn't depend solely on threads. At the hardware level, different instructions can run simultaneously (instruction-level parallelism), and SIMD (Single Instruction, Multiple Data) instructions process several data items at once using vector operations (data parallelism) [Wik25b]. Since concurrency involves managing multiple tasks at once, parallelism can be viewed as a specific form of concurrency [SMR09]. To leverage parallelism, a program must demonstrate concurrency in some form. Therefore, concurrency is generally regarded as a broader concept than parallelism.

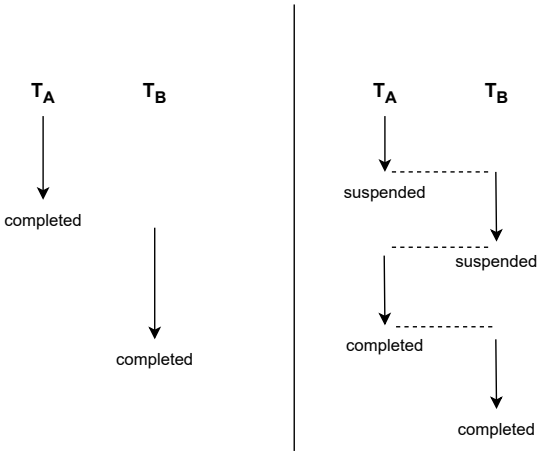

Figure 2.1: Sequential vs Concurrent control flow.

2.2 Fork-Join Parallelism and Concurrency Platform

"The fork–join model is a way of setting up and executing parallel programs, such that execution branches off in parallel at designated points in the program, to "join" (merge) at a subsequent point and resume sequential execution" [MRR12]. This model is also called the *task-based (parallel) programming model*. More simply, fork-join algorithms are parallel adaptations of traditional "divide-and-conquer" techniques that divide a bigger problem into independent parts or subtasks, solve the subtasks and join them to compose the final solution as shown in Figure 2.2.

A common approach to efficiently handling fine-grained tasks is to ***spawn*** multiple worker threads to execute them concurrently. A key aspect of this parallel model is that the programmer does not specify which tasks in a computation *must* run in parallel, only which tasks *may* run in parallel [Cor+09].

Figure 2.3 shows the implementation of a parallel program to calculate the n^{th} Fibonacci number. The `spawn` keyword indicates that the calling function (`fib(n)`), **may** run in parallel with the function that was called (`fib(n-1)`). Whereas the `sync` keyword indicates the point at which all spawned parallel tasks join or return. Modern programming languages have primitives to support such a parallel model as part of their language specification. In fork-join paradigms, it is primarily the runtime system that orchestrates the parallel execution of concurrent tasks. This combination of modern language with primitives to support parallelism, it's accompanying compiler and the runtime system together are referred to as a *concurrency platform*. When implemented on

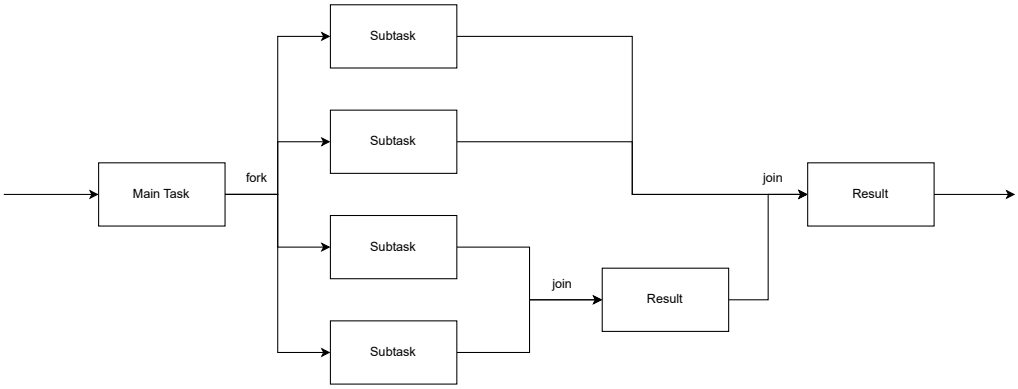

Figure 2.2: Illustration of the fork-join paradigm.

```
1  fib(n) {
2    if(n<2) return n;
3    x = spawn fib(n-1);
4    y = fib(n-2);
5     sync ;
6    return x+y;
7  }
8
```

Figure 2.3: Fibonacci function with concurrency keywords

a concurrency platform, the Fibonacci function would benefit from the cooperation between the language layer and the runtime system to effectively utilize computational resources. This synergy facilitates *dynamic task parallelism* which is a distinct characteristic of certain concurrency platforms [Cor+09; Sch+21b].

Most of the popular concurrency platforms, including Intel's TBB and Cilk Plus, have a lock-based synchronization mechanism within the platform's runtime-system whereas some newer runtime systems employ a wait-free approach to unlock additional performance on multi-core systems [Sch+21b]. In any case, all of them execute parallel programs with the fork-join model and work-stealing.

2.3 Work-Stealing and Work-Stealing Queue

Work-stealing is a decentralized scheduling mechanism architected for a rigorous fork-join parallel computation paradigm, It has demonstrated high efficiency in scheduling parallel tasks across multi-core architectures and has attracted considerable research attention over the years. The concept of work-stealing can be traced back to Halstead's implementation of *Lisp* for multiprocessor systems (Multilisp) [Hal84], as well as the work of Burton and Sleep on concurrent functional programming [BS81] in the 1980s. However, the work done by Blumofe and Leiserson was the first instance of a theoretical analysis of work-stealing. The scheduler that they implemented executes

a fully strict multithreaded computation on P processors within an expected time of

$$\frac{T_1}{P} + O(T_\infty) \tag{2.1}$$

, where T_1 is the minimum serial execution time of the multithreaded computation (the work of the computation) and T_∞ is the minimum execution time with an infinite number of processors (the span of the computation) [BL99].

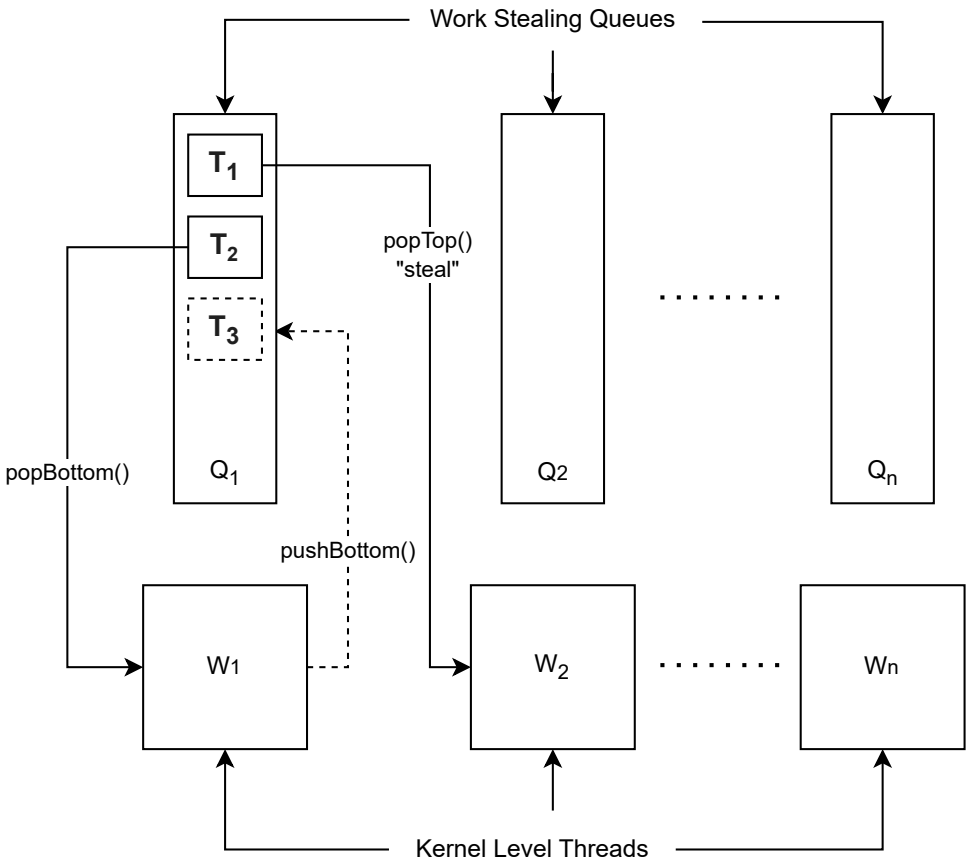

Figure 2.4: Kernel Level Threads and their Work-Stealing Queues.

The basic principle of work-stealing (see Figure 2.4) is that each kernel level thread (worker) manages a local queue of tasks that are runnable. These queues are double-ended queues (deques), with one end designated as the *bottom* and the other end as the *top*. The worker is referred to as the *owner* of the queue. Newly spawned tasks are typically enqueued (*pushBottom()*) to the *bottom* of this local deque by its owner. During execution, owner dequeues (*popBottom()*) tasks from the bottom of their own deque in a last-in, first-out (LIFO) manner. This strategy benefits from cache locality, as recently added tasks are more likely to have relevant data still in the cache [DB82]. When a worker's deque becomes empty, indicating that it has no tasks left to execute it becomes a *thief*. To avoid idling, the thief selects another worker (the *victim*) and attempts to steal (*popTop()*) a task

from the *top* of the victim's queue. As a result, as long as the queue contains multiple tasks, only one worker accesses each end at a time. Also, while it is possible to remove items from both ends, it is only possible to add an item from the bottom of the queue. If a steal is successful, the task is executed. If a steal fails, it can mean one of two things — either the victim's queue is empty, or the victim has managed to execute the task before the *stealer* could finish the steal operation. In both cases, the thief will select another victim and retries the steal operation.

"All queue operations must be partially multithread-safe. That is, *popTop()* can be used concurrently with itself and at most one of the two bottom operations on the same queue instance. However, since *pushBottom()* and *popBottom()* are only ever used by the same worker, work-stealing queues do not need to support concurrent bottom operations" [Sch+21b]. Table 2.1 gives a tabular overview of these properties which work-stealing algorithms exploit. Further information about memory consistencies in concurrent data structures are discussed in subsection 2.3.2.

	pushBottom()	**popBottom()**	**popTop()**
pushBottom()	Conflict	Conflict	Allowed
popBottom()	Conflict	Conflict	Allowed
popTop()	Allowed	Allowed	Allowed

Table 2.1: Concurrent operation interactions

By allowing idle processors to proactively seek out work, work-stealing provides a decentralized and efficient method for dynamic load balancing. This technique is particularly effective in scenarios where the workload is unpredictable or irregular, as it adapts in real-time to the system's state, thereby enhancing parallel processing efficiency. Although the practical benefits of work-stealing have been widely recognized and studied for decades, researchers continue to develop novel variants and extensions of the classic work-stealing paradigm to further enhance throughput and performance.

2.3.1 Garbage Collection

Garbage collection (GC) is an automatic memory management technique whereby the system reclaims memory previously allocated by a program but no longer in use. Such unused memory is aptly referred to as *garbage*. GC's inception can be dated back to 1960 with the first ever implementation of automated memory management in Lisp [McC60]. In modern programming languages, garbage collection is either mandated as part of the language specification (e.g., Java, .NET, Go, and most scripting languages) or is essential for practical implementation (e.g., formal languages like lambda calculus) [Hel23].

In managed languages like Java, the GC frees developers from manual deallocation and prevents common errors such as memory leaks. By scanning the heap to identify unreachable objects and freeing their memory, the GC "reclaims objects that are no longer being used, clears their memory, and keeps the memory available for future allocations" [Mic23]. This automatic reclamation is crucial for performance and reliability, as it prevents memory exhaustion and dangling references that would otherwise degrade program execution.

2.3.1.1 Parallel Garbage Collection using Work-Stealing

To scale on modern multi-core hardware, many collectors employ parallel algorithms. For example, HotSpot's JVM (Java Virtual Machine) includes multithreaded collectors (Parallel Scavenge, CMS, G1, etc.) that divide the heap into regions or object sets so that multiple GC threads can work concurrently [Wik25a]. Each GC thread may perform tasks like marking live objects, copying survivors, or updating pointers in parallel. Effective scheduling of these tasks is required because an unbalanced workload can leave some threads idle while others still work. In practice, managed runtimes like the JVM relies on GC to maintain performance on multi-core systems [Has16], so dynamic load balancing across GC threads is critical. Typically, each GC thread draws tasks (such as "process this memory region") from a shared or private queue until all work for the GC phase is complete.

Work-stealing is typically used to manage work among threads in parallel GC [Hor+18]. By allowing idle threads to take work from busy ones, work-stealing dynamically redistributes remaining tasks during stop-the-world GC phases and improves load balance. Heap regions or object sets are used as the payload or work items that are pushed, popped and stolen. OpenJDK's HotSpot VM provides a concrete implementation of GC work-stealing using specialized task queues. The HotSpot source code defines a class `GenericTaskQueue` for GC threads. According to the information provided in the open source Github repository for OpenJDK, `GenericTaskQueue` implements an *ABP (Arora–Blumofe–Plaxton)* double-ended-queue [ABP98; Ope25]. In practice, when a GC thread (the owner) has work to do it pushes tasks (for example, "scan this memory region" or "copy these objects") onto its deque. If another GC thread becomes idle, it will call `pop_global()` on a victim thread's queue to steal one of those tasks. Importantly, `GenericTaskQueue` is designed to be lock-free and allows wrap-around of its fixed-size array, so operations like push, pop, and steal can proceed with minimal synchronization. Aside from the wrap-around optimization, this implementation closely follows the classic ABP work-stealing deque semantics of Arora *et al.* as discussed in subsubsection 2.4.3.1.

2.3.2 Memory Consistency and Concurrent Data Structures

Memory consistency defines the order in which memory operations (loads/stores) appear to execute in a multiprocessor system. The strongest model is **Sequential Consistency**: all processors' memory operations appear as if linked in a single global sequence that respects each thread's program order [Mos93]. Sequential consistency is easy for programmers to use, but it forbids hardware optimizations like write buffering or reordering, which are critical components of concurrency which prefer weaker memory models. While multicore processors use various kinds of consistency models on the hardware level, we mainly focus on memory barriers and synchronization on the software level when it comes to work-stealing algorithms.

Synchronization primitives provide ordering in software. For example, C/C++11 atomics and fences implement *acquire–release* semantics: a load with *memory_order_acquire* prevents any later memory operations in that thread from moving before the load, and a store with *memory_order_release* prevents earlier operations from moving after the store. An atomic read-modify-write operation (compare-and-swap or CAS) with *memory_order_acq_rel* combines both these semantics [cpp24]. Unlike traditional locking primitives that explicitly serialize access, CAS does not impose serialization by itself. However, when multiple threads contend on the same memory location, CAS operations effectively serialize due to repeated retries and the need for exclusive cache line ownership. Before a CAS can complete, it must wait for all prior memory operations

to finish and for the relevant cache line to be available and owned exclusively by the thread. This blocks the CPU pipeline and introduces latency, especially on contention-heavy workloads such as in a work-stealing algorithm. If CAS fails in a queue operation, it usually retries in a spin loop which adds additional overhead. While a fully sequential fence (*memory_order_seq_cst*) can give guarantees of the order of operations, in practice, they are costly: enforcing a global total order requires expensive CPU instructions that drains store buffers and stalls execution. On the opposite end, a relaxed memory order *memory_order_relaxed* allows most reordering and only guarantees the atomicity of the load/store.

When we are dealing with work-stealing deques, queue operations require a combination of the semantics discussed so far. Sequential ordering leads to heavily bottlenecked performance, and without fences or atomic operations, the following invariants can be violated by processor reordering:

- **Push (owner)** - The owner thread writes a new task into the array before incrementing the bottom index. If these become visible out of order, a thief might see an increased bottom but not the task data. Thus, a memory barrier must be used so that the task store is globally visible before bottom is updated [Lê+13].
- **Pop (owner)** - When popping, the owner first decrements `bottom` and then checks `top`. These operations must not be observed by other threads in the wrong order. In a circular CL (Chase-Lev) deque, simply using a release-store to update bottom and an acquire-load to read top is not sufficient to order the store before the load [CL05]. A release/acquire fencing does not enforce an order between the store and the load themselves. Hence, the pop operation often uses an *atomic compare and exchange* or *atomic_fetch_sub* to order the tail update with respect to the head load.
- **Steal (thief)** - A thief reads the current `bottom` index and then the `top` index, then tries a CAS on top to claim an item. If both operations race with a concurrent pop, one of them must fail safely. In the single-element case, Chase and Lev's algorithm ensures correctness by having both the pop and the steal use a CAS on the same variable. The correctness proof relies on the fact that at least one of the threads will have a consistent view of `bottom` and `top` – guaranteed by placing full barriers in both routines. Placing two barriers—one in the pop and one in the steal—ensures that at least one thread sees the updated deque size. If the pop sees the size, it will perform the CAS; if the steal sees the size, its index reservation leads to an empty return [Lê+13].

2.4 Motivation for Novel Approaches

Although work-stealing is simple in description (section 2.3), the queue is the heart of the paradigm. Hence, analyzing these approaches requires us to understand the most basic form of the queue first.

2.4.1 Global Queue Scheduling

Let us consider the following scenario — all processor cores (or kernel level threads) share a single, centralized queue for task scheduling. We can call this a "Global Queue". In this setup, tasks are enqueued into a global queue and idle processors dequeue tasks for execution. After completing a task, processors return to the global queue to dequeue the next available task. Naturally, only

the *pushBottom()* and *popBottom()* operations are utilized in such a scenario which means that the top of the queue is never used, and *"steal"* never occurs. This approach greatly simplifies task distribution in multicore systems and thus the implementation is quite straightforward. Global queue schedulers perform well under two conditions: (a) when the system has a small number of cores, regardless of the number of tasks, and (b) when the system has a large number of cores but the tasks are uniform and predictable (common in GPU scheduling scenarios) [KS14]. However, in general-purpose multicore systems, where task execution times and workloads are irregular, global queue scheduling introduces several key limitations:

L1: Scalability Issues. As the number of processors increases, contention for the global queue intensifies. Multiple processors attempting to access the queue simultaneously can lead to bottlenecks, reducing overall system throughput.

L2: Serialization of Access. Synchronization mechanisms enforce a sequential order of access to the queue, effectively serializing operations that could otherwise be performed in parallel. This serialization limits the degree of concurrency achievable in the system.

L3: Priority Inversion. Implementing priority scheduling within a global queue is highly problematic. High-priority tasks may be delayed if the queue is congested with lower-priority tasks, leading to suboptimal scheduling outcomes.

A natural evolution of this model is to decentralize task queues. More clearly speaking, instead of having a single global queue for task scheduling, each core maintains its own local queue with shared memory access. To maintain data consistency during concurrent access, we can use mutual exclusion mechanisms such as locks. This forms the basis of distributed scheduling models, which are better suited for dynamic and irregular workloads.

2.4.2 Locked Work-Stealing Queue

A straightforward approach to distributing task scheduling among multiple cores is to assign each core its own task queue. Instead of accessing a centralized global queue, a core enqueues tasks to its own local queue and dequeues tasks from it when idle. However, to support load balancing and prevent cores from idling unnecessarily, work-stealing is introduced — a core can "steal" tasks from the top of another core's queue when its own queue is empty. The *LockedQueue* implementation is a simple realization of this idea. Each queue is protected by a `mutex`, ensuring safe concurrent access in shared-memory systems. Tasks are enqueued and dequeued using the classic queue operations as discussed in section 2.3. By separating the access points — bottom for the owner and top for stealers — this design supports a basic form of producer-consumer paradigm.

However, since every queue operation requires a mutex lock as shown in Figure 2.5, no two core or kernel level threads can perform operations concurrently, even if those operations are at the opposite ends of the queue, i.e. top and bottom. So, while we do achieve a modicum of load balancing when compared with a global queue, this design introduces the following drawbacks:

L1: Lock Contention. Every operation on the queue acquires a mutex lock, which severely limits scalability. Under high contention — especially during frequent stealing — mutex acquisition can become a performance bottleneck.

L2: Poor Stealing Throughput. Frequent locking during *popTop()* leads to reduced throughput for stealing threads, making this approach inefficient for fine-grained tasks with short lifetimes.

```
1  LockedQueue<E, CAPACITY>::pushBottom(E item) ->
2    lock_guard<mutex> queue_mutex;
3    if(deque == CAPACITY)
4      return false;
5    deque.push_back(item);
6    return true;
7
8  LockedQueue<E, CAPACITY>::popBottom(E *item) ->
9    lock_guard<mutex> queue_mutex;
10   if(deque.empty())
11     return false;
12   *item = deque.back():
13   deque.pop_back();
14   return true;
15
16 LockedQueue<E, CAPACITY>::popTop(E *item) ->
17   lock_guard<mutex> queue_mutex;
18   if(deque.empty())
19     return Empty;
20   *item = deque.front();
21   deque.pop_front();
22   return Stolen;
23
```

Figure 2.5: Pseudocode for Locked Queue

L3: Non-determinism in Performance. Since locks can lead to unpredictable wait times, real-time guarantees and consistent latency are hard to achieve with this approach.

Nevertheless, the *Locked Queue* serves as a critical stepping stone. It validates the fundamental idea of distributed queues and illustrates how work-stealing could operate in a shared memory model. From this baseline, more advanced, lock-free, or cache-aware designs evolve to address the limitations listed above.

2.4.3 Lock-Free Work-Stealing Queue

2.4.3.1 ABP Queue

In 2001, Arora, Blumofe, and Plaxton presented the first instance of a non-blocking work-stealing algorithm, henceforth referred to as the *ABP* queue, with their paper "Thread Scheduling for Multiprogrammed Multiprocessors" [ABP98]. It quickly became the multi-processor load balancing technology of choice in both academia and industry. In fact, subsequent research in the field has continued to build upon the ABP algorithm to address its limitations. Like the locked queue, each owner uses its deque as a LIFO stack for local tasks, and idle threads steal tasks from the opposite end. However, unlike the locked queue which uses a mutex based lock to serialize access to the deque to accommodate many stealers, the ABP queue removes locks altogether by using atomic CAS instructions to synchronize concurrent *popTop()* queue operations for stealers [ABP98; CL05; Sch+21b].

In the low-level implementation of ABP, each deque is a fixed-size array indexed by two 32-bit counters. Even though a 32-bit unsigned integer can only represent values up to $2^{32} - 1 = 4,294,967,295$ which can be reached by modern 64-bit multi-core processors, the counters would not overflow because they are only ever updated to the size of CAPACITY. As shown in Figure 2.6, the `bottom` counter is updated only by the owner thread (incremented or decremented) and never by thieves while the `top` counter is managed by an atomic structure: Arora *et al.* introduce a custom

```
1  Struct Age {
2    int top;
3    int tag;
4  };
5  atomic<uintptr_t> bottom;
6  atomic<Age> top;
7
8  ABP<E, CAPACITY>::pushBottom(E item) ->
9    if(bottom ≥ CAPACITY)
10     return false;
11   queue[bottom] = item;
12   bottom++;
13   return true;
14
15 ABP<E, CAPACITY>::popBottom(E *item) ->
16   if(bottom == 0)
17     return false;
18   --bottom;
19   *item = queue[bottom];
20   Age oldAge = top;
21   if(bottom > oldAge.top + 1)
22     return true;
23   bottom = 0;
24   Age newAge;
25   newAge.top = 0;
26   newAge.tag = oldAge.tag + 1;
27   if(top.CAS(oldAge, newAge))
28     return true;
29   top = newTop;
30   return false;
31
32 ABP<E, CAPACITY>::popTop(E *item) ->
33   Age oldAge = top;
34   if(bottom ≤ oldAge.top)
35     return Empty;
36   *item = queue[oldAge.top];
37   Age newAge = oldAge;
38   ++newAge.top;
39   if(top.CAS(oldAge, newAge))
40     return Stolen;
41   return LostRace;
42
```

Figure 2.6: Pseudocode for ABP Queue

Age struct containing two variables, `top` and `tag` i.e. the current top index and a version tag (lines: 1 – 6), and by design it fits in a single machine word so that it can be updated atomically with a single CAS instruction, i.e. they are 16-bits each [ABP98]. Thus, a thief executes *popTop()* by checking if the queue is not empty (line 34), reading the task at the `top` index in *Age*, and attempting to increase the `top` counter with a CAS. The `bottom` counter is updated non-atomically as it is uncontested in a concurrent setting.

The key innovation is the version `tag` in *Age* to prevent the classic ABA problem. Without a tag, a thief might read the top index, get preempted while the owner empties and refills the deque (so that the top index happens to return to its old value), and then resume – its CAS would incorrectly succeed because `top` looks unchanged. To avoid this, the ABP design increments the `tag` whenever the deque's top is reset. In practice this means each (top, tag) pair is unique: if a thief is delayed and the queue is emptied and filled to the same top position again, the `tag` will have changed, so the thief's CAS will fail rather than "see" a stale value.

The ABP queue suffers from a pathological case where its queue capacity may decrease dynamically. As we can see from the pseudocode for the ABP queue, *pushBottom()* and *popTop()* only increment

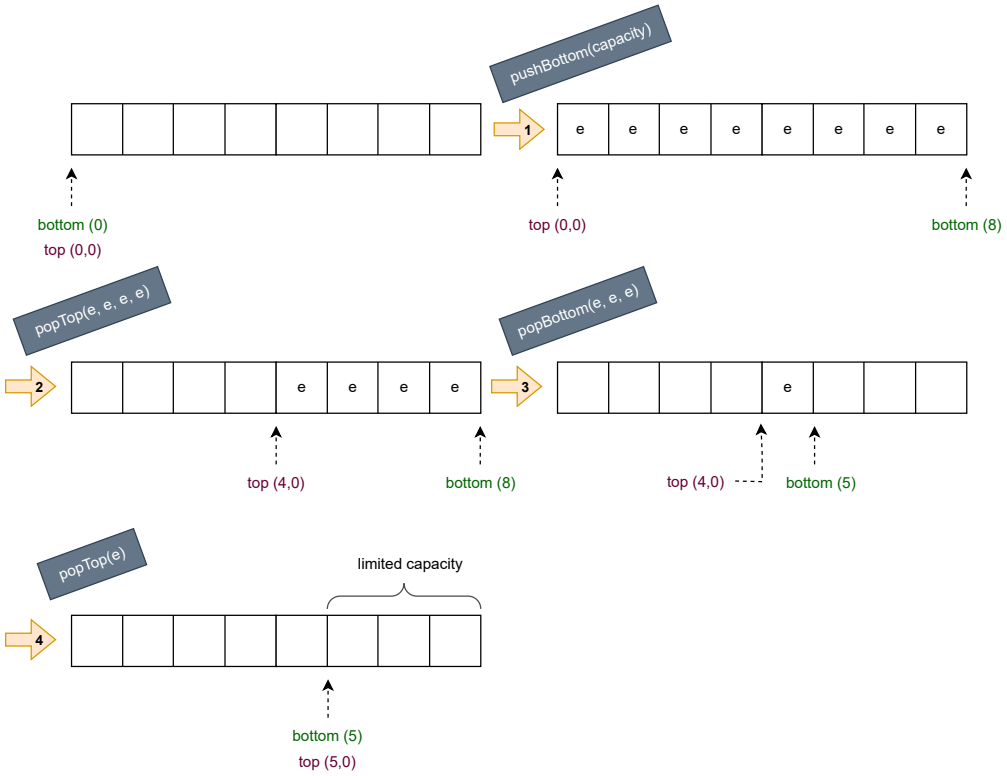

Figure 2.7: Pathological issue with the ABP queue

counters and never decrement them. Furthermore, due to the use of fixed-size non-circular array, there is never the possibility to wrap around to the same index. As a result, the space released by *popTop()* cannot be reused for inserting new elements [Sch+21b]. The design mitigates this issue by somewhat, when the owner removes the last item in the queue (line 21), it will reset the `bottom` index and atomically reset the `top` index and increases the `tag` version with a CAS (line 27). However, the reduced capacity would persist until this reset is made in the owner's *popBottom()*. This is illustrated in Figure 2.7 where an ABP queue with a capacity of 8 elements is shown. After some queue operations the queue reaches the state as shown in step (4) where the `bottom` index is 5 and the queue has limited capacity (3 elements) until the *popBottom()* procedure resets the index to 0.

2.4.3.2 CL Queue

David Chase and Yossi Lev introduced their version of a work-stealing algorithm, based on the work of Arora, Blumofe and Plaxton, with their paper "Dynamic Circular Work-Stealing Deque" in 2005 [CL05]. Their algorithm, henceforth referred to as the *CL* Queue, was based around cyclic arrays that can dynamically grow when it overflows and the use of two 64-bit unsigned integer counters `top` and `bottom`. These 2 key changes circumvent the issues of the ABP queue.

Firstly, due to the cyclic nature of the queue, it does not require the reset-on-empty mechanism used in the original ABP algorithm [CL05]. Analyzing the pseudocode for the CL queue, we can observe that, if a *pushBottom* operation discovers that the current circular array is full (line 7),

```
atomic<uint64_t> bottom;
atomic<uint64_t> top;

CL<E, CAPACITY>::pushBottom(E item) ->
  if(bottom - top > CAPACITY)
    return false;
  queue[bottom % CAPACITY] = item;
  bottom++;
  return true;

CL<E, CAPACITY>::popBottom(E *item) ->
  localBottom = --bottom;
  localTop = top;
  if(localTop > localBottom)
    bottom = localTop;
    return false;
  *item = queue[localBottom % CAPACITY];
  if (localBottom > localTop)
    return true;
  if(top.CAS(localTop, localTop + 1))
    bottom = localBottom + 1;
    return true;
  return false;

CL<E, CAPACITY>::popTop(E *item) ->
  oldTop = top;
  if(bottom ≤ oldTop)
    return Empty;
  *item = queue[oldTop % CAPACITY];
  newTop = oldTop + 1;
  if(top.CAS(oldTop, newTop))
    return Stolen;
  return LostRace;

```

Figure 2.8: Pseudocode for CL Queue

the deque elements stored in the circular array are indexed modulo its capacity, and therefore effectively wrap around the cyclic array. This modulo-based circular indexing allows the `bottom` and `top` counters to increase indefinitely without being reset or wrapped. Even as the absolute counter values grow beyond the physical array size, the logical positions of elements remain consistent, determined by the relative difference between the counters. Since the only operation that modifies `top` is steal, it is never decremented, and there is no need for a `tag` field as in the original ABP implementation.

Second, since the counters are 64-bit unsigned integer, they can have a max value of $2^{64} - 1$ which is large enough to accommodate a conservative 64 years of enqueues, dequeues and steals executing with a similar throughput of $4x10^9$ ops per second. This means that even though the counters increase indefinitely, they are highly unlikely to reach maximum capacity even with big workloads in the most efficient of concurrent environments.

2.4.4 Motivation for BWoS

As is evident by the discussion on work-stealing above, stealing incurs a high overhead due to contention. This overhead is further multiplied if the workload consists of smaller tasks. In 2023, Wang *et al.* introduced a novel concept of a work-stealing algorithm, especially tailored to improving performance for fine-grained tasks in their paper "BWoS: Formally Verified Block-based Work Stealing For Parallel Processing" [Wan+23]. The authors note the main sources of inefficiency in a classical work-stealing algorithm such as:

L1: Synchronization Overhead. To accommodate potential task stealing, local queues must employ strong atomic operations, such as CAS and memory barriers. These operations introduce significant synchronization costs, especially when managing fine-grained tasks. Wang *et al.* observe that when the throughput of algorithms like the ABP Queue, goroutineq [The25], tokioq [Tok25a] is measured on a sequential setup (without steals) and compared with a single threaded last-in, first-out (LIFO) and first-in, first-out (FIFO) queue implementation, due to synchronization operations on these algorithms, the throughput measured is less than $0.25x$ for FIFO-based queue and $0.19x$ for LIFO-based queue compared to the upper bound [Wan+23].

L2: Memory Access Time. Traditional work-stealing assumes uniform memory access time across cores and hence the transfer time between these cores is also same. However, in practice, when working with clusters of multicore systems and Non-Uniform Memory Access (NUMA) architectures, the memory access time of all cores are different. As classical work-stealing utilizes pure random stealing, this can interfere with locality optimizations [KS14].

L3: Cache Coherency. Concurrent access to shared queues by multiple cores can lead to increased cache coherence traffic. This traffic results from the need to maintain consistency across processor caches, causing delays and reducing overall system efficiency.

L4: Victim Selection. Victim selection is perhaps one of the most researched sub-topics of this domain. Several stealing policies have been proposed to determine the most suitable queue (*victim*) from which to steal tasks [Hor+19; KS14; Kum20; Liu+14; Mit01; SLS16]. The choice of policy can significantly impact performance, leading to substantial speedups depending on the specific use case and workload characteristics. To enhance workload distribution, sophisticated victim selection policies often involve scanning the metadata of multiple queues—for example, to identify the longest queue or one within the same NUMA domain. However, this approach introduces contention with the queue owners and significantly undermines the benefits of using such advanced selection strategies [Wan+23].

L5: Correctness under Weak Memory Model (WMM). Additional memory barriers are required when working with weak memory architectures. Redundant barriers diminishes the performance of work-stealing, while a lack of barriers often lead to errors, such as ABA, related to atomic operations.

Thus a new algorithm, henceforth referred to as BWoS, was introduced to provide practical solutions to these inefficiencies. The following section describes the underlying design of BWoS in detail, outlining key innovations and discussing its advantages. Throughout this section, we will see how the design differs from existing work-stealing approaches and recap the motivation of this approach.

3 Design

3.1 Architecture of the BWoS Queue

BWoS relies on a simple underlying idea: the work-stealing queue is partitioned into several blocks, each with explicit ownership semantics. At any given time, a block is exclusively owned either by the *owner* or by the *thieves*, creating a granular ownership model with the primary aim of reducing contention.

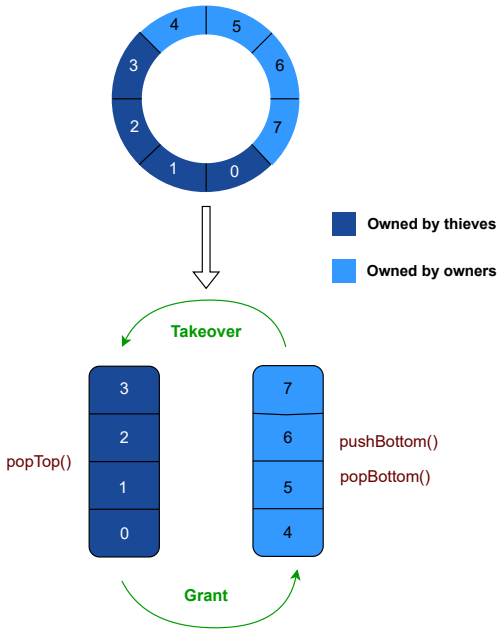

Figure 3.1: Bird's-Eye View of the BWoS queue

For the purposes of this report, the ownership semantics are represented with a color scheme where **light blue** (LB) blocks represent blocks owned by the owner and **dark blue** (DB) blocks represent blocks owned by the thieves as shown in Figure 3.1. To understand this block-based approach, consider the *pushBottom()*, *popBottom()* and the *popTop()* operations that are typical on a work-stealing algorithm as discussed in section 2.3. Only *pushBottom()* and *popBottom()* can be

performed on light-blue blocks whereas *popTop()* can be performed on dark-blue blocks. Essentially, the blocks in a BWoS queue can be visualized as 2 separate queues themselves.

3.1.1 Block Metadata

In a typical work-stealing queue, there are global metadata shared between the owner and the thieves which allows for synchronization. Generally, there is a top index and a bottom index that define the boundary of valid data at their respective ends in the deque. The values of these counters are modified with every successful queue operation. In a non-blocking queue implementation, atomic compare-and-swap (CAS) operations are used to synchronize concurrent queue operations in a lock-free manner.

In the BWoS queue, the global metadata is replaced with per-block metadata instances. Since owners and thieves generally operate on different blocks, they update different metadata and do not interfere with each other for contention. This allows for synchronization at the block level [Wan+23] on concurrent operations. To support these operations, 4 metadata variables are used: **`b_pos`** (committed) which indicates the back position, **`f_pos`** (consumed) which indicates the front position, **`s_pos`** (reserved) which is the stealing position, and **`s_cnt`** (stolen) which is a counter of finished steals in the block. Each queue operation updates a combination of these variables to make the block based approach viable.

3.1.2 Overview of the Queue

Due to the block-level metadata instances, each operation in the queue can be split into the *fastpath* i.e. operations inside a single block without synchronization with thieves and *block advancement* i.e. operations across different blocks with synchronization. The block advancement procedures for *pushBottom()*, *popBottom()* and *popTop()* are henceforth referred to as **`adv_blk`**$_{put}$, **`adv_blk`**$_{get}$ and **`adv_blk`**$_{steal}$ respectively. The fastpath is extremely lightweight because all synchronization efforts are delegated to the block advancement procedure. This strategy enables the fastpath to approach the theoretical upper bound (near single-threaded performance) that was observed by the BWoS authors [Wan+23]. In the implementation section of this report, Table 3.3 and Table 3.4 give an overview of the metadata updates that occur in the fastpath and block advancement each operation for FIFO and LIFO BWoS queues respectively.

There are two variants of the BWoS queue — LIFO BWoS and FIFO BWoS. These differ in not just the order of enqueue and dequeue operations, but also in the way that block level metadata values are initialized and updated during operation. Perhaps the most important distinction between the two variants is how the block to work on is selected for each operation. Considering a circular queue (ring buffer) design as shown in Figure 3.2, each queue has 3 unique pointers that point to specific blocks in the queue in which the three queue operations can be performed. These pointers are aptly named and henceforth referred to as **`blk_to_put`**, **`blk_to_get`** and **`blk_to_steal`**. The former two are owner-exclusive and are unavailable to the thieves. In FIFO BWoS, block-local metadata allows stealing from the middle of the queue which bypasses the restrictions of a traditional FIFO single producer multiple consumer queue of always stealing the oldest task.

During initialization of the queue, in LIFO BWoS, the owner-specific `blk_to_get` and `blk_to_put` point to the bottom block in the queue whereas `blk_to_steal` points to the top block in the queue. During block advancement, the owner-exclusive pointers are incremented so that they always point to the same block, no matter the current state of the queue. Whereas in FIFO BWoS, `blk_to_get`

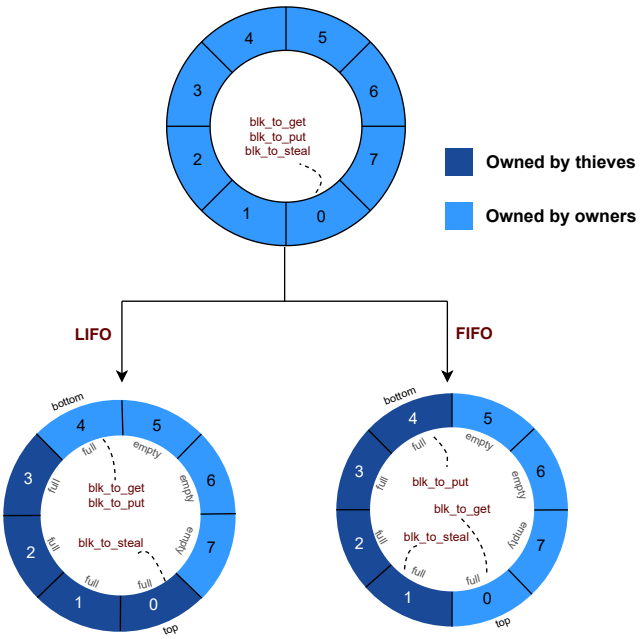

Figure 3.2: Variants of BWoS

points to the top block and `blk_to_put` points to the bottom block in the queue. Since it is already established that only DB blocks can be stolen from, `blk_to_steal` cannot point to either the top block or the bottom block, but instead it is initialized to the following block from the top of the queue and can advance to either of the DB blocks as shown in Figure 3.2.

Despite the advantages that the block-level metadata instances provide, the lack of global metadata also means that certain block-level invariants need to be strictly followed by the block advancement in order to ensure proper synchronization during concurrent operations. These invariants can be summarized as:

- *popBottom()* and *popTop()* never read the same data — they happen on different blocks,
- *pushBottom()* never overwrites unread data,
- *popBottom()* and *popTop()* never read data that has already been read,
- *popBottom()* can safely takeover a block from theives without waiting on the result of *popTop()*

To ensure that these invariants hold during concurrent workloads, the following 2 concepts are used: *block-level synchronization* and *round control*. The following provides an overview of how these procedures work. The technical architecture of these concepts are discussed in detail in Chapter 4.

Block-level Synchronization - This is the main paradigm that facilitates block advancement and ensures the aforementioned block-level invariants are not violated. As discussed earlier, light-blue (LB) blocks belong to the owner and dark-blue (DB) blocks belong to the thief. The owner can *grant* a block to the thieves (LB to DB), or *takeover* a block back from the thieves (DB to LB) with block

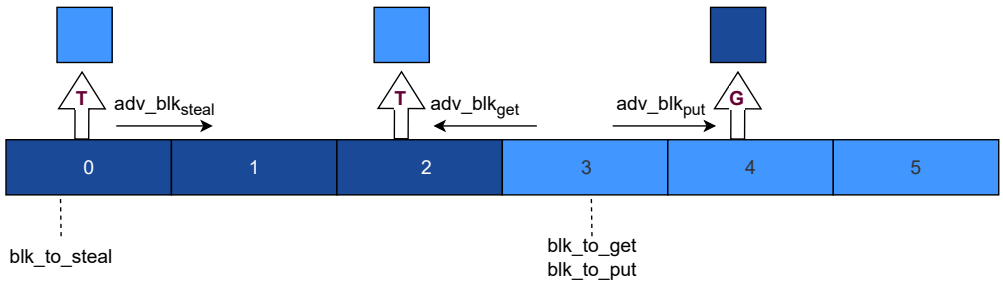

Figure 3.3: Block Advancement in LIFO BWoS. The queue is performing a takeover (**T**) procedure on block 0 and block 2 (DB to LB) and a grant (**G**) procedure on block 4 (LB to DB)

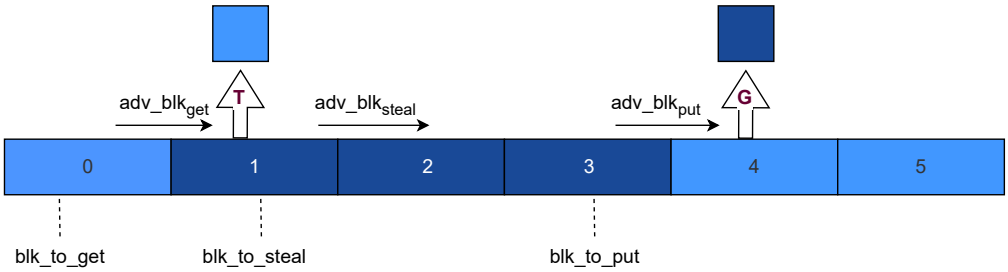

Figure 3.4: Block Advancement in FIFO BWoS. The queue is performing a takeover (**T**) procedure on block 1 (DB to LB) and a grant (**G**) procedure on block 4 (LB to DB)

advancement. Figure 3.3 and Figure 3.4 show how this process differs between the two variants of the queue highlighting the fact that **`adv_blk`**$_{get}$ in LIFO BWoS advances to the preceding block whereas in FIFO BWoS, it advances to the following block.

The value of the metadata variable `s_pos` (reserved) is what enables the grant and takeover procedures. This variable keeps track of where stealing has started inside a block. If the value of this variable is equal to the block's margin, that indicates that all items have been reserved and that no stealers are currently working on the block, and owners can take ownership of the block. In the takeover procedure, the owner atomically sets `s_pos` to the block's margin and uses the previous value as a cutoff point—this separates the portion still being accessed by thieves from the rest of the block. This way, the owner avoids interfering with any ongoing steals. Because the cutoff is updated atomically, the owner and thieves are guaranteed access to different parts of the block, avoiding conflicts. Similarly, in the grant procedure, the owner hands off part of the block to thieves by writing the cutoff point to `s_pos`, clearly marking which part of the block is available for stealing.

Round Control - For FIFO BWoS, each block also maintains a version number which represents the order of the last data access in the block. During block advancement for *pushBottom()* operations, the current block's metadata version of variables `b_pos` (committed), `s_pos` (reserved) and `s_cnt` (stolen) is copied to the next block's respective metadata. Whereas during block advancement for *popBottom()*, the current block's metadata version of variable `f_pos` (consumed) is copied to the next block's respective metadata. Considering a circular queue (ring buffer) design, when there is a wrap around in either of the above operations, the version number is increased by 1. This design ensures that unread data is never overwritten and prevents data from being read twice.

For LIFO BWoS, there is no need for metadata variables to maintain version numbers as the two owner operations *pushBottom* and *popBottom* never need to synchronize because the `blk_to_put` and `blk_to_get` pointers always point to the same block in the ring buffer. So, only synchronization between the owner and thieves is required which can be done with block level synchronization as described above.

3.2 Inherent Issue with FIFO BWoS Queue

Due to the algorithmic design of the FIFO BWoS queue, there is a common scenario that occurs during concurrent execution where the queue goes into a deadlock (stuck) state. In this state neither *pushBottom()*, *popBottom()* nor *popTop()* operations are permitted in the queue. In the author's own words, this is a completely sane behavior where the queue is "empty" and "full" at the same time [Tok25b].[1] This pathological case occurs mainly due to how the *round control* procedure works during **`adv_blk`**$_{put}$.

Consider the following scenario (Figure 3.5) — a FIFO BWoS queue is initialized with capacity of 16 items having 4 blocks, and therefore 4 elements per block. The manner in which the metadata variables are initialized are discussed in the implementation (4) section of this report. In step (1), the queue is filled with *pushBottom()* and a stealer reserves two entries in block 1 with *popTop()* thus setting `s_pos` to 2. Due to the atomic read-modify-write operations that occur during steal, the two entries are only reserved (`s_pos`(1, 2)) and not stolen yet (`s_cnt`(1, 0). Concurrently, in step (2), *popBottom()* empties block 0 and thus sets `f_pos` to the block margin. Following that, in step (3), *pushBottom()* fills the queue again after a wraparound which updates `b_pos` version to 2. Finally, in step (4), *popBottom()* empties the queue starting with block 1 (**`adv_blk`**$_{get}$) and wrapping around to block 0. In this process, the *popBottom()* operation stops the stealers in block 1 with a takeover that atomically sets `s_pos` to the block margin and adds the difference between the old `s_pos` and block margin to `s_cnt`(1, 2). The wraparound to block 0 updates the `f_pos` version to 2. In this state, the queue is completely empty. However, we cannot add new items to the queue. This is because since `b_pos` is at the block margin in block 0 (`blk_to_put`), it must advance to block 1. To do this, as explained in subsection 4.4.1, the queue checks if thieves have finished reading all data by checking if `s_cnt` is equal to block margin (NE) and their version is $x - 1$. While the latter is satisfied in this case, the former is not. Hence, no new elements can be added to the queue.

This is a big drawback to the FIFO BWoS queue which hasn't been discussed in their paper by Wang *et al*. In theory, the version check already determines if the thieves have finished stealing elements of the previous round and thus, we could omit the check for `s_cnt` with block margin. However, this would require further exploration with a verification checker to ensure correctness of all possible execution scenarios mainly in terms of consistency i.e. each element that has been pushed is read once by either the owner or the thief before the new round of operations.

[1] Commit `60d9ed7`, file `bwosqueue/tests/blocked_stealer.rs`, line 2: https://github.com/tokio-rs/tokio/blob/60d9ed76b35f504e2423213586f40ef49fbf3d8e/bwosqueue/tests/blocked_stealer.rs#L2

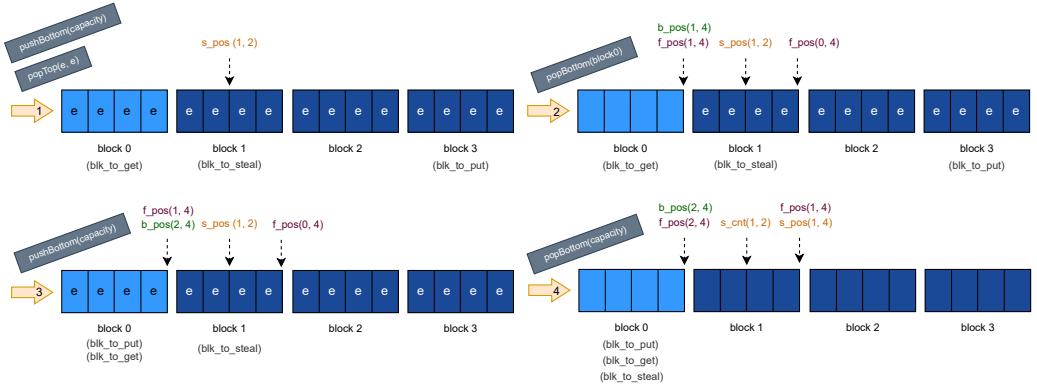

Figure 3.5: Pathological Issue with FIFO BWoS

3.3 Comparison with ABP and CL Queues

To illustrate the structural and operational differences between BWoS and classical work-stealing deques such as the ABP queue and the CL deque described in the background section (2) of this report, it is helpful to examine the high-level logic of their queue operations. These operations form the fundamental interface through which owners and thieves interact with their respective queues. The pseudocode sketches in **??** are highly abstracted versions of the algorithm, to highlight the similarities and the differences of these queues at a conceptual level to focus on the main algorithmic structure. The full pseudocode for these algorithms can be found in Figure 2.6, Figure 2.8 and Figure 4.2.

```
ABP<E, CAPACITY>::pushBottom(E item) ->
  queue[Bottom] = item;
  Bottom ++;

ABP<E, CAPACITY>::popBottom(E *item) ->
  localBottom = --Bottom;
  oldTop = Top;
  *item = queue[localBottom];
  if(localBottom > oldTop) return true;
  Bottom = 0;
  return CAS(Top, oldTop, Top + 1);

ABP<E, CAPACITY>::popTop(E *item) ->
  *item = queue[Top];
  newTop = Top + 1;
  return CAS(Top, oldTop, newTop)
```

```
CL<E, CAPACITY>::pushBottom(E item) ->
  queue[Bottom % CAPACITY] = item;
  Bottom ++;

CL<E, CAPACITY>::popBottom(E *item) ->
  localBottom = --Bottom;
  localTop = Top;
  if(localTop > localBottom) return false;
  *item = queue[localBottom % CAPACITY];
  if(localBottom > localTop) return true;
  res = CAS(Top, localTop, localTop + 1);
  Bottom = localBottom + 1;
  return res;

CL<E, CAPACITY>::popTop(E *item) ->
  *item = queue[Top % CAPACITY];
  newTop = Top + 1;
  return CAS(Top, oldTop, Top + 1);
```

```
BWoS<E, CAPACITY, NE>::pushBottom(E item) ->
  block[b_pos] = item;
  b_pos ++;
  if(b_pos ≥ NE)
    adv_blk_put;

BWoS<E, CAPACITY, NE>::popBottom(E *item) ->
  *item = block[b_pos];
  b_pos --;
  if(f_pos ≥ NE)
    adv_blk_get;

BWoS<E, CAPACITY, NE>::popTop(E *item) ->
  localspos = s_pos;
  if (CAS(s_pos, localspos, localspos + 1))
    *item = block[localspos];
    atomic(s_cnt++);
  else return LostRace;
  if(s_pos ≥ NE)
    adv_blk_steal;
```

Figure 3.6: Pseudocode for queue operations of ABP, CL and BWoS Queues

At a glance, all three designs share the high-level structure of bottom-push, bottom-pop, and top-steal operations. However, important differences emerge in how synchronization and memory layout are handled. ABP and Chase-Lev (CL) operate over a single contiguous array, where the owner and thieves coordinate through shared `top` and `bottom` indices. A thief's *popTop* is a CAS on the `top` pointer, and the owner must perform a CAS in *popBottom* only when removing the last item to prevent races [ABP98]. In contrast, BWoS deliberately splits each per-core queue into multiple

fixed-size blocks, each with its own metadata that includes head/tail counters (`f_pos`/`b_pos`). The owner and thieves operate mostly on separate blocks: owners push and pop tasks within their current block without needing CAS operations, while thieves only perform CAS when stealing from their designated block. This design fully decouples owner and thief operations at the block level, reducing synchronization overhead and minimizing contention across threads.

Memory management also diverges significantly between the designs. The ABP queue statically allocates a shared buffer divided among all threads, while the CL queue allows for dynamic reallocation to accommodate growth. BWoS instead fixes the total queue capacity and partitions it into N equally sized blocks (e.g., a queue with capacity 1024 split into 8 blocks). This block-partitioned buffer simplifies memory management, avoids unbounded growth, and constrains per-block task counts. BWoS also contains 3 pointers for each queue, and four atomic variables as its metadata which is quite a bit more than the ABP and CL queues. While cache padding is added to prevent false sharing, the memory overhead from this metadata is also something to take into consideration while conducting an evaluation.

3.3.1 Memory Barrier Analysis

	pushBottom()	**popBottom()**	**popTop()**
bottom	RW	RW	R
top	-	$R(W)_{CAS}$	$R(W)_{CAS}$

Table 3.1: Metadata updates in queue operations for ABP Queue

	pushBottom()	**popBottom()**	**popTop()**
bottom	RW	RW	R
top	R	$R(W)_{CAS}$	$R(W)_{CAS}$

Table 3.2: Metadata updates in queue operations for CL Queue

Efficient work-stealing queue designs depend critically on how atomic operations and memory barriers are placed along common execution paths. In both the ABP and CL queues, as shown on Table 3.1 and Table 3.2, *popBottom()* and *popTop()* interact with the `top` counter via an atomic read and a CAS. An observation can be made that both the owner and stealer must read and attempt to update the same shared variable `top`, which causes contention in multi-stealer scenarios. Aside from this, an acquire-release barrier is required while reading and updating the `bottom` counter on owner operations in both queues.

	pushBottom()		**popBottom()**		**popTop()**	
	fastpath	**adv_blk_{put}**	**fastpath**	**adv_blk_{get}**	**fastpath**	**adv_blk_{steal}**
b_pos (committed)	RW	W	R	R	R	-
f_pos (consumed)	-	R	RW	W	-	-
s_pos (reserved)	-	W	-	$R(W)_{exchange}$	$R(W)_{CAS}$	R
s_cnt (stolen)	-	RW	-	$(RW)_{fetch\_add}$	$(RW)_{fetch\_add}$	-

Table 3.3: Metadata updates in queue operations for FIFO BWoS

	pushBottom()		popBottom()		popTop()	
	fastpath	**adv_blk_{put}**	**fastpath**	**adv_blk_{get}**	**fastpath**	**adv_blk_{steal}**
b_pos (committed)	RW	W	R	R	R	-
f_pos (consumed)	-	R$(W)_{exchange}$	RW	RW	-	-
s_pos (reserved)	-	RW	-	$(W)_{exchange}$	R$(W)_{CAS}$	R
s_cnt (stolen)	-	RW	-	-	$(RW)_{fetch\_add}$	-

Table 3.4: Metadata updates in queue operations for LIFO BWoS

We compare that to the BWoS queue as shown in Table 3.3 and Table 3.4 where crucially, most operations in BWoS take a fast path inside a block where there are no CAS operations on the shared metadata. Additionally, the LIFO BWoS only has relaxed memory barriers on the fast path while FIFO BWoS has one acquire-release fence to enforce visibility. Only when a block becomes empty (for owners) or exhausted (for thieves) does a block-advancement step involving atomic updates occur. Because these block transitions are infrequent and localized, multiple thieves can steal simultaneously from different blocks, and the owner can operate independently without interference. The result, in theory, is dramatically higher concurrency. As reported by the BWoS authors, the design yields up to an order-of-magnitude faster micro-execution performance than CL-style deques and delivers substantial speedups ($\sim$ 25%) on complex real-world parallel workloads [Wan+23].

However, the trade-off lies in the additional metadata synchronization. We can observe that the *popTop()* in FIFO BWoS performs an atomic fetch_add on a shared stealing counter which adds a read-modify-write penalty even in uncontended cases. This is in addition to a CAS opeartion on the shared `s_pos` metadata. Additionally, there are two atomic exchanges that occur in the block-advancement of the owner in both LIFO and FIFO BWoS which can add overhead when moving across blocks.

3.3.2 Management of Free Slots

Finally, there is a key distinction among these implementations in how free slots—representing the capacity available for pushing new tasks—are defined and managed. In the ABP and CL queues, the number of used slots is typically determined by the difference between the `bottom` and `top` indices. Consequently, the number of free slots is calculated as the queue's total capacity minus this difference. In contrast, the BWoS queue departs from this linear model due to its explicit block ownership semantics. To recap, light blue (LB) blocks represent blocks owned by the owner and dark blue (DB) blocks represent blocks owned by the thieves. Ownership is strictly regulated as shown in Figure 3.2, Figure 3.3 and Figure 3.4: the owner can only operate on a single LB block at any given time and cannot own multiple blocks simultaneously. Even though empty blocks are LB, the owners cannot technically work on them until a block advancement occurs. As a result, even though *popTop()* may have released space in DB blocks, these blocks are not considered to have "free" slots from the owner's perspective until the takeover procedure reclaims them. Thus, in BWoS, free slots are localized to the owner's current block, and the presence of space in DB blocks does not contribute to the queue's available capacity until ownership transitions occur. This introduces a fundamental departure from traditional work-stealing queues, where free space is globally derived from index arithmetic rather than tied to local block semantics.

4 Implementation

4.1 A C++ Implementation of the BWoS Queue

For analyzing the work-stealing queue implementations, we leverage pre-existing implementations from the *EMPER* (Efficient Massive Parallelism Execution Realm) concurrency platform [Flo25], including a lock-free ABP queue, CL queue, and a simple locked queue for baseline comparisons. We extended EMPER, which is primarily developed in C++, by implementing both FIFO and LIFO variants of the BWoS queue. Notably, our implementation remains independent of EMPER-specific programming constructs and can function as a standalone program.

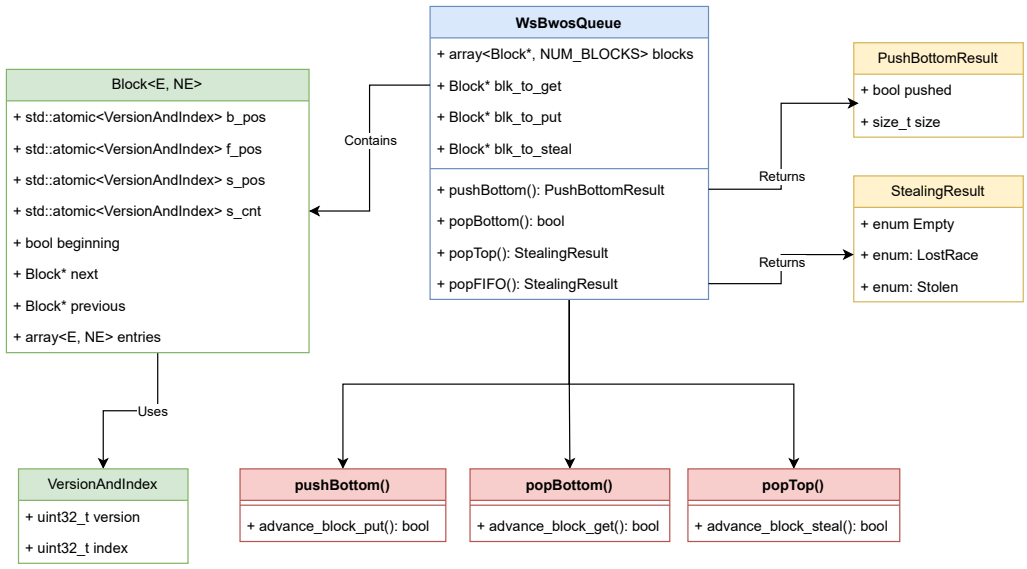

Figure 4.1: Class and functional hierarchy of BWoS Queue implementation

The class diagram in Figure 4.1 shows the main components of the queue which are described in the sections below.

4.2 Structure of a Block

In the implementation of a BWoS queue as described in the design section 3, the most fundamental unit of the queue is the **Block**. The entire algorithm is based around the structure and metadata

variables of a block and how they are updated to facilitate block advancement in owner blocks and thief blocks. Each block maintains four critical atomic variables; each aligned to a cache line to minimize false sharing and adhere to the cache coherency protocol [DB82]. The purpose of these variables differs slightly between the FIFO and LIFO variants of the BWoS queue.

- **b_pos (committed)** - Tracks the index up to which entries in the queue have been committed (i.e. enqueued)
- **f_pos (consumed)**
  - In FIFO BWoS, it tracks the index up to which entries have been consumed (i.e. dequeued).
  - In LIFO BWoS, it acts as a boundary for valid data
- **s_pos (reserved)** - Indicates the stealing location in the block (i.e. items reserved for steal)
- **s_snt (stolen)** - Tracks how many entries have been stolen by thieves.

Each of these metadata fields is declared as a `std::atomic<VersionAndIndex>`, and a static assertion checks that these atomics are always lock-free. The `VersionAndIndex` structure comprises of two 32-bit unsigned integers **version** and **index** that represent order of the last data access and the entry index within the block respectively. By combining these into a single 64-bit entity, `VersionAndIndex` ensures atomic updates can be performed efficiently on modern 64-bit architectures. A static assertion enforces that the structure occupies exactly 8 bytes, maintaining alignment and atomicity guarantees.

Blocks are arranged in a doubly linked structure using two raw pointers: *next* and *preceding* which point to the next block and the preceding block in the queue in a circular ring buffer fashion. This bidirectional linkage enables both forward and backward traversals. The block itself stores an array of NE elements of type E, which in our evaluation benchmarks are just raw integer values. This statically sized array allows for efficient batch operations and minimizes dynamic memory allocation overhead. Additionally, the 3 block pointers described in subsection 3.1.2 are all set to the first index (0).

A small note on to the queue itself — the number of blocks in a queue (NUM BLOCKS) is calculated based on two template arguments CAPACITY and ENTRIES PER BLOCK that the user needs to specify when initializing the queue with the only restriction being that NUM BLOCKS must be a power of 2.

4.2.1 Block Metadata Initialization

During construction, a block is initialized based on its position in the queue and the queue's operational variant (FIFO or LIFO). The key difference as shown in Table 4.1 is that LIFO doesn't use the version and the values are always set to 0.

This setup ensures that `blk_to_get` in FIFO BWoS (i.e. the first block in the queue) always belongs to the owners while rest of the blocks are owned by thieves. Whereas in LIFO BWoS, since `blk_to_get` and `blk_to_put` always point to the same block and operate at the bottom of the queue, all the blocks during initialization belong to the owner.

	FIFO		LIFO
	first block	**remaining blocks**	
b_pos (committed)	VersionAndIndex(1, 0)	VersionAndIndex(0, NE)	VersionAndIndex(0, 0)
f_pos (consumed)	VersionAndIndex(1, 0)	VersionAndIndex(0, NE)	VersionAndIndex(0, 0)
s_pos (reserved)	VersionAndIndex(1, NE)	VersionAndIndex(0, NE)	VersionAndIndex(0, NE)
s_cnt (stolen)	VersionAndIndex(1, NE)	VersionAndIndex(0, NE)	VersionAndIndex(0, NE)

Table 4.1: Metadata initialization in BWoS

4.3 Single-Block Operations (Fast Path)

Now we define the three queue operations *pushBottom()*, *popBottom()* and *popTop()* and firstly, their fast path i.e. their intra block operation.

4.3.1 pushBottom()

```
1  WsBwosQueue::pushBottom(E item) -> pushBottomResult {
2    while(true) {
3      Block* blk = blk_to_put;
4      if(b_pos < NE) {
5        blk->entries[b_pos.index] = item;
6        blk->b_pos.index ++;
7        return {pushed = true, size = b_pos.index};
8      }
9      adv_blk_put();
10   }
11 }
12
```

Figure 4.2: Fastpath for pushBottom()

To push an item into the queue (Figure 4.2), *pushBottom()* first takes the current blk_to_put and checks if the `b_pos` (committed) index reaches the block margin (number of entries) (lines 3 – 4), if not, enqueues the item into the queue and increments the `b_pos` index (line 6). If `b_pos` index is already full, then it goes to the block advancement procedure (line 9) defined in subsection 4.4.1.

4.3.2 popBottom()

```
1  WsBwosQueue::popBottom(E* item) -> bool {
2    while(true) {
3      Block* blk = blk_to_get;
4      if(f_pos < NE) {
5        if (b_pos.index == f_pos.index)
6          goto slowpath;
7        *item = blk->entries[f_pos.index];
8        if (FIFO)
9          blk->f_pos.index ++;
10       else if (LIFO)
11         blk->b_pos.index --;
12       return true;
13     }
14   slowpath:
15     adv_blk_get();
16   }
17 }
18
```

Figure 4.3: Fastpath for popBottom()

To pop an item from the queue (Figure 4.3), *popBottom()* first takes the current `blk_to_get` and checks whether the `f_pos` (consumed) index reaches the block margin (lines 3 – 4), or if the block has run out of data (line 5), if not, dequeues the item and increments the `f_pos` index (line 9 for FIFO variant) or decrements the `b_pos` index (line 11 for LIFO variant). If `f_pos` index is already full, it goes to the block advancement procedure (line 15 defined in subsection 4.4.2.

4.3.3 popTop()

```
1  WsBwosQueue::popTop(E* item) -> StealingResult {
2    while(true) {
3      Block* blk = blk_to_steal;
4      if(s_pos < NE) {
5        if (s_pos.index == b_pos.index)
6          return Empty;
7        VersionAndIndex localspos = s_pos;
8        localspos.index += 1;
9        if (blk->s_pos.CAS(s_pos, localspos))
10         *item = blk->entries[s_pos.index];
11         blk->s_cnt.fetch_add(1);
12         return Stolen;
13       return LostRace;
14     }
15     Block* nblk = blk->getNextBlock();
16     adv_blk_steal(nblk);
17   }
18 }
19
```

Figure 4.4: Fastpath for popTop()

To steal an item from the queue (Figure 4.4), *popTop()* first takes the current `blk_to_steal` and checks whether the `s_pos` (reserved) index reaches the block margin (lines 3 – 4), or if the block has run out of data (line 5). It then atomically updates `s_pos` using a CAS to point to the next entry, reads the item, and finally increments the stolen count (`s_cnt`) with an atomic `fetch_add` (lines

9 – 11). If the CAS fails, then that means the current thief thread lost the race to pop the item i.e. either another concurrent thief succeeded, or the owner thread took over the block and dequeued the item with *popBottom()*. If s_pos index is already full, it goes to the block advancement procedure (line 16) defined in subsection 4.4.3.

4.4 Block Advancement

In the event that the relevant metadata variable reaches the block margin NE in any of the fastpath operations, the operations move to the next block to work on. If advancing is possible, then takeover (by *popBottom()*) allows owners to take ownership of the block from thieves and grant (by *pushBottom()*) allows owners to grant ownership to the current block to the thieves.

4.4.1 adv_blk_put

```
1  adv_blk_put() {
2    Block* nblk = blk->getNextBlock();
3    if(FIFO) {
4      blk_version = if(nblk.isHead()) ? b_pos.version + 1 : b_pos.version;
5      expected_f_pos = blk_version - 1;
6      bool is_consumed = ((nblk->f_pos.inedx == NE) && (expected_f_pos == nblk->f_pos.version));
7      if(!isConsumed) return false;
8      bool no_stealers = ((nblk->s_cnt.index == NE) && (expected_f_pos == nblk->s_cnt.version));
9      if(!noStealers) return false;
10     // grant next blk to thief
11     nblk->b_pos = VersionAndIndex(blkVersion, 0);
12     nblk->s_pos = VersionAndIndex(blkVersion, 0);
13     nblk->s_cnt = VersionAndIndex(blkVersion, 0);
14     blk_to_put = nblk;
15   } else if (LIFO) {
16     if(s_pos != NE) return false;
17     // grant current blk to thief
18     old_f_pos = blk->f_pos.exchange(VersionAndIndex(0, NE);
19     blk->reserved = old_f_pos;
20     // takeover next block by owner
21     nblk->b_pos = VersionAndIndex(0, 0);
22     nblk->f_pos = VersionAndIndex(0, 0);
23     nblk->s_pos = VersionAndIndex(0, NE);
24     nblk->s_cnt = VersionAndIndex(0, 0);
25
26     blk_to_get = blk_to_put = nblk;
27   }
28 }
29
```

Figure 4.5: Block Advancement for pushBottom()

FIFO - To check whether block advancement is possible for *pushBottom()*, the FIFO queue employs the round control procedure to check if the items in the next block have already dequeued by *popBottom()* or stolen by *popTop()*. It does this by checking if the f_pos (consumer) and s_cnt (stolen) index in the next block are equal to NE and if their version numbers are equal to $x - 1$ where x is the b_pos version of the current block. If these conditions do not hold, then it means that there are active *popBottom()* or *popTop()* operations ongoing in the block. If these conditions hold, then the metadata variables b_pos, s_pos and s_cnt are updated with version x and index 0, thus resetting the block for the next round of *pushBottom()*. In the case of a wrap around, the b_pos version of the

current block is incremented by 1 and this new version is then used for the next round of queue operations.

LIFO - In the LIFO variant, the `adv_blk_put` just checks if the `s_pos` index is equal to NE indicating that there are no stealers and the block can be taken over. If this condition holds, then first it grants the current block to thieves by atomically exchanging `s_pos` with NE and setting `f_pos` to the previous `s_pos` value. Second it performs takeover of the next block by setting the `s_pos` index to NE and resetting the index of the remaining three metadata variables to 0.

4.4.2 adv_blk_get

```
1  adv_blk_get(FIFO) {
2    Block* nblk = blk->getNextBlock();
3    curr_f_pos_version = blk->f_pos.version
4    if(nblk.isHead())
5      expected_f_pos_version = curr_f_pos_version + 1;
6    else
7      expected_f_pos_version = curr_f_pos_version;
8    if(nblk->s_pos.version != expected_f_pos_version)
9      return false;
10   //Stop stealers
11   old_s_pos = nblk->s_pos.exchange(VersionAndIndex(↩
         expected_f_pos_version, NE);
12   nblk->s_cnt.fetch_add(NE - old_s_pos.index);
13   nblk->f_pos = old_s_pos;
14   blk_to_get = nblk;
15 }
16
```

```
1  adv_blk_get(LIFO) {
2    Block* pblk = blk->getPrecedingBlock();
3    old_s_pos = pblk->s_pos.exchange(VersionAndIndex(0, NE)↩
         ;
4    pblk->f_pos = old_s_pos;
5    if(old_s_pos.index != pblk->b_pos.index) {
6      if(blk_to_steal == pblk)
7        blk_to_steal = blk_to_steal - 1;
8      blk_to_get = blk_to_put = pblk;
9    }
10   return false;
11 }
12
```

Figure 4.6: Block Advancement for popBottom()

FIFO - To check whether block advancement in possible for *popBottom()*, the FIFO queue checks if there are any *pushBottom()* operations happening in the next block. This is done by checking if the `f_pos` version of the current block is equal to the `s_pos` version of the next block. In this case, `s_pos` is last updated by *pushBottom()* so this version denotes the last data access for enqueue operations. If these versions are equal, that means the takeover is possible. For this, it will first stop stealers in the next block by atomically exchanging `s_pos` index with NE and version with the `f_pos` version of current block, setting `f_pos` to the previous `s_pos` value and adding the difference between the new `f_pos` and NE to the `s_cnt`. This difference indicates the number of entries that cannot be stolen anymore due to the takeover. It is important to note that this is misleading as per the strict definition of the `s_cnt` (stolen) metadata because the items denoted by the difference are not actually stolen (but will be dequeued by *popBottom()* after the current advancement). Similar to `adv_blk_put`, in case of a wrap around, the `f_pos` version of the current block is incremented by 1 and this new version is used for the next round of queue operations.

LIFO - In the LIFO variant, in exactly the same fashion as the FIFO variant, an atomic exchange is performed with `s_pos` and NE and the `f_pos` is set to the previous `s_pos` value, this time on the previous block. However, this time, in order to stop the stealers in the previous block, we first check if the block is empty and if not, decrement the index of `blk_to_steal` so that owners and thieves do not operate on the same block. After this is successful, the `blk_to_put` and `blk_to_get` is updated to point to the previous block.

4.4.3 adv_blk_steal

```
1  adv_blk_steal() {
2    Block* nblk = blk->getNextBlock();
3    if(FIFO) {
4      expected_s_pos_version = blk->s_pos.version + (nblk->isHead() ? 1 : 0);
5      if(nblk->s_pos.version != expected_s_pos_version)
6        return false;
7    } else if (LIFO) {
8      if(nblk->s_pos.index == NE)
9        return false;
10   }
11   Block* desired = blocks[blk_to_steal + 1];
12   blk_to_steal.CAS(blk_to_steal, desired);
13
14 }
15
```

Figure 4.7: Block Advancement for popTop()

FIFO - To check whether block advancement is possible for *popTop()*, the FIFO queue checks if there are already active stealers in the next block by checking if the next block's `s_pos` (reserved) version is equal to the current block's `s_pos` version, unless it is a wrap around in which case the comparison is incremented by 1. If the condition holds true, the `blk_to_steal` pointer is updated to point to the next block in the array.

LIFO - Similarly in the LIFO queue, if there are no active stealers in the next block, the `blk_to_steal` pointer is updated in the same way.

5 Evaluation

To assess the practical performance and design trade-offs of the work-stealing queue implementations mentioned in this report, this evaluation section focuses on both microbenchmarks and real-world application benchmarks. The goal is to compare how different queues behave under controlled conditions and in end-to-end scenarios with varying workloads. Four queue designs are evaluated: LIFO BWoS, the ABP Queue, the CL Queue as well as a standard mutex based locked queue for baseline comparisons. FIFO BWoS is notably excluded from these benchmarks because of the pathological issue (discussed in section 3.2) causing the queue to deadlock. This issue can be resolved by allowing batched operation on steals but since the ABP and CL Queues do not allow batched queue operations, the benchmark results would be inequitable. The microbenchmark isolates core queue operations to measure performance characteristics such as throughput, latency, and steal behavior using varied payload sizes. Complementing this, application benchmarks demonstrate how these queues perform under different computational and I/O-intensive workloads. Together, these benchmarks provide a comprehensive picture of the queues' suitability for different task-parallel workloads. Exemplary results in the form of visual charts are provided throughout this chapter which dissect the differences between the various approaches and help us draw meaningful conclusions.

5.1 Setup

All experiments listed below are performed on a system with an AMD Ryzen 75800H processor (x86 architecture) with 8 CPU cores which have AMD simultaneous multithreading support to effectively double the core count to 16 threads. This particular processor has 16 MB of L3 cache and operates at 3.2 GHz by default, but can boost up to 4.4 GHz, depending on the workload. The operating system is Ubuntu 24.04.1 LTS running on Windows Subsystem for Linux (WSL2) with Linux kernel version 5.15.167.4-microsoft-standard-WSL2.

The microbenchmark experiments are standalone programs which do not depend on constructs from any specific runtime environments. The application benchmarks, however, are part of the *EMPER* concurrency platform alluded to in the implementation section (4) of this report. EMPER is a concurrency platform with a programming language abstraction and an associated runtime system that implements both lock-free child-stealing and wait-free "Nowa" continuation-stealing approach as described by Schmaus *et al.* [Sch+21b].

It is important to note that this minimal setup introduces several key factors that directly impact benchmark results when compared with benchmark results in the research papers.

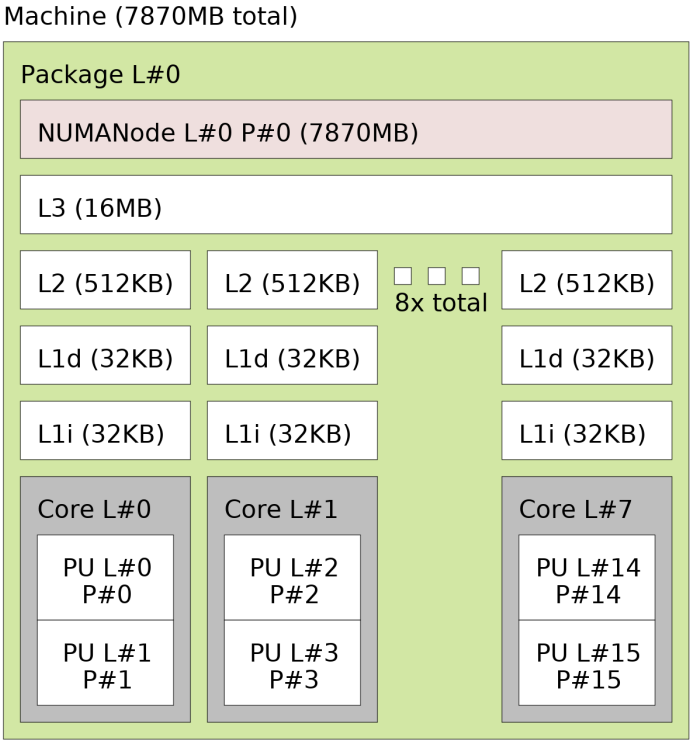

Figure 5.1: Processor and Memory heirarchy of experimental setup

- **Cache Hierarchy** - Algorithms like BWoS are designed to exploit high concurrency and cache locality. It would be at its most performant when whole blocks can be stolen instead of single tasks. This would then be scaled with the amount of L1/L2 cache contention which cannot be realized in this small-scale setup.
- **Execution Environment** - Even though WSL2 is efficient, it adds overhead on system calls, scheduling and memory ordering. Performance critical lock-free queues are sensitive to even small sources of latency.
- **Benchmark Setup** - The current benchmark only consists of a pool of one owner thread and multiple stealer threads. Calculating improvements for scalable algorithms accurately might require a multi-thread pool design.
- **Implementation Fidelity** - Even though well documented designs for work-stealing algorithms are used, the implementation details such as use of atomics, cache padding, queue size tuning may also result in a performance hit.

5.2 Microbenchmarks

The microbenchmark simulates a typical producer-consumer scenario where a single worker thread repeatedly executes a workload loop for a configurable duration. In each loop iteration, the worker enqueues items into the queue until it is full, then dequeues items until the queue is empty. Concurrently, one or more stealer threads attempt to steal work from the queue at a user-defined

frequency. The benchmark exposes several configurable parameters to allow fine-grained control over the workload:

- **num-stealers** - Number of stealer threads to launch for the benchmark run.
- **steal-frequency** - The frequency (in Hz) at which the stealer should carry out the *popTop()* operation.
- **queue-implementation** - The queue implementation to use for the current benchmark run.
- **duration-ms** - The total duration of the benchmark run in milliseconds (default to 10000ms or 10 seconds for all benchmark runs in this report).

Each run reports:

- Worker throughput (*pushBottom()* and *popBottom()* operations/sec)
- Stealer throughput (*popTop()* operations/sec)
- Total throughput (sum of all above operations/sec)
- Average time per push/pop operation (in nanoseconds)
- Average time per steal operation (in nanoseconds)

All reported results are mean values of five benchmark runs, unless otherwise stated.

To establish baseline performance, we configure the benchmark with zero stealers and a steal frequency of zero. The worker thread runs a loop that executes *pushBottom()* until the queue is full and then executes *popBottom()* until the queue is empty. This loop runs for the duration that was configured at runtime. We opt to check the deadline for every n operation to minimize the performance cost of querying the system clock which has a non-negligible overhead. By amortizing the cost of the time check over many operations, we can ensure that overshoot is minimal and predictable, and the actual performance is the key metric being measured. When the deadline is reached, a shared atomic flag is updated to indicate the end of the benchmark run.

When stealer threads are introduced, each one repeatedly calls the *popTop()* operation on the queue in a loop while respecting a target steal frequency (steals per second). To regulate this steal frequency, the thief thread employs a busy-wait delay mechanism between successive steals which is representative of a real-world workload scenario where steals are rare.

$$D = \left\lfloor \frac{10^9}{f_s} \right\rfloor - T_s$$

where:

f_s is the target steal frequency (in steals/second)
T_s is the time taken for the last steal attempt (in nanoseconds)
D is the computed delay before the next steal (in nanoseconds)

We convert all time-based calculations to nanoseconds to ensure precision. The performance characteristics of the queue are affected by both successful steals, which remove work from the queue, and failed steals, which contribute contention without productive work. From a throughput perspective:

- **Successful Steals** contribute to useful work being processed concurrently with the worker thread.
- **Failed Steals** (Empty and LostRace) introduce contention and CPU cycles without productive output, degrading the throughput
- **Average Time Per Steal** to monitor overhead of *popTop()* in each queue implementation.

At the end of each benchmark run, the consistency of the queue is validated by checking if total push operations are equal to total pop + total steal + remaining items in the queue. This check ensures that there were no duplicate or phantom data read/writes in the benchmark.

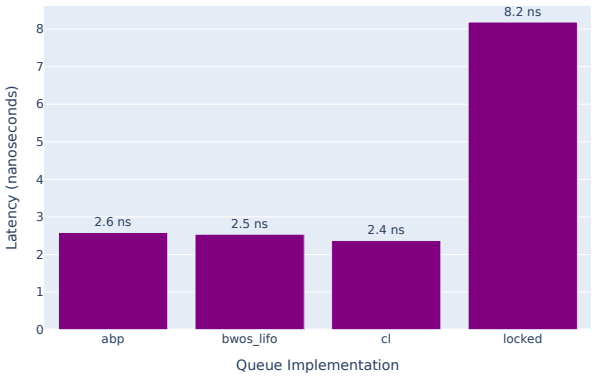

Figure 5.2: Latency of push and pop operations on owner thread

5.2.1 Single-Threaded Throughput

Figure 5.3 and Figure 5.4 for steal frequency equal to 0 show the performance of the queue without any stealer threads. In this run, each queue has a capacity of 1024 entries, with 4-byte integer items as payload and the LIFO BWoS queue is configured to have 8 blocks with 64 entries per block. All the queues have similar performance levels with the CL queue recording the highest throughput by a small margin and the Locked queue recording the lowest by a large margin. Additionally, the results in Figure 5.2 also backs these results seeing as average latency for push and pop operations on the owner side are quite similar for all 3 queue implementations except, again, the locked queue. This means that in the absence of thieves, the owner throughput is nearly identical in all cases. We can see that despite the presence of an additional atomic read-modify-write (CAS) operation to reset the top counter in both the ABP and CL queues when compared to LIFO BWoS, this does not affect the sequential performance of the queues in the absence of thieves. On the other hand, the memory overhead from additional metadata variables and pointers in the BWoS queue might add additional overhead in this use case.

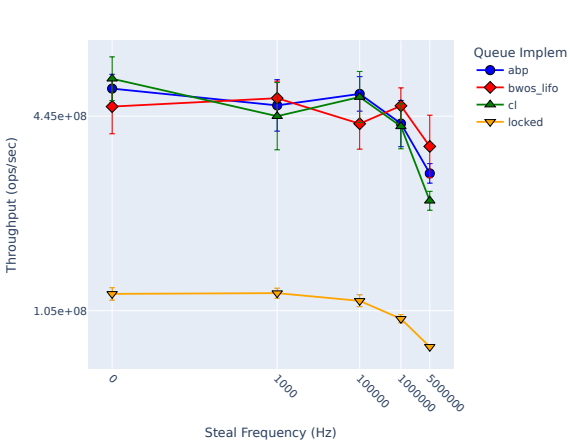

Figure 5.3: Throughput of LIFO BWoS, ABP, CL and Locked queues with 1024 item capacity and 1 stealer thread. Block size for BWoS is 64 (8 blocks)

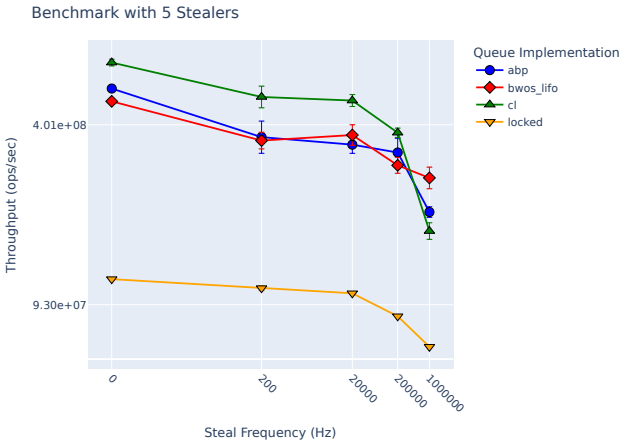

Figure 5.4: Throughput of LIFO BWoS, ABP, CL and Locked queues with 1024 item capacity and 5 stealer threads. Block size for BWoS is 64 (8 blocks)

5.2.2 Throughput in the Presence of Thieves

As soon as a stealer thread is introduced, the overall throughput of the queue decreases due to overhead from contention. Steal frequencies are adjusted based on the number of stealers to ensure fairness and avoid queue inconsistencies (stolen items > pushed items). Again, we see quite similar performances across increasing steal frequencies with LIFO BWoS having the most consistent throughput across increasing steal frequencies where it suffers only minor performance drops, which is consistent with the results found by Wang *et al.* [Wan+23]. This is also consistent for a single stealer thread and multiple stealer threads. For example, with 1 million steals per second and 5 stealer threads, BWoS outperforms ABP by 24%, the CL queue by 38% and the Locked queue by 95% when it comes to total throughput.

Table 5.1: Relative performance compared to LIFO BWoS

Metric	ABP	CL	Locked
Total Throughput	−24.07%	−38.51%	−95.30%
Steal Throughput	+6.81%	+7.42%	+24.35%
Worker Throughput	−24.30%	−38.80%	−96.16%

However, there is still performance left on the table, as by using 8-byte `int64_t` items as the payload, we can slightly decrease the percentage of cache misses as shown on Table 5.2. Moreover, this results in a 4.2% improvement in **Instructions Per Cycle** ratio which is a strong indicator of better CPU execution efficiency. Combine this with the fact that in the BWoS queue, larger block sizes limit the number of block advancements and ensure control flow stays in the fast path longer, we can make significant performance gains in our benchmark. We can apply these adjustments by increasing the block size of LIFO BWoS to 256 (4 blocks) and using 8-byte payload in the benchmark run. As shown in Figure 5.5 and Figure 5.6, these adjustments result in improved throughput performance, especially for the LIFO BWoS queue which now outperforms the ABP queue by 2x across increasing steal frequencies. The LIFO BWoS also has 1.5x the performance of the CL queue in higher steal frequency ranges and maintains its level of performance with increasing steal frequencies.

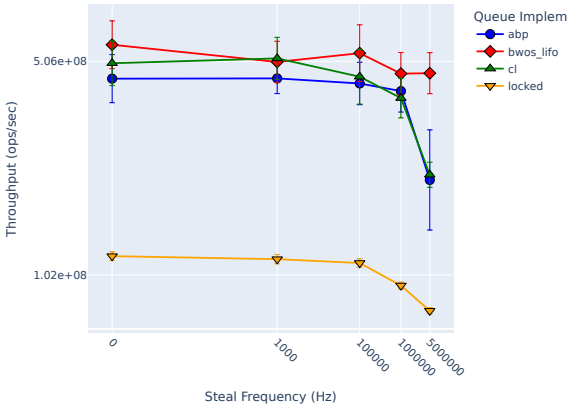

Figure 5.5: Throughput of LIFO BWoS, ABP, CL and Locked queues with 8-byte payload, 1024 item capacity and 1 stealer thread. Block size for BWoS is 256 (4 blocks)

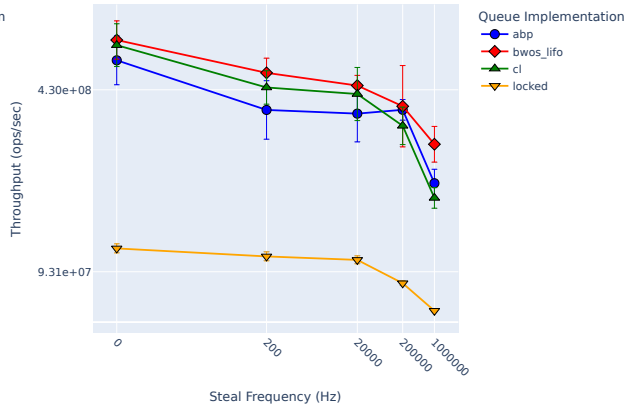

Figure 5.6: Throughput of LIFO BWoS, ABP, CL and Locked queues with 8-byte payload, 1024 item capacity and 5 stealer threads. Block size for BWoS is 256 (4 blocks)

Metric	**`int64_t`**	**`int`**
Cache References	23,768,562	21,427,227
Cache Misses	2,016,542	1,951,179
Cache Miss Rate	8.4%	9.1%
Instructions Per Cycle Ratio	1.97	1.51

Table 5.2: Cache performance comparison between `int64_t` and `int` in queue operations.

Block Sizes for BWoS: After a certain block size, the overhead of advancing the block doesn't yield in an improvement of throughput performance. The selection of block size and number of blocks should hence be fine-tuned according to the memory size constraints of the benchmark setup. These figures also do not account for the fact that having fewer blocks is detrimental to stealing as owners and thieves cannot work on the same block. These results are discussed below.

Steal Throughput: Figure 5.7 shows the steal throughput in a benchmark run with 2 stealers. The Locked queue has the highest rate of stealing. This is because the use of a global lock serializes access to the queue, ensuring that steal operations encounter minimal contention and no complex CAS operations. This is why there are no stealing attempts that return the status *Lost Race* for locked queue. In contrast, the BWoS queue has the lowest rate of stealing and the highest number of failed steals. This is due to the inherent characteristic of the BWoS design that prevents stealing when work items are less than the block size.

Table 5.3: Percentage of Failed Steal Reasons per Queue

Queue	**Queue Empty (%)**	**Lost Race (%)**
LIFO BWoS	92.16%	7.84%
ABP	66.15%	33.85%
CL	54%	46%
Locked	100.00%	0.00%

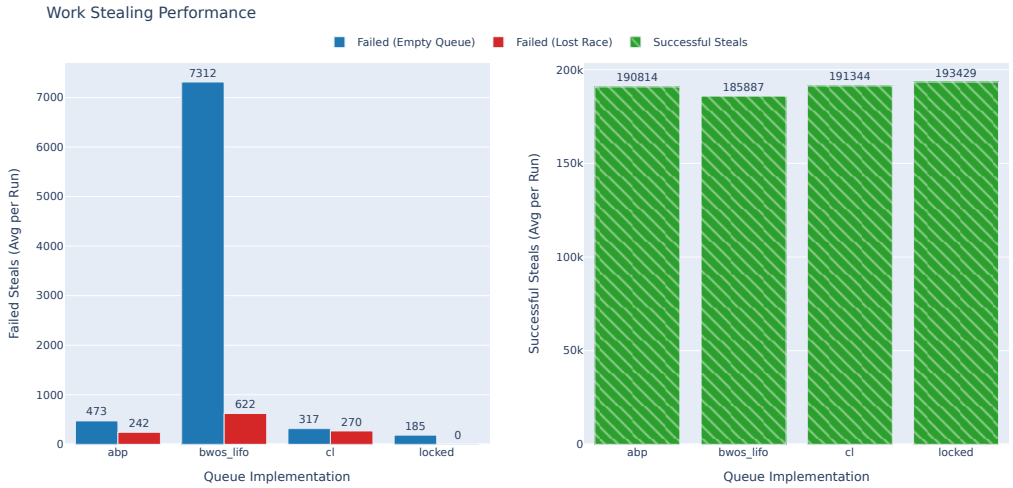

Figure 5.7: Distribution of the status of steal operations for ABP, LIFO BWoS, CL and Locked Queues. Benchmark run with 2 stealer threads.

In fact, ~ 95% of the failed steals in LIFO BWoS are due to an empty queue status. In reality, the queue is not always empty but blocks are owned by the owner (LB). So, thieves cannot work on those blocks until the owner grants ownership to the thieves. Furthermore, if remaining work items are less than the size of a block, then stealing is completely prevented if the owner has ownership of that block. Hence, from a statistical perspective, the queue state is not always as it is perceived to be. The advantage of the block-based design is that only ~ 7.5% of the failed steals occur due to a *Lost Race* status. This clearly indicates that the block-based design significantly lowers contention in concurrent queues. In comparison, only ~ 65% of the failed steals in the ABP queue are due to empty queues and ~ 35% are due to contention. Similarly, ~ 54% of the failed steals in the CL queue and are due to empty queues and ~ 46% are due to contention.

Median Time to Steal: Figure 5.8 and Figure 5.9 presents the distribution of steal latencies for the 3 lock-free queues with 1 stealer (steal frequency of 100, 000 Hz) and 5 stealers (steal frequency of 20000 Hz) respectively. The locked queue is excluded from this metric because due to the overhead that is incurred during contention on a mutex based lock, the median steal times are much higher than the lock-free queues. We can immediately see that the ABP queue demonstrates the lowest median latency with a narrow distribution profile around the median, indicating low variance and a predictable steal mechanism. The density peaks are tightly concentrated, confirming the consistency of their performance. The CL queue has slightly higher latency and a broader range, but it is still within expectation. It is interesting to observe that even with higher stealer threads, the density distribution for the CL queue remains relatively stable (116 ns for 1 stealer vs 106 ns for 5 stealers). The BWoS queue exhibits a noticeably higher median latency which is most likely due to the grant procedure on the owner side and the additional atomic fetch and add operation in fast path for *popTop()*. The fact that the LIFO BWoS keeps its performance level on par and even better than its classical counterparts despite the higher median steal latency proves that the block-based design of keeping control flow in the fast path is indeed effective in decreasing contention.

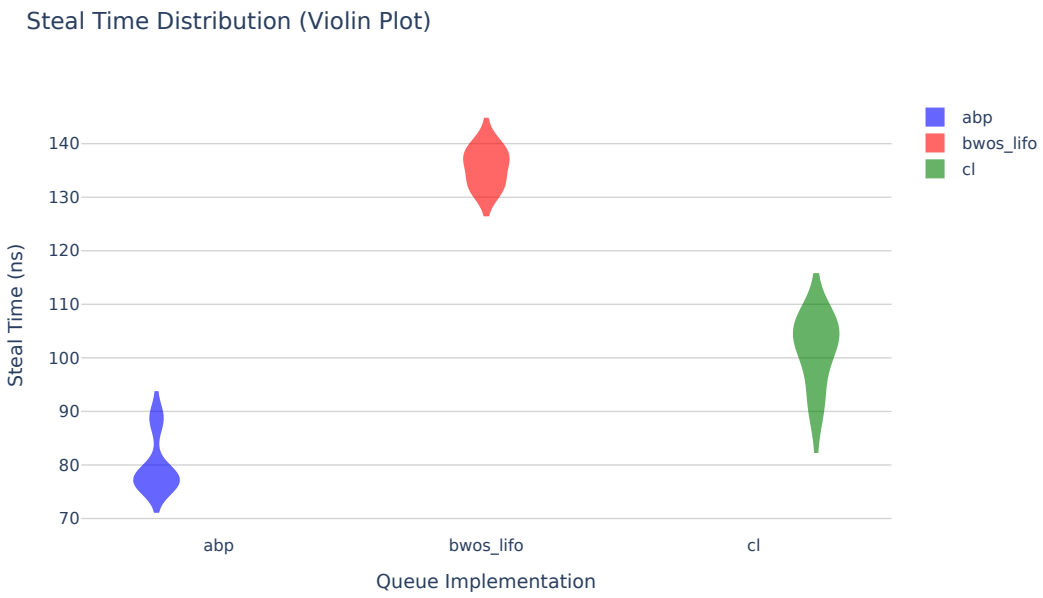

Figure 5.8: Latency of steal operations with one thief thread.

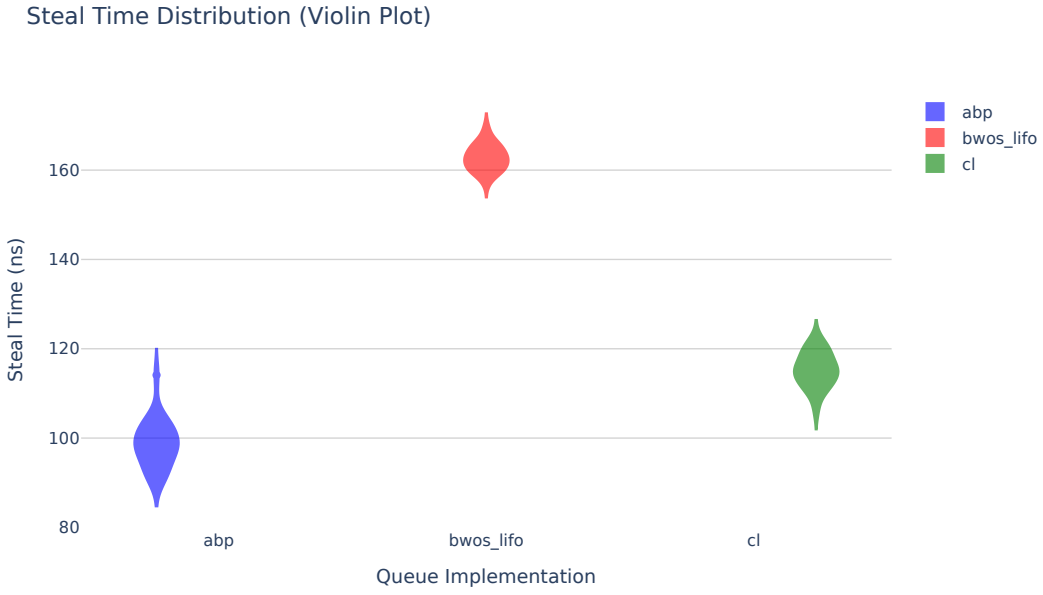

Figure 5.9: Latency of steal operations with five thief threads.

5.3 Application Benchmarks

Application benchmarks are standardized suites of tasks designed to emulate typical workloads within specific application domains. In the context of evaluating work-stealing algorithms, these benchmarks are used for assessing performance, scalability, and efficiency under realistic workloads.

Work-stealing schedulers dynamically balance load by allowing idle processor cores to steal tasks from busy ones, making them particularly effective for irregular and fine-grained parallel workloads. To accurately evaluate such schedulers, benchmarks must reflect the diverse computational patterns encountered in real-world applications. For the purposes of this report, we use 2 different application benchmarks - *Quicksort* and *fsFindDuplicate*.

5.3.1 Quicksort

Quicksort is a widely utilized sorting algorithm. Its divide-and-conquer nature makes it amenable to parallelization. In a parallel quicksort implementation, the recursive calls made after each partitioning step are independent and can thus be executed concurrently. To optimize performance, especially when dealing with small subarrays, a common strategy is to define a cutoff threshold: if the size of the subarray falls below this threshold, the algorithm switches to a sequential sorting method, such as insertion sort in our case. This approach avoids the overhead associated with parallelism when it's unlikely to yield significant benefits.

It is interesting to note that, as in many parallel quicksort implementations, the partitioning step remains sequential. This step involves selecting a pivot and rearranging the elements of the array so that those less than the pivot precedes it and those greater follows it. Since this process requires examining and moving each element, it inherently involves $O(n)$ operations, where n is the number of elements in the array. Consequently, the partitioning step becomes a bottleneck, limiting the algorithm's scalability and parallel efficiency. In our context though, this limitation is less critical where we are just evaluating the performance of various work-stealing implementations. Our primary interest lies in how effectively these implementations manage dynamic load balancing and task distribution during the recursive sorting phases.

The quicksort implementation used for this benchmark is based on the one used for the blog "Resource efficient Thread Pools with Zig" by Protty [Pro21]. There are 3 different block sizes of the LIFO BWoS queue used for the benchmark — to showcase how selection of block size is critical for performance based on the granular signature of the workload. The results are an average across 10 benchmark runs with an array of size $n = 10,000,000$ that are filled by 8-byte `int64_t` integers with a cutoff threshold of 32.

Queue	Voluntary Context Switches
CL	4,335
ABP	162
Locked	3,913
LIFO BWoS (8)	5,128
LIFO BWoS (16)	37,472
LIFO BWoS (64)	1,211,343

Table 5.4: Voluntary context switches by queue

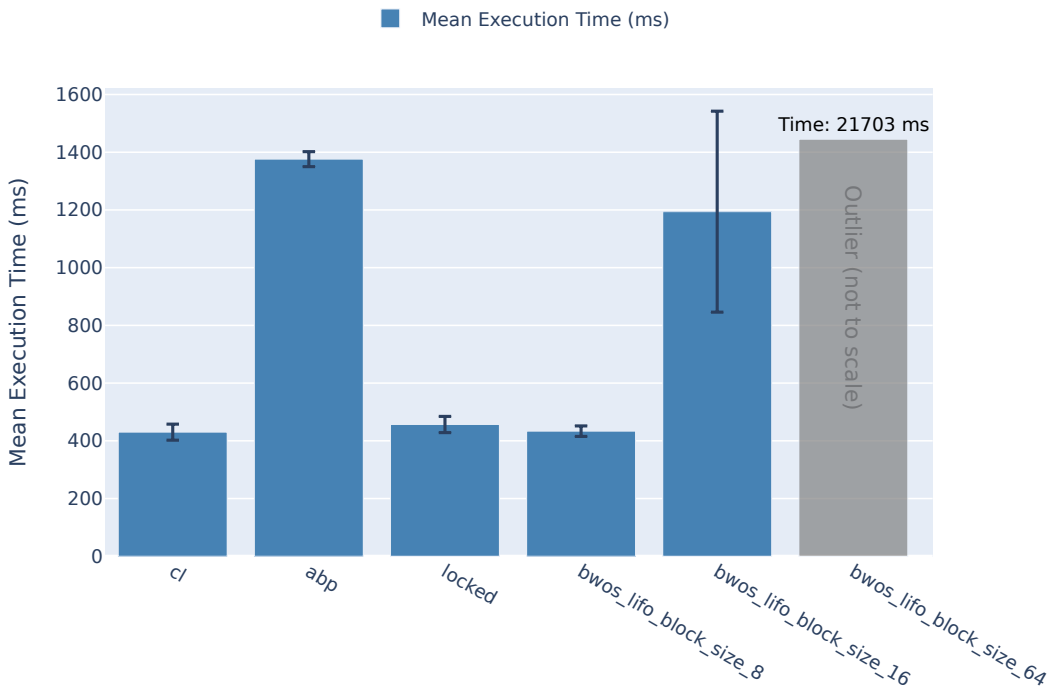

Figure 5.10: Mean execution times for Quicksort.

As shown in Figure 5.10 and Figure 5.11, the CL queue and LIFO BWoS (block size 8) have the best performance across the board when it comes to execution time and stealing performance. The LIFO BWoS queue shows dramatic performance variations depending on the block size. The block size 64 variation performs extremely poorly (~ 65x slower than block size 8) due to the excessive amount of context switches (Table 5.4). This is in part related to the fine-grained nature of the tasks in the quicksort benchmark that causes threads to frequently yield, due to higher number of lost races during contention which result in no work being stolen. This can be mitigated by adjusting the block size to be smaller which although incurs more block advancement, the overhead is still negligible compared to the performance gains achieved from parallelism.

Surprisingly, the locked queue performs competitively with the former two i.e. within ~ 5% of the median time, despite its poor average performance in our microbenchmarks as discussed above. This suggests that the lock contention may be low for benchmarks such as quicksort due to task locality. This result also challenges the conventional wisdom that lock-free data structures always outperform lock-based ones in concurrent scenarios. This highlights the importance of empirical evaluation over theoretical analysis alone.

The ABP queue's performance is notably worse, which is unexpected given its theoretical advantages and comparable performance in the microbenchmarks. Additionally, we recorded no stolen fibers and very few context switches across our benchmark runs which indicate that the scheduler failed to effectively redistribute work and the benchmark effectively ran sequentially. This might be due to a potential mismatch between the queue's design assumptions and the characteristics of our quicksort workload.

Figure 5.11: Total stolen fibers for each queue during Quicksort

Based on these findings, we can conclude that implementation complexity does not necessarily translate to better real-world performance and that effective work distribution is crucial for performance. The dramatic performance differences observed between BWoS configurations emphasize the critical importance of parameter tuning in work-stealing algorithms. For production environments, careful consideration should be given to both the queue implementation choice and its configuration parameters based on workload characteristics.

5.3.2 fsFindDuplicate

The `fsFindDuplicate` program, developed by Leonhard Kohn [Koh25], scans a given directory and its subdirectories in parallel to detect duplicate files by computing their hash values. It is implemented in EMPER and leverages asynchronous I/O using *liburing* [Axb25], which provides utilities to manage `io_uring` instances on Linux. The program supports both lock-free (child-stealing) and wait-free (continuation-stealing) work-stealing strategies; however, only the lock-free variant is used in the experiments presented. This application was selected for benchmarking because it provides valuable insights on how the different queue implementations perform under I/O bound conditions.

The process begins with a parallel recursive search of the directory tree, filtering for regular files. A user-defined function—in this case, hash computation—is applied to each file via EMPER's `recursive_directory_walk` which uses asynchronous system calls. Files are read in 4096-byte blocks, matching the standard memory page size for efficient I/O and reduced page faults. Hash values are stored in a global hash map, and an optional simulated computational load can be introduced to

emulate CPU-bound scenarios. Finally, files sharing the same hash are compared block by block to confirm duplication, with the first encountered file treated as the original and subsequent matches marked as duplicates. A directory of size 256 MiB with ~ 600 subdirectories and around 2500 files were used for this evaluation. The results shown in Figure 5.12 are an average across 10 benchmark runs.

Figure 5.12: Comparison of characteristics for fsFindDuplicate across queue implementations

Queue	x	σ	Fibers	Stolen	I/O Stolen	nvcsw	nivcsw
cl	549.7	29.0	11589	3903	291	14462	310
abp	567.7	65.7	10065	1979	42	12779	39
locked	538.4	37.8	12561	5256	714	11686	73
bwos_lifo(256)	823.0	38.1	5068	256	153	23307	103
bwos_lifo(64)	864.6	44.2	10081	597	91	31938	122
bwos_lifo(16)	882.3	40.0	10350	502	203	33891	196

Table 5.5: Raw values for fsDuplicate benchmark

Execution Time Comparison - Immediately, we observe a more compressed range of execution times when compared with the quicksort benchmark. The locked queue performs the best closely followed by the CL queue and the ABP queue, with all of them performing within ~ 5% of one another. The LIFO BWoS queue, however, performs quite poorly (40 – 45% slower) when compared to the former 3, even at various block sizes. This suggests that I/O operations introduce a performance bottleneck that is not well suited for the block-based design.

Work-Stealing Effectiveness - The total number of fibers stolen reveals dramatic differences in work distribution effectiveness. The mutex-locked queue achieved the highest number of stolen fibers (5, 256), significantly outperforming the CL queue (3, 903) and ABP queue (1, 979). All BWoS implementations show poor stealing performance, with block size 256 achieving only 256 stolen fibers. Particularly noteworthy is the ”total_io_fibers_stolen” metric, which tracks the stealing

of I/O-bound tasks specifically. The mutex-locked queue excels here with 714 I/O fibers stolen, followed by the CL queue with 291, while the ABP queue steals only 42 I/O fibers. This suggests that the mutex-locked queue's simple design may be advantageous for stealing I/O-bound work, perhaps due to reduced overhead during the relatively long I/O wait times.

Context Switching Patterns - Voluntary context switches (nvcsw) show a pattern inverse to performance. The best performers (mutex-locked and CL queues) have lower context switch counts, while the BWoS implementations trigger significantly more (2x-3x). This suggests that the BWoS implementations cause more frequent thread yielding, mainly due to blocking the owner or thief thread, which is the inherent characteristic of the block-based design. Non-voluntary context switches (nivcsw) are highest in the CL queue (310), potentially indicating higher CPU utilization that triggers preemptive scheduling.

CPU Bound vs I/O Bound - When we compare the patterns to the quicksort benchmark, we see that the ABP queue now exhibits meaningful work-stealing, even though it is significantly lower than other queues. This points to the likelihood that natural pauses during system calls are beneficial for the queue reset procedure in the ABP queue. For LIFO BWoS, larger block sizes perform better than smaller block sizes in this instance. This might be because lock contention is less significant with I/O bound operations which are more coarse-grained tasks. The locked queue outperforms its lock-free counterparts, possibly because, again, lock contention is less significant when threads wait for I/O bound system calls.

5.4 Overall

After examining both the application benchmarks and microbenchmarks, several important patterns emerge that provide deeper insights into work-stealing queue behavior in different contexts. The main conclusion that can be drawn is that work-stealing queue performance is highly context-dependent, with significant variations across different workloads and operating conditions. This means that performance is far more nuanced than often portrayed in theoretical analyses. The stark contrasts between microbenchmark and application benchmark results highlight the danger of optimization based solely on isolated operations.

The CL queue's consistent performance across diverse scenarios makes it a strong default choice for general-purpose work-stealing systems. The BWoS queue can achieve excellent performance when properly tuned i.e. it is extreme sensitivity to configuration parameters which makes it challenging to deploy it in systems with varied or unpredictable workloads. The formal verification benefits must be weighed against this tuning complexity. Additionally, the dramatic difference between application and microbenchmark performance for the locked queue highlights the importance of realistic workload testing. The locked queue's simplicity appears to provide advantages in complex application scenarios where factors beyond raw queue operations—such as work distribution quality and reduced algorithm complexity—become significant.

These findings reinforce the importance of empirical evaluation using realistic workloads when selecting and configuring work-stealing queue implementations. The ideal choice depends not only on theoretical algorithmic properties but also on the complex interactions between queue behavior, workload characteristics, and hardware capabilities.

6 Conclusion and Future Work

This section summarizes the results of our research and outlines some possible directions for further work.

6.0.1 Conclusion

The purpose of this research was twofold. **First**, to reverse engineer the Rust implementation of the FIFO BWoS queue and create a robust C++ implementation, while also extending it with a LIFO BWoS variant based solely on the conceptual design details provided by Wang *et al.* in their 2023 paper *"BWoS: Formally Verified Block-based Work Stealing for Parallel Processing"*. This required significant design interpretation and adaptation, as the original work provided only theoretical foundations without implementation specifics for the LIFO variant. To the best of our knowledge, this work is the only open-source and publicly available (LGPL-3.0-or-later) implementation of the BWoS queue that includes both FIFO and LIFO variants. It is available in the public repository for *EMPER* [Flo25], specifically in the `WsBwosQueue.hpp` file under `emper/lib/adt/`. **Second**, to analyze the performance of the BWoS queue using both synthetic and real-world workloads, and to compare its performance against established work-stealing implementations, namely, the ABP queue (Arora, Blumofe, and Plaxton), the CL queue (Chase and Lev), and a locked queue used as a baseline.

The original plan was to benchmark both the FIFO and LIFO variants of the BWoS queue. However, during implementation (section 3.2), we discovered a pathological condition inherent to the design of the FIFO variant that can lead to a deadlock, requiring manual intervention to resolve. A potential solution to this issue involves introducing batched queue operations, which would prevent the specific block metadata state that causes the deadlock. Nevertheless, to ensure fairness in benchmarking, particularly because the ABP and CL queues do not support batched operations—we chose to exclude the FIFO BWoS variant from our performance analysis and instead focused solely on the LIFO variant, which does not exhibit this deadlock behavior.

The results from the research can be summarized into a few main points.

- **Tradeoffs of the block-based approach** - In theory, the block-based approach has many benefits. By using block-level metadata instead of global `top` and `bottom` counters, it is possible to effectively split the queue into two separate queues with blocks that belong to the owner and blocks that belong to the thief. This ensures that owners and thieves do not need to synchronize within blocks. Furthermore, all the expensive atomic operations can be deligated to the block advancement procedure to make the fastpath extremely fast. However, in practice, performance gains from this approach are heavily dependent on tuning the queue based on available workloads in the system. As evident from the microbenchmark results, under the right

conditions, the BWoS queue can outperform classical work-stealing approaches like the CL and ABP queue handily. However, this comes at a big tradeoff, in that the block-based approach can and does prevent stealing. This can cause spikes in usage and energy consumption from cores that execute the owner specific operations and idling in cores that execute the thief specific operations. Additionally, when it comes to real applications with unpredictable workloads or I/O bound workloads for example, tuning the BWoS queue is not very straightforward. For fine grained workloads, if the block-sizes are reduced to facilitate better load balancing, then that in turn adds more overhead from constant block advancement procedures.

- **Locked queue viability** - The mutex-locked queue displayed a diverse range of performance that challenges conventional wisdom about concurrent data structures. While it performed poorly in isolated microbenchmarks (130% worse than alternatives), it excelled in complex real-world applications, delivering the best performance for I/O-bound workloads (538.4ms) and competitive results for CPU-bound tasks (456.5ms). This suggests that in practical scenarios, the advantages of its simple design, predictable behavior, and superior work distribution effectiveness (highest fiber stealing across benchmarks) outweigh the theoretical overhead of lock contention. The dramatic disconnect between its raw operation throughput and application-level performance highlights an important principle: queue efficiency is not solely determined by the cost of individual push/pop operations but also by how effectively the implementation distributes work across threads and integrates with the broader system architecture, particularly when I/O operations introduce natural pauses that reduce lock contention.
- **Versatility** - The CL queue was the most consistent performer across all our microbenchmark, and application benchmark runs. As discussed in the background section(2), the CL queue was proposed as an improvement on the ABP queue with its circular data structure fixing the inefficiencies of the original ABP queue. This means that the classical formula for work-stealing is still quite versatile when it comes to efficient synchronization and robust work distribution regardless of workload characteristics.

6.0.2 Future Work

Future research can explore a more comprehensive analysis of both the LIFO and FIFO variants of the BWoS queue by incorporating batched queue operations. This would allow the FIFO variant to avoid the pathological deadlock condition identified in this work, thereby enabling a fair comparison between the two designs under realistic workloads. While the LIFO variant demonstrates advantages such as fewer atomic operations and improved cache locality owing to its stack-like behavior that naturally favors temporal data reuse, the FIFO variant offers important fairness guarantees. These guarantees are particularly relevant in the context of system-level scheduling, where task starvation must be minimized and predictable execution ordering is often desirable. An in-depth comparison could explore not only throughput and latency under various workloads (e.g., compute-bound vs. I/O-bound tasks), but also fairness metrics such as task wait time variance and starvation frequency. Additionally, examining the impact of BWoS's design on energy efficiency and CPU core utilization under modern architecture may provide valuable insights.

Another promising direction is the formal verification of the C++ implementation. While Wang's original implementation was formally verified using the GenMC model checker [KV21], the correctness of the C++ translation remains informal. With formal verification, we could even confirm our running theory that reworking the round control procedure in the design detail of the FIFO variant whereby introducing stronger atomics for version updates could alleviate the need to compare

the metadata index to the block margin and thus prevent the deadlock condition from happening. Moreover, leveraging model checkers to reason about memory safety, data race freedom, and progress guarantees could further strengthen confidence in the implementation.

Finally, exploring alternative stealing policies—such as size-based policies, NUMA-aware policies using a locality aware work-stealing scheduler [CGG14], and probabilistic approaches—could enable more effective selection of the optimal strategy for each queue implementation, depending on workload characteristics. Of particular interest is the probabilistic stealing approach, which was reported by Wang *et al.* to yield up to 1.5x performance improvements in the BWoS queue. Applying this strategy to other work-stealing queue implementations could reveal similar gains. Future research could evaluate its effects on steal throughput, median steal latency, and overall load-balancing efficiency under varying workload conditions.

Acronyms

ABP Arora–Blumofe–Plaxton.

BWoS Block-Based Work Stealing.

CAS Compare And Swap.

CL Chase-Lev.

FIFO First In First Out.

GC Garbage Collection.

JVM Java Virtual Machine.

LIFO Last In First Out.

NUMA Non-Uniform Memory Access.

SIMD Single Instruction Multiple Data.

SMT Simultaneous Multithreading.

TBB Thread Building Blocks.

WMM Weak Memory Model.

WSL2 Windows Subsystem for Linux.

References

[ABP98] NS Arora, RD Blumofe, and CG Plaxton. Thread scheduling for multiprogrammed multiprocessors. In: *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA '98. Puerto Vallarta, Mexico: Association for Computing Machinery, 1998, pp. 119–129. https://doi.org/10.1145/277651.277678 (pages 8, 11, 12, 23).

[Axb25] J Axboe. liburing: io_uring Library for Linux. https://github.com/axboe/liburing. Accessed: 2025-05-11. 2025 (page 45).

[BL99] RD Blumofe and CE Leiserson. Scheduling multithreaded computations by work stealing. In: *J. ACM* 46(5):(Sept. 1999), 720–748. https://doi.org/10.1145/324133.324234 (page 6).

[BS81] FW Burton and MR Sleep. Executing functional programs on a virtual tree of processors. In: *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*. FPCA '81. Portsmouth, New Hampshire, USA: Association for Computing Machinery, 1981, pp. 187–194. https://doi.org/10.1145/800223.806778 (page 5).

[CL05] D Chase and Y Lev. Dynamic circular work-stealing deque. In: *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '05. Las Vegas, Nevada, USA: Association for Computing Machinery, 2005, pp. 21–28. https://doi.org/10.1145/1073970.1073974 (pages 9, 11, 13).

[CGG14] Q Chen, M Guo, and H Guan. LAWS: locality-aware work-stealing for multi-socket multi-core architectures. In: *Proceedings of the 28th ACM International Conference on Supercomputing*. ICS '14. Munich, Germany: Association for Computing Machinery, 2014, pp. 3–12. https://doi.org/10.1145/2597652.2597665 (page 51).

[Cor+09] TH Cormen et al. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009 (pages 4, 5).

[cpp24] cppreference.com. std::memory_order - cppreference.com. Accessed: 2025-05-02. 2024. https://en.cppreference.com/w/cpp/atomic/memory_order (page 8).

[Det+04] D Detlefs et al. Garbage-first garbage collection. In: *Proceedings of the 4th international symposium on Memory management*. 2004, pp. 37–48 (page 2).

[DB82] M Dubois and FA Briggs. Effects of cache coherency in multiprocessors. In: *SIGARCH Comput. Archit. News* 10(3):(Apr. 1982), 299–308. https://doi.org/10.1145/1067649.801739 (pages 6, 28).

[Flo25] FF Florian Schmaus. *Emper: The Efficient Massive-Parallelism Execution Realm (EMPER) is a concurrency platform to develop and execute parallel applications.* Accessed: 2025-04-28. 2025. https://gitlab.com/flowpack/manycore/emper (pages 2, 27, 49).

[FLR98] M Frigo, CE Leiserson, and KH Randall. The implementation of the Cilk-5 multithreaded language. In: *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*. PLDI '98. Montreal, Quebec, Canada: Association for Computing Machinery, 1998, pp. 212–223. https://doi.org/10.1145/277650.277725 (pages 2, 3).

[Hal84] RH Halstead. Implementation of multilisp: Lisp on a multiprocessor. In: *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*. LFP '84. Austin, Texas, USA: Association for Computing Machinery, 1984, pp. 9–17. https://doi.org/10.1145/800055.802017 (page 5).

[Has16] W Hassanein. Understanding and improving JVM GC work stealing at the data center scale. In: June 2016, pp. 46–54 (page 8).

[Hel23] M Heller. *What is garbage collection? Automated memory management for your programs*. Accessed: 2023-02-03. Feb. 2023. https://www.infoworld.com/article/2337816/what-is-garbage-

collection-automated-memory-management-for-your-programs.html (pages 2, 7).

[Hor+18] M Horie et al. Balanced double queues for GC work-stealing on weak memory models. In: *SIGPLAN Not.* 53(5):(June 2018), 109–119. https://doi.org/10.1145/3299706.3210570 (pages 2, 8).

[Hor+19] M Horie et al. Scaling up parallel GC work-stealing in many-core environments. In: *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management*. ISMM 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 27–40. https://doi.org/10.1145/3315573.3329985 (page 15).

[Koh25] L Kohn. *Evaluation and Analysis of the Interaction Between Concurrency Platform and Operating System*. Referenced pages: 19, 23–26. 2025 (page 45).

[KV21] M Kokologiannakis and V Vafeiadis. GenMC: A Model Checker for Weak Memory Models. In: *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part I*. Berlin, Heidelberg: Springer-Verlag, 2021, pp. 427–440. https://doi.org/10.1007/978-3-030-81685-8_20 (page 50).

[KS14] S Kumar and A 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*. 2014, pp. 28–35 (pages 10, 15).

[Kum20] V Kumar. PufferFish: NUMA-Aware Work-stealing Library using Elastic Tasks. In: *2020 IEEE 27th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. 2020, pp. 251–260 (page 15).

[Lê+13] NM Lê et al. Correct and efficient work-stealing for weak memory models. In: *ACM SIGPLAN Notices* 48(8):(2013), 69–80 (page 9).

[Liu+14] C Liu et al. 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)*. 2014, pp. 149–152 (page 15).

[McC60] J McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. In: *Communications of the ACM* 3(4):(Apr. 1960). Reprinted at the author's website, 184–195. https://www-formal.stanford.edu/jmc/recursive.html (page 7).

[MRR12] M McCool, J Reinders, and A Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012 (page 4).

[Mic23] Microsoft Docs. *Fundamentals of garbage collection*. Accessed: 2023-02-03. Feb. 2023. https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals (page 7).

[Mit01] M Mitzenmacher. The power of two choices in randomized load balancing. In: *IEEE Transactions on Parallel and Distributed Systems* 12(10):(2001), 1094–1104 (page 15).

[Mos93] D Mosberger. Memory consistency models. In: *ACM SIGOPS Operating Systems Review* 27(1):(1993), 18–26 (page 8).

[Ope25] OpenJDK Contributors. OpenJDK JDK: Java Development Kit Main-Line Development. Accessed: 2025-05-10. 2025. https://github.com/openjdk/jdk (page 8).

[Pro21] Protty. Resource Efficient Thread Pools with Zig. Accessed: 2025-05-09. Sept. 2021. https://zig.news/kprotty/resource-efficient-thread-pools-with-zig-3291 (page 43).

[Sat+12] N Satish et al. Can traditional programming bridge the ninja performance gap for parallel computing applications? In: *ACM SIGARCH Computer Architecture News* 40(3):(2012), 440–451 (page 1).

[Sch+21a] F Schmaus et al. *Modern concurrency platforms require modern system-call techniques*. Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), 2021 (page 1).

[Sch+21b] F Schmaus et al. Nowa: A Wait-Free Continuation-Stealing Concurrency Platform. In: *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2021, pp. 360–371 (pages 2, 3, 5, 7, 11, 13, 35).

[SMR09] MJ Sottile, TG Mattson, and CE Rasmussen. *Introduction to Concurrency in Programming Languages*. Boca Raton, FL: Chapman and Hall/CRC, 2009. https://www.taylorfrancis.com/books/mono/10.1201/b17174/introduction-concurrency-programming-languages-matthew-sottile-timothy-mattson-craig-rasmussen (page 4).

[SLS16] W Suksompong, CE Leiserson, and TB Schardl. On the efficiency of localized work stealing. In: *Information Processing Letters* 116(2):(2016), 100–106. https://www.sciencedirect.com/science/article/pii/S0020019015001726 (page 15).

[Sut05] H Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. In: *Dr. Dobb's Journal* 30(3):(2005), 202–210. http://www.gotw.ca/publications/concurrency-ddj.htm (page 3).

[07] The Foundations for Scalable Multi-Core Software in Intel ® Threading Building Blocks. In: 2007. https://api.semanticscholar.org/CorpusID:263027563 (pages 2, 3).

[The25] The Go Programming Language. The Go Programming Language Official Website. Accessed: 2025-03-27. 2025. https://go.dev/ (page 15).

[Tok25a] Tokio Developers. Tokio: Asynchronous Runtime for Rust. Accessed: 2025-03-27. 2025. https://github.com/tokio-rs/tokio (page 15).

[Tok25b] Tokio Developers, Jonathan Schwender. BWoS FIFO implementation in Tokio: Asynchronous Runtime for Rust. Commit: 60d9ed76b35f504e2423213586f40ef49fbf3d8e. Accessed: 2025-03-27. 2025. https://github.com/tokio-rs/tokio/blob/60d9ed76b35f504e2423213586f40ef49fbf3d8e/bwosqueue/tests/blocked_stealer.rs#L2 (page 21).

[Wan+23] J Wang et al. BWoS: Formally Verified Block-based Work Stealing for Parallel Processing. In: *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. Boston, MA: USENIX Association, July 2023, pp. 833–850. https://www.usenix.org/conference/osdi23/presentation/wang-jiawei (pages 14, 15, 18, 25, 39).

[Wei19] E Weisman. Making the Tokio Scheduler 10x Faster. https://tokio.rs/blog/2019-10-scheduler. Accessed: 2025-05-11. Oct. 2019. https://tokio.rs/blog/2019-10-scheduler (page 1).

[Wik25a] Wikipedia contributors. HotSpot (virtual machine) — Wikipedia, The Free Encyclopedia. [Online; accessed 12-May-2025]. 2025. https://en.wikipedia.org/w/index.php?title=HotSpot_(virtual_machine)&oldid=1283651557 (page 8).

[Wik25b] Wikipedia contributors. Single instruction, multiple data — Wikipedia, The Free Encyclopedia. [Online; accessed 12-May-2025]. 2025. https://en.wikipedia.org/w/index.php?title=Single_instruction,_multiple_data&oldid=1287294728 (page 4).

[X-b14] X-bit labs. Intel Pentium 4 3.06GHz CPU with Hyper-Threading Technology: Killing Two Birds with a Stone. Archived article. Archived on 31 May 2014, Retrieved 4 June 2014. 2014. https://web.archive.org/web/20140531105602/http://www.xbitlabs.com/articles/cpu/display/pentium4-3066.html (page 3).

Declaration of Authorship

Ich erkläre hiermit gemäß § 9 Abs. 12 APO, dass ich die vorstehende Mastersarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Des Weiteren erkläre ich, dass die digitale Fassung der gedruckten Ausfertigung der Mastersarbeit ausnahmslos in Inhalt und Wortlaut entspricht und zur Kenntnis genommen wurde, dass diese digitale Fassung einer durch Software unterstützten, anonymisierten Prüfung auf Plagiate unterzogen werden kann.

Bamberg, den _____ _____

Shibesh Duwadi