BWoS: Formally Verified Block-based Work Stealing for Parallel Processing

Jiawei Wang, *Huawei Dresden Research Center, Huawei Central Software Institute, Technische Universität Dresden;* Bohdan Trach, Ming Fu, Diogo Behrens, Jonathan Schwender, Yutao Liu, and Jitang Lei, *Huawei Dresden Research Center, Huawei Central Software Institute;* Viktor Vafeiadis, *MPI-SWS;* Hermann Härtig, *Technische Universität Dresden;* Haibo Chen, *Huawei Central Software Institute, Shanghai Jiao Tong University*

https://www.usenix.org/conference/osdi23/presentation/wang-jiawei

This paper is included in the Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-34-2

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

BWoS: Formally Verified Block-based Work Stealing for Parallel Processing

Jiawei Wang[1,2,3], Bohdan Trach[1,2], Ming Fu[1,2,*], Diogo Behrens[1,2], Jonathan Schwender[1,2], Yutao Liu[1,2], Jitang Lei[1,2], Viktor Vafeiadis[4], Hermann Härtig[3], and Haibo Chen[2,5]

[1]*Huawei Dresden Research Center* [2]*Huawei Central Software Institute*
[3]*Technische Universität Dresden* [4]*Max Planck Institute for Software Systems*
[5]*Shanghai Jiao Tong University*

Abstract

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

We present Block-based Work Stealing (BWoS), a novel and pragmatic design that splits per-core queues into multiple blocks. Thieves and owners rarely operate on the same blocks, greatly removing interferences and enabling aggressive optimizations on the owner's synchronization with thieves. Furthermore, BWoS enables a novel probabilistic stealing policy that guarantees thieves steal from longer queues with higher probability. In our evaluation, using BWoS improves performance by up to 1.25x in the Renaissance macrobenchmark when applied to Java G1GC, provides an average 1.26x speedup in JSON processing when applied to Go runtime, and improves maximum throughput of Hyper HTTP server by 1.12x when applied to Rust Tokio runtime. In microbenchmarks, it provides 8-11x better performance than state-of-the-art designs. We have formally verified and optimized BWoS on weak memory models with a model-checking-based framework.

1 Introduction

Many language runtimes and similar systems (*e.g.*, JVM [104], Go [36], Rust's Tokio [38]) divide their work into smaller units called *tasks*, which are executed asynchronously on multiple cores and whose execution can generate further tasks. To achieve good performance, the task scheduler has to ensure a good *workload distribution* (preventing idle cores while there are pending tasks) with a low scheduling *overhead*.

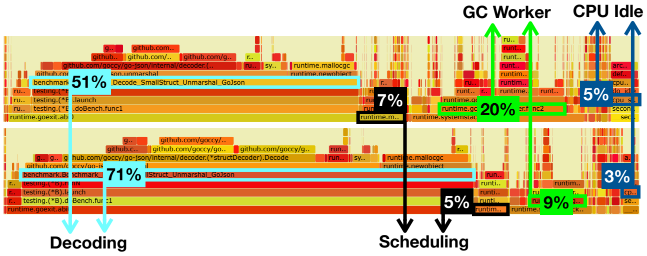


Figure 1: Profiling results for go-json complex object decoding (~1μs/op) benchmark [9], with the original work stealing queue (up) and with BWoS *(down).*

Achieving these goals, however, is non-trivial. Storing the tasks in a single queue shared by all cores achieves optimal workload distribution, but incurs a huge overhead due to contention. Using per-core task queues minimizes the overhead per operation, but can easily lead to a skewed workload distribution, with some cores remaining idle while others have queued work.

Work stealing [51] is a trade-off between these two extremes: each core owns a queue (*owner*) and acts as both the *producer* and the *consumer* of its own queue to put and get tasks. When a core completes its tasks and the queue is empty, it then steals another task from the queue of another processing core to avoid idling (*thief*). A number of stealing policies [69, 76, 77, 83, 88, 100] have been proposed to choose the proper queue (*victim*) to steal from, which can bring significant speedups depending on the use-cases. Due to these features, work stealing is widely used in parallel computing [22, 35, 56, 64, 65, 85, 93, 97], parallel garbage collection [60, 68, 69, 96, 101], GPU environment [52, 54, 99, 102, 103], programming language runtimes [26, 36, 38, 50, 63, 80, 81], networking [86] and real-time systems [82].

However, as parallel processing is applied to more workloads, current implementations of work stealing become a bottleneck, especially for small tasks. For example, web frame-

[*] Ming Fu (ming.fu@huawei.com) is the corresponding author.

works running over lightweight threading abstractions, such as Rust's Tokio and Go's goroutines, often contain many very small tasks, leading the Tokio runtime authors to observe that "the overhead from contending on the queue becomes significant" and even affects the end-to-end performance [28]. Similarly, high-performant garbage collectors, such as Java G1GC [13], rely on work stealing for parallelizing massive mark/sweep operations, which comprise only a few instructions. The work stealing overhead becomes a performance bottleneck for GC [68, 69, 96, 101].

As a third example, in Fig. 1, we profile the GoJson object decoding benchmark, which uses goroutines both for GC workers and for parsing complex objects. Only 51% of all CPU cycles constitute the useful workload (JSON decoding). The remaining cycles are spent on the runtime code, including 7% on lightweight thread scheduling, 20% on GC, 5% on kernel code idling the CPUs, *etc*. As both the scheduler and the GC code rely on work stealing, improving its performance can result in massive efficiency gains. Furthermore, the benefit is not limited to the above-mentioned scenarios, but expands to all fine-grained tasks parallel processing scenarios. Thus, we ask the following question: *How can we improve performance of work-stealing queues for fine-grained tasks to the benefit of a large range of common applications*?

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

P1: Synchronization overhead. Due to the possibility of a steal, local queues must use stronger atomic primitives (*e.g.*, atomic compare-and-swap and memory barriers) than a purely sequential queue. Queues with a FIFO policy are generally implemented as single-producer multiple-consumer (SPMC) queues [8, 17, 39, 47], thereby treating *steal* similarly to *get*, and thus distributing the costs of stealing equally between owner and thieves. This also applies to the existing block-based queues [106], which lack any optimizations specific to the work-stealing use case to achieve high performance in the presence of thieves (§6.2, §7). Queues with a LIFO policy, such as the well-known and widely-used ABP queue [35, 48, 104], suffer from memory barrier overhead [83, 98] to avoid the conflict between the owner and thieves, even when they operate on different tasks.

P2: Thief-induced cache misses. Since steals update the metadata shared between the owner and the thieves, they cause cache misses on subsequent accesses to the queue by its owner. This problem is especially apparent on unbalanced workloads, which feature high steal rates—for example, in the JVM Renaissance benchmarks [95], 10% of all items are stolen on average. Although strategies such as batching (*e.g.*, *steal-half* [66]) can reduce the frequency of steals, they often cause *overstealing* which introduces additional overhead (§2.1.3).

P3: Victim selection. To improve the workload distribution, advanced policies for selecting the victim queue to steal from require scanning the metadata of several queues, *e.g.*, to find the longest queue. Doing so, however, causes contention for its owner and severely limits the improvement from advanced victim selection policies (§2.1.3).

P4: Correctness under weak memory models (WMMs). Correctly implementing concurrent work-stealing queues on weak memory architectures, such as Arm servers for datacenters [46, 70], is very challenging because it requires additional memory barriers to prevent unwanted reordering. Using redundant barriers can greatly reduce the performance of work-stealing [79], while not including enough barriers can lead to errors, such as in the C11 [6] version of the popular unbounded Chase-Lev deque translated from formally verified ARMv7 assembly [90]. Even the popular Rust Tokio runtime required a fix to its implementation of work stealing [2].

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

First, we split each per-core queue into multiple blocks with independent metadata and arrange for the owner and the thieves to synchronize at the block level. Therefore, in the common case where operations remain within a block, we can elide synchronization operations and achieve almost single-threaded performance (§3.2). Similarly, since a queue owner and the thieves share only block-local metadata, they do not interfere when operating on different blocks (§2.1). We can arrange for that to happen frequently by allowing stealing tasks from the middle of the queue.

Second, we improve victim selection with a *probabilistic policy*, which approximates selecting the longest queue (§3.4), while avoiding the severe interference typical of the prior state-of-the-art (§2.1), to which we can integrate NUMA-awareness and batching.

Finally, we ensure correctness under WMMs by verifying BWoS with the GenMC model checker [74, 75] and optimizing its choice of barriers with the VSync toolchain [92] (§5).

As a result, BWoS offers huge performance improvements over the state-of-the-art (§6). In microbenchmarks, BWoS achieves up to 8-11x throughput over other algorithms. In representative real-world macrobenchmarks, BWoS improves performance of Java industrial applications by up to 25% when applied to Java G1GC, increases throughput by 12.3% with 6.74% lower latency and 60.9% lower CPU utilization for Rust Hyper HTTP server when applied to the Tokio runtime, and speeds up JSON processing by 25.8% on average across 9 different libraries when applied to the Go runtime.

Returning to our motivating example (Fig. 1), applying BWoS to the Go runtime removes 29% of scheduling time, 55% of GC time, and 40% of CPU idle time, while increasing the CPU time ratio for useful work from 51% to 71%.

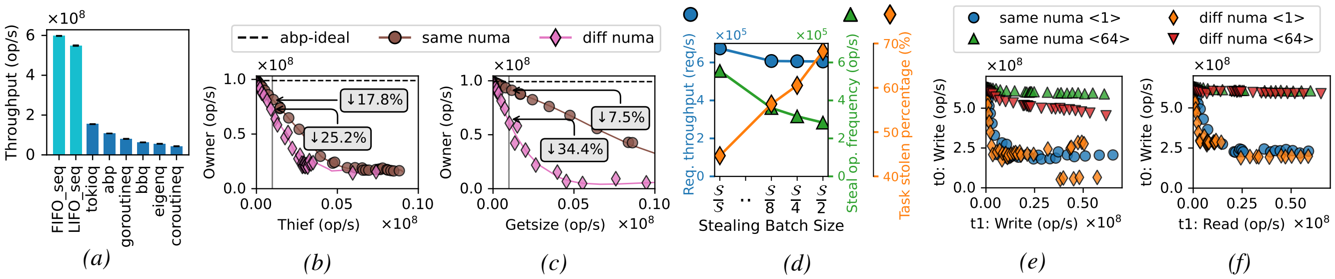


Figure 2: Motivating benchmarks: (a) Sequential performance of state-of-the-art work stealing algorithms. (b,c) Performance of the ABP queue owner depending on the frequency of (b) steal and (c) getsize operations. (d) Hyper HTTP server performance with different stealing batch sizes with the original Tokio work stealing queue: S is the victim queue size and S/2 refers to the default steal half policy [66]. (e,f) Interference between two threads for two sizes of cacheline sets.

2 Background

Task processing. Tasks vary a lot among benchmarks. Their processing time ranges from a few nanoseconds (*e.g.*, Java G1GC [13]), to microseconds (*e.g.*, RPC [55, 73, 108]), and even to seconds (*e.g.*, HPC tasks [35]). In this paper, we mainly focus on the nanosecond- and microsecond-scale tasks. Ignoring steals, tasks may be processed either:

- in FIFO (first-in-first-out) order, when minimizing processing latency is important (*e.g.*, network connections), or
- in LIFO (last-in-first-out) order, when only the overall execution time matters, as is often the case with multithreaded fork-join programs [57].

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

Victim selection. There are multiple policies for selecting the victim queue to steal from. *Random* [51] chooses one of the remaining queues uniformly at random: it has the least complexity but achieves poor load balancing. Size-based policies (*e.g.*, *best of two* [88] and *best of many* [69]) scan the queues' size to improve the load balance by stealing from a large queue. The NUMA-aware policy [77] was proposed to optimize the remote communication cost, by tending to steal from the queues in the local cache domain. Batch-based policies (*e.g.*, *steal half* [66] is used in Go and Rust's Tokio runtimes) allow thieves to steal multiple tasks at once to reduce their interference with the owner. Later in this section (§2.1.3), we will quantify these overhead sources to guide our queue design.

2.1 Performance Overhead Breakdown

Next, we analyze the state-of-the-art work stealing algorithms to dissect their performance issues, and motivate the design decisions of BWoS. Fig. 2 contains our experimental results on an x86 server [71].

2.1.1 Cost of Synchronization Operations

As steals may happen at any time, strong atomic primitives are introduced for local queue manipulation. To quantify their cost, we first measure the throughput of the state-of-the-art work stealing algorithms on a sequential setup where an owner puts and gets data from its local queue, without any tasks ever being stolen (§6.2). We compare the results with the theoretical performance upper bound: a single-threaded FIFO (FIFO_seq) or LIFO (LIFO_seq) queue implementation [72] without support for steals. Although there is no owner-thief interference, these synchronization operations pose a huge overhead (Fig. 2a): throughput of these work stealing algorithms is less than 0.25x for FIFO-based (0.19x for LIFO-based) compared to the upper bound.

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

Table 1: Reducing the stealing overhead with a NUMA-aware policy.

2.1.2 Interference Cost with Thieves

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

As we can see in Fig. 2b, the thief significantly degrades the owner's throughput: *e.g.*, by stealing only 1% of the tasks, the owner's throughput drops by 17.8% when the thief is in the same NUMA node, and by 25.2% when it runs in a different NUMA node. This degradation happens because of the cache interference between the owner and the thief on the shared metadata. We will further explain this in §2.2.

2.1.3 Overhead due to Victim Selection

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

Size-based policies. Policies like *best of two* [88] or *best of many* [69] read global metadata of multiple queues (their length) to determine the victim. Somewhat surprisingly, as shown in Fig. 2c, these reads introduce significant overhead for the owner, especially in the cross-NUMA scenario: even with a *getsize* frequency of only 1%, the owner throughput drops by 34.4%. This is further amplified as *getsize* is called multiple times for a single steal.

Therefore, for size-based policies, although reading more queues' sizes (*e.g.*, *best of many* [69]) can achieve better load balance, it inevitably induces more slowdown to the owners of these queues (§6).

NUMA-aware policies. NUMA-aware policies [77] try to reduce the overhead of each steal by prioritizing the stealing from queues in the same NUMA node. We observe that although such NUMA-aware policies can reduce the overhead of steals by 56% in the case of our ABP queue benchmark, they fail to achieve their full potential.

In Table 1, we break down the overhead of stealing in the ABP queue into its two main parts: the thief's communication cost and the owner's interference penalty. The former is 141ns when the thief and owner run on different NUMA nodes (measured by Intel MCA [21]), and reduces to 15ns (consistent with the L3 cache access latency [7]) when they are at the same NUMA node. The victim's interference penalty is 170ns and 278ns for cases of thief and victim running on the same (I_s) and different (I_d) NUMA domains respectively. NUMA-aware policies with existing queues can typically eliminate the first communication overhead, while leaving the second interference overhead not sufficiently optimized.

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

Batch-based policies. Batch-based policies steal more tasks at once with the aim of reducing the frequency of steals. Indeed, in the Hyper HTTP server benchmark (see Fig. 2d), choosing larger batch sizes leads to a reduction in the number of steal operations. These larger steal operations, however, make the workload even less balanced (*i.e.* percentage of stolen tasks increases), which results in additional overhead (*e.g.*, task ping-pong), canceling out the overhead reduction due to the fewer steals: the end-to-end throughput remains roughly the same.

2.2 Recap to Motivate BWoS

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

To better understand the effects of these types of cache contention, we conduct a simple microbenchmark with two threads: thread t_0 continuously writes to a cacheline, while thread t_1 either reads or writes to a cacheline with a specified frequency (Figs. 2e and 2f). The cachelines for t_0 and t_1 are independently and randomly chosen on each iteration out of the cacheline sets of two sizes: 1 or 64.

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

3 Design

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

The structure of BWoS queue facilitates abstracting the operations into *block advancement* that works across blocks, and *fast path* that operates inside of the block chosen by the block advancement (§3.1). Moving most of the synchronization from the fast path to the block advancement allows BWoS to fully reap the performance benefit indicated by our previous observation (§2.1.1) thus approaching the theoretical upper bound. *get* and *steal* always happen on different blocks. We carefully construct the algorithm such that thieves cannot obstruct the progress of *get*, while *get* can safely *takeover* a block from thieves operating on it without waiting for them. For complexity consideration, we don't prohibit *put* and *steal* in the same block[1], as they can synchronize with the weak barriers without losing performance (§6.2).

As metadata is also split per block, thieves and the owner are likely to operate on different blocks and thus update different metadata. As explained in §2.2, this reduces the interference between thieves and the owner. For FIFO-based BWoS, block-local metadata allows stealing from the middle of the queue, without enforcing the SPMC queue restriction of always stealing the oldest task, which is not required by the workloads.

BWoS can benefit from NUMA-aware policies more than other queues because the reduction in interference for the victim makes both constituents of cross-NUMA-domain stealing overhead negligible (Table 1). Furthermore, unlike batch-based policies, stealing policies integrated with BWoS can focus on balancing the workload itself without worrying about the interference from frequently called *steal*.

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

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

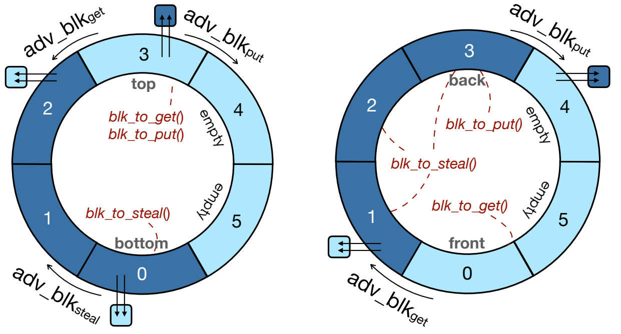
Figure 3: Pseudocode of put, get, and steal operations.

3.1 Bird's-Eye View of the Queue

To better understand the block-based approach, let's consider the *put*, *get*, and *steal* operations of the BWoS queue (Fig. 3). For each of these operations, the first step is to select a block to work on (lines 3, 14, and 27). The owner uses the *top* block for put and get for the LIFO BWoS, and gets from the *front* block and puts to the *back* block for the FIFO BWoS. In this case, top, back, and front block pointers are owner-exclusive metadata which is unavailable to the thieves. For *steal*, the choice of the block is more complicated and we will explain it in a later section (§3.4).

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

Splitting the global metadata into block-level instances enables splitting the operations into the fast path and block advancement, which increases the performance by keeping the fast path extremely lightweight. However, the lack of global mutable metadata shared between owner and thieves raises additional challenges, which are mostly delegated to the block advancement—it is now responsible for maintaining complex block-level invariants. We introduce the following invariants:
(1) *put* never overwrites unconsumed data;
(2) *steal* and *get* never read the same data;
(3) *steal* and *get* never read data that has been read before;
(4) *steal* in progress cannot prevent *get* from reading from a thieves' block. Before explaining fast path and block advancement implementations, we introduce two key concepts we rely on to ensure that the abovementioned invariants hold: *block-level synchronization* (§3.2) and *round control* (§3.3).



(a) LIFO BWoS *(b) FIFO BWoS*

Figure 4: Block-level synchronization in BWoS.

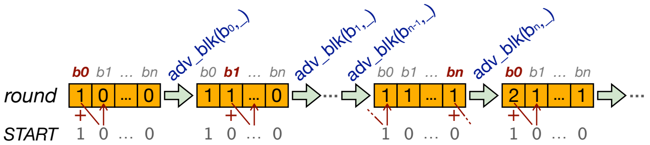


Figure 5: Update of round numbers in each block.

3.2 Block-level Synchronization

Block-level synchronization is the key responsibility of the block advancement and ensures that thieves never steal from the block currently used for get operations. Each block is owned either by the owner or by the thieves. For example, in Fig.4, blocks with lighter and darker colors belong to the owner and thieves respectively. The owner grants a block to the thieves, or takes a block back from them with block advancement. More specifically, for LIFO BWoS, *get* advances to the *preceding* block (3 to 2) and takes it over from thieves; *put* grants the current one and advances to the *following* block (3 to 4). For FIFO BWoS, *get* (resp. *put*) advances and takes over (resp. grants) the following block.

The grant and takeover procedures are based on the *thief index*—an entry in the block metadata that indicates the stealing location inside the block. Takeover sets this index to the block margin with an atomic exchange, and uses the old value as the threshold between the owner and the ongoing thieves in this block. This ensures that owner is not blocked by thieves when it takes over the block. Moreover, concurrent owner and thieves never read the same data because the threshold between them is set atomically. Similarly, the grant procedure transfers the block to thieves by writing the threshold to the thief index. We will introduce the details in §4.2.1.

3.3 Round Control

Each block also records *round numbers* of the last data access. When advancing block, the current block's round is copied over to the next block; except in the case of a wrap-around, where the block number is increased by 1 (Fig. 5).

In fact, there are producer, consumer, and thief round numbers in each block. When the producer tries to write round r's data into a block, the consumer and thieves must have finished reading all data with round $r-1$ from that block; so that the producer never overwrites any unread data. Similarly, when the consumer or a thief tries to read round r's data from a block, the producer's round at that block must already be r; this prevents reading any data twice, or reading data that was never written. Details can be found in §4.2.2.

3.4 Probabilistic Stealing

As discussed in §2.1.3, size-based policies can achieve better load balance at the cost of degrading the performance of the owner of each queue. Calculating the size is even harder in our setting because the appropriate metadata is distributed across all blocks. However, BWoS brings an opportunity to have a new size-based, probabilistic stealing policy, which can provide strong load balance without adversely affecting the owner's performance.

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

Therefore, given a pool of N queues each with the same capacity and a selector in P_{select} that selects each queue with equal probability, P_{accept} can guarantee that the probability of a thief stealing from a queue is proportional to its size.

To minimize the impact on the owner's performance, instead of measuring $\mathcal{S}$, we estimate $\mathcal{S}/\mathcal{C}$ directly by sampling. The thief chooses a random block from *all* blocks of the queue and checks if it has data *available for stealing*, where the probability of returning true is close to $\mathcal{S}/\mathcal{C}$. As the thief reads only one block's metadata, its interference with the owner is minimal (cf. §2.2).

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

For LIFO BWoS, stealing still happens from the *bottom* block (Fig. 4). Thieves advance to the following block when they finish the current one. For FIFO BWoS, thieves do not advance block when randomized stealing is enabled, and fall back to the stealing policy for selection of the new queue and block instead (§3.4). In this case, the operation to advance to the next block on stealing (Fig. 3 line 36) becomes a no-op.

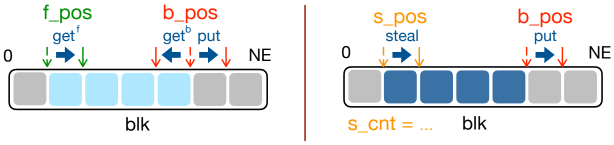


Figure 6: Put, get, and steal operations inside the block.

Moreover, we can further combine the probabilistic stealing policy with a variety of selectors for P_{select} phase (*e.g.*, from NUMA-aware policy), to benefit from both better workload balance and reduced stealing cost. Results show that the hybrid probabilistic NUMA-aware policy brings the best performance to BWoS (§6).

4 Implementation

4.1 Single-Block Operations (Fast Path)

Let's consider how *put*, *get*, and *steal* operations inside the block are implemented (lines 4, 15, and 28 in Fig. 3). Because *get* and *steal* always happen on different blocks, we only need to consider two cases of multiple operations in a block: producer-consumer and producer-thieves (Fig 6).

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

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

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

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

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

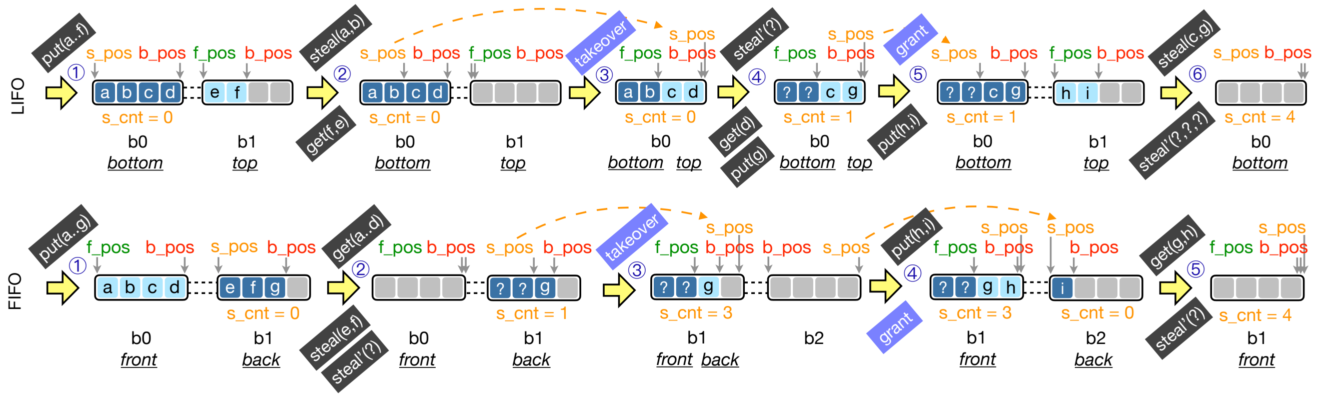


Figure 7: Takeover and grant procedures in block advancement.

4.2 Block Advancement

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

4.2.1 Takeover and Grant Procedures

We explain the takeover and grant procedures using a queue with 4-entry blocks as an example (Fig. 7).

LIFO. Let us assume that 6 elements (a-f) were put into the queue. Thus, the owner is in the block b_1; b_pos in b_0 and b_1 becomes 4 and 2 respectively, while f_pos and s_pos remain at the initial value (0) (state ①). Then, two actions happen concurrently: two thieves try to steal entries, updating s_pos in b_0 to 2, and start to copy out the data (steal on Fig. 7), while the owner gets 3 values, consuming f, e (state ②), and advancing to b_0, thus starting the takeover. To perform the takeover, the owner atomically exchanges s_pos with the block margin (4), and then sets f_pos to the previous s_pos value (2) (state ③). After the takeover, the owner gets d and puts g. Meanwhile, one ongoing *steal* completes (steal' on Fig. 7), increasing s_cnt by 1 (state ④). It does not matter which of the two completes first. When the owner puts new items h and i, it grants b_0 to thieves and advances to b_1. To perform the grant, it sets s_pos to the f_pos value (2), indicating to thieves that the block is available (state ⑤). After thieves steal all entries in b_0, s_cnt reaches the block margin (state ⑥). Thus, b_0 can be reused in the next round.

FIFO. First, the producer puts 7 elements (a-g) into the queue. The producer and the consumer are in b_1 and b_0 respectively, and thieves can steal from b_1 (state ①). Then, the consumer gets all elements in b_0, and advances to b_1 (state ②). This requires taking over b_1 from thieves: for this purpose, it updates s_pos and f_pos in the same way as the LIFO BWoS, but also adds the difference between the new f_pos (2) and the block margin (i.e. length of the block) to s_cnt (state ③). This way, when all thieves finish their operation in b_1, its s_cnt will be equal to the block margin. After that, the producer puts a new item h, and advances to b_2 granting it to thieves (state ④). Finally, both thieves and the consumer have read all entries from b_1, its f_pos and s_cnt are equal to the block margin (state ⑤). The producer uses this condition to check if the block can be reused for producing new values into it.

4.2.2 Round Control and Reset Procedure

To implement round control (§3.3), the position variables in block metadata (f_pos, b_pos, s_pos, s_cnt) contain both the index or counter (idx field) as described in §4.2.1 and the round number (rnd field). We fit both components into a 64-bit variable that can be updated atomically.

Consider, for example, the put operation of FIFO BWoS (Fig. 8). In *put*, when the producer idx reaches the block margin NE of the block blk (step ①), the new round x of the next block nblk is calculated as described in §3.3 (step ②). When advancing to the block nblk with the producer round x, the producer checks that the consumer and the thieves have finished reading all data from the previous round in nblk by checking if their idx fields are equal to NE and their rnd fields are x − 1 (step ③). When the check succeeds, the new value with the index 0 and the round x will be written into the producer position variable (step ④), thus *resetting* the block for the next round producing. Otherwise, a "queue full" condition is reported.

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

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

5 Verification and Optimization

The complexity of the BWoS algorithm necessitates the use of formal verification techniques to ensure that there are no lurking design or implementation bugs, and to optimize the

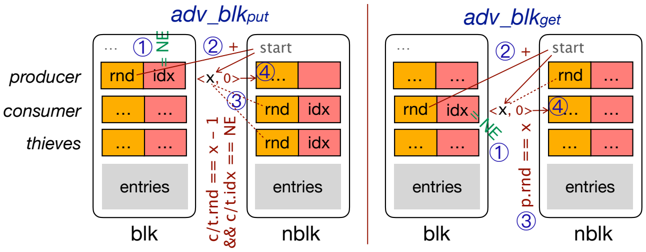


Figure 8: Round control in FIFO BWoS.

use on WMMs. One can easily imagine several tricky cases with block advancements. For example, for LIFO BWoS, when the owner calls *puts* and *gets* and advances to the next block, it may easily trigger ABA [67] bugs during the round control and takeover.

Unlike simpler algorithms like ABP [79], it is virtually impossible to justify the correctness of an optimal memory barrier placement by inspection. Luckily, model checking tools [62, 74, 84, 92] are widely used to check the correctness of concurrent algorithms and optimize the memory barrier under WMMs automatically, improving both performance and developer confidence. For example, the Tokio library uses the model checker Loom [27, 91], which has helped them find more than 10 bugs [28].

5.1 Verification Client

A model checker takes as input a small *verification client* program that invokes queue operations. It verifies that all possible executions of the input program satisfy some generic correctness properties, such as memory safety and termination [45], as well as any algorithm-specific properties that are included in the verification client as *assertions*. Whenever verification fails, the model checker returns a concrete erroneous execution as a counterexample.

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

Verified properties. We have verified the following properties with the GenMC model checker [74, 75]:

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

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

Figure 9: Verification and optimization client code.

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

Table 2: Statistics of the verification and optimization.

All possible executions, including those that occur due to weak memory reordering under the IMM [94] and RC11 [78] memory models, have been explored, and the aforementioned properties hold for each of them. With GenMC we were able to verify safety properties and termination of loops, but not the properties of individual operations.

Contending scenarios. As in any model checking verification, our models have a limited size within which the above properties hold. The client code for verifying and optimizing *put*, *get*, *steal* operations of BWoS is shown in Fig. 9. We configure the queue to have two blocks, each with the capacity of two entries (line 23). It is thus sufficient to put 5 entries to trigger the queue wraparound. We then launch 3 threads that run in parallel: The owner thread T0 has 3 rounds of *put* and *get* (lines 24-26) with different numbers of entries, trying to trigger block advancement for both producers and consumers in each round. Thief threads T1 and T2 steal one and two entries respectively, and thus together with T0, they trigger the queue empty condition, takeover, grant, and reset procedures, as well as conflicts between thieves.

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

5.2 Results

We have optimized and verified the C code of LIFO and FIFO BWoS with the VSync framework and the GenMC model checker. We have also verified the ABP queue using our verification client as a baseline. The statistics are shown in Table 2, broken down by memory barrier type: sequentially consistent (SEQ), acquire (ACQ), release (REL), and relaxed (RLX, i.e. plain memory accesses).

For BWoS, barrier optimization and verification finished in about an hour on a 6-core workstation [59], with over 1 million execution explorations. For ABP, the checking finishes in 16 minutes. More executions are explored for ABP since thieves and owner synchronize for every operation, which brings more interleaving cases.

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

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

Optimization. For BWoS, most concurrent accesses are converted to relaxed barriers, with the few remaining cases being release or acquire barriers. For the owner's fast path that determines the performance, we have only one release barrier in the FIFO BWoS. In contrast, the highly optimized ABP [79] contains many barriers. In particular, owner operations contain 2 sequentially consistent, 1 acquire, and 1 release barriers, which significantly degrade its performance.

We note that these optimization results are optimal: relaxing any of these barriers produces a counterexample. To further increase our confidence in the verification result, we added another thief thread stealing one entry, and checked the optimized BWoS with GenMC. BWoS passes the check in 3 days with around 200 million execution explorations.

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

6 Evaluation

Experimental setup. We perform all experiments on two x86 machines connected via 10Gbps Ethernet link, each with 88 hyperthreads (x86) [71], and one Arm machine with 96 cores (arm) [70]. The operating system is Ubuntu 20.04.4 LTS with Linux kernel version 5.7.0.

6.1 Block Size and Memory Overhead

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

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

Therefore, selecting an appropriate block size is straightforward. Further fine-tuning of these parameters may be beneficial for extreme scenarios where memory-size constraints are present or the overly large block size becomes detrimental to stealing (§8).

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

6.2 Microbenchmarks

To verify our claims, we have designed a microbenchmark which supports both LIFO and FIFO work stealing and compared BWoS with the state-of-the-art algorithms: an off-the-shelf ABP [48] implementation from Taskflow v3.4.0 [35] with barrier optimization [79] (abp), the block-based bounded queue [106] (bbq), work stealing queues from Tokio v1.17.0 [38] (tokioq), Go's runtime v1.18 [36]

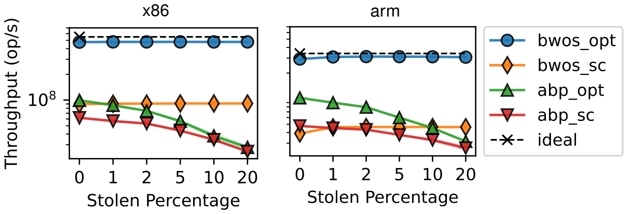


Figure 10: Throughput of LIFO BWoS and ABP with (opt) or without (sc) memory barrier optimization on x86 and arm running with different stolen percentages.

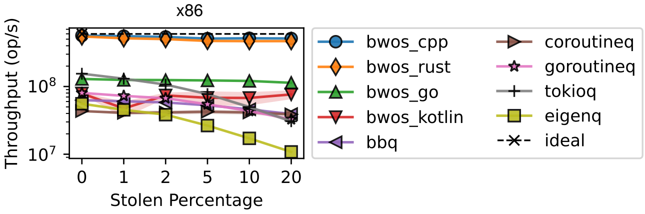


Figure 11: Throughput of FIFO BWoS and other state-of-the-art FIFO work stealing algorithms.

(goroutineq), Kotlin coroutines v1.6.4 [26] (coroutineq), and Eigen v3.3 [8] (eigenq).

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

- Single queue without stealing (§6.2.1): The owner thread executes the workload in a loop: it first puts until the queue is full and then gets until the queue is empty.
- Single queue with stealing (§6.2.2): An additional thief thread calls *steal* operations on the queue in a loop. We adjust the *put*/*get* ratio, and the idle time between each *steal* to perform the experiment at varying stolen percentages[2].
- Pool consisting of 8 queues (§6.2.3): 8 threads perform the following operations in a loop: put items to its queue until it is full, then get until it is empty, and then attempt to steal $k * C$ items from the pool, where k is the *balancing factor* (in percent), and C is the queue capacity. The threads are distributed equally between two NUMA domains, and within each NUMA domain between two L3 cache groups [58].

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

6.2.1 Queue without Stealing

Overall performance. Figures 10 and 11 for stolen percentage equal to 0 show the performance of the queue without stealing. BWoS outperforms other algorithms by a significant margin. For example, LIFO BWoS (bwos_opt) has 4.55x higher throughput than ABP (abp_opt) on x86, and FIFO BWoS written in C/C++, Rust, Go, and Kotlin outperform bbq in C, eigenq in C++, tokioq in Rust, goroutineq in Go, coroutineq in Kotlin by 8.9x, 10.15x, 3.55x, 1.61x, and 1.82x accordingly.

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

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

6.2.2 Queue with Stealing

Overall performance. As the stolen percentage increases, BWoS continues to outperform other work-stealing algorithms. For example, with 10% stolen percentage, LIFO BWoS outperforms abp by 12.59x, while FIFO BWoS outperforms bbq, eigenq, tokioq, goroutineq, coroutineq by 11.2x, 30.1x, 9.41x, 2.78x, and 1.64x respectively.

Effectiveness of the block-based approach. Unlike other algorithms, BWoS suffers only a minor performance drop as the stolen percentage increases. For example, for 20% stolen percentage, the throughput of LIFO and FIFO BWoS drops only by 0.53% and 9.35%, while for abp_opt, tokioq and goroutineq it degrades by 71.9%, 80.2%, and 59.3% respectively. Note that the BBQ concurrent FIFO queue [106], which is also a block-based design, does not reach performance comparable to BWoS, stressing the importance of our design decisions for the work stealing workloads.

6.2.3 Pool with Different Stealing Policies

Stealing policies. We perform this experiment with 6 stealing policies, namely the random choice policy (rand), a policy that chooses the victim based on a static configuration (seq), a policy that chooses the last selected one as the victim [104] (last), best of two (best_of_two), best of many (best_of_many), and NUMA-aware policy (numa). For best_of_many we choose best of half (i.e. best of four).

Overall performance. In this experiment, we compare BWoS only with the second-best algorithm from the previous experiments: abp and tokioq for LIFO and FIFO work stealing respectively. Fig. 12 shows that BWoS performs consistently better than other algorithms. When the balancing factor is 0%, BWoS outperforms abp by 4.69x and tokioq by 2.68x. As the

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

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

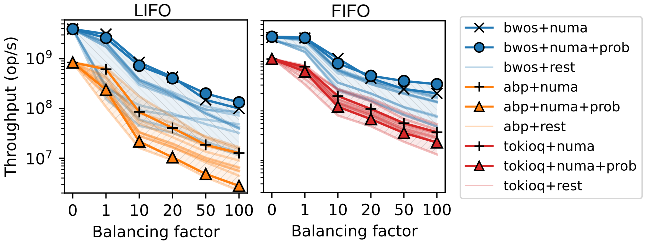


Figure 12: Throughput of the pool (8 queues) with different stealing policies and different balancing factors on x86. rest refers to all non-numa policies with and without probabilistic stealing.

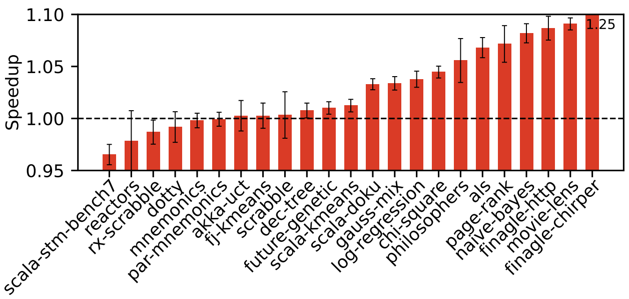


Figure 13: Speedup of 23 benchmarks from Renaissance benchmark suite on x86.

balancing factor increases, the throughput of BWoS variants is 7.90x higher than of abp and 6.45x higher than of tokioq.

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

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

6.3 Macrobenchmarks

6.3.1 Java G1GC

We replace the task queue [24] in Java 19 HotSpot [37] with LIFO BWoS, and run the Renaissance benchmark suite v0.14.0 [33], which consists of 25 modern, real-world, and concurrent benchmarks [95] designed for testing and optimizing garbage collectors. Two database benchmarks are omitted since they don't support JDK 19. JVM enables `-XX:+DisableExplicitGC` [30,68] and `-XX:+UseG1GC` flags when running the benchmark. All other parameters (*e.g.*, number of GC threads, VM memory limit) are default. We run 10 iterations for each benchmark with the modified and the original JVM, and measure the end-to-end program run time via the Renaissance testing framework.

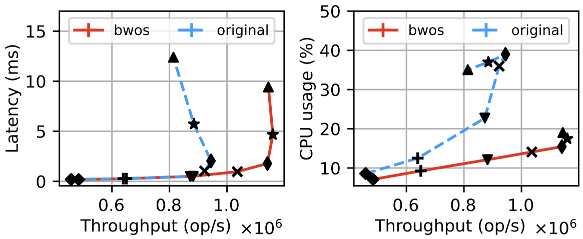


Figure 14: Throughput and latency results of Hyper HTTP server with BWoS and the original algorithm.

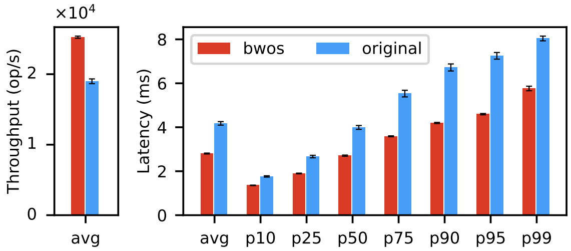


Figure 15: Throughput and latency of Tonic gRPC server with BWoS and the original algorithm.

Figure 13 shows the speedup of all 23 benchmarks on x86. When BWoS is enabled, 17 of them get performance improvement. The average speedup of all benchmarks is 3.55% and the maximum speedup is 25.3%. The applications that benefit more from concurrent GC also get greater speedup from BWoS. Results on arm are similar where the average speedup is 5.20%, 18 benchmarks are improved and the maximum speedup is 17.2%.

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

6.3.2 Rust Tokio Runtime

We replace the run queue [39] in Tokio v1.17.0 [38] with FIFO BWoS, and run Hyper HTTP server v0.14.18 [20] and Tonic gRPC server v0.6.2 [40] with the modified runtime. Tokio runtime (also Go runtime) provides a batch stealing interface. Based on observations from benchmarks similar to Fig. 2d, we configured the thief of BWoS to steal all available entries from its block at once. Benchmarks are performed on two x86 machines, one running the server, the other running the HTTP benchmarking tool wrk v4.2.0 [43] or the gRPC benchmarking and load testing tool ghz v0.017 [14]. All parameters of Hyper and Tonic are default. Each benchmark runs 100 seconds and

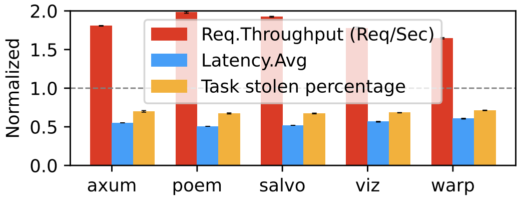


Figure 16: Request throughput, average latency, and task stolen percentage comparison results of 5 Rust web frameworks of BWoS *(normalized to the original algorithm results) with* rust-web-benchmarks *workload on* x86*.*

has 10 iterations. The latency and throughput are measured by wrk or ghz, while the CPU utilization of the server is collected through the Python psutil library [32]. wrk and ghz run the echo workload and SayHello protocol respectively and are configured to utilize all hyperthreads of their machine.

Figure 14 shows the throughput-latency and throughput-CPU utilization results of Hyper with different connection numbers (100, 200, 500, 1k, 2k, 5k, and 10k). Before the system is overloaded, BWoS provides 1.14×10^6 op/s throughput while dropping 60.4% CPU usage with similar latency, the original algorithm provides only 9.44×10^5 op/s throughput. With 1k connections, BWoS increases throughput by 12.3% with 6.74% lower latency and 60.9% lower CPU utilization.

Figure 15 shows the throughput and latency results of Tonic. Using BWoS increases throughput by 32.9%, with 32.8% lower average latency and 36.6% lower P95 latency.

To prove the generality of BWoS when applied to web frameworks, we also benchmark another 5 popular Rust web frameworks [4, 31, 34, 41, 42] that used Tokio runtime with rust-web-benchmarks [5] workload on x86 (Fig. 16). Results show that BWoS increases the throughput by 82.7% while dropping 45.1% of average latency. In addition, the task stolen percentage drops from 69.0% to 49.2%. We have made our implementation for the Tokio runtime available to the open-source community [3].

6.3.3 Go Runtime

We replace the runqueue [17] in the Go programming language [36] v1.18.0 runtime with BWoS and benchmark 9 JSON libraries [1, 9–12, 15, 18, 19, 25]. The benchmark suite [16] comes from the go-json library and runs 3 iterations with default parameters. We record the latency of each operation (*e.g.*, encoding/decoding small/medium/large JSON objects) reported by the benchmark suite, and calculate the speedup.

As shown in Fig. 17, when BWoS is enabled, operations get 25.8% average performance improvement on x86. arm produces similar results with 28.2% speedup on average. In general, encoding operations have better speedup compared to decoding operations. We observe no improvement for encoding booleans and integers.

7 Related Work

Block-based queues. Wang *et al.* proposed a block-based bounded queue [106] (BBQ) that splits the buffer into multiple blocks, thus reducing the producer-consumer interference. BWoS differs from BBQ in the following ways: (1) although BBQ also applies metadata separation, the producer-consumer interference it reduces is not an issue for work stealing as these always execute on the same core. By introducing block-level synchronization, steal-from-middle property, and randomized stealing, FIFO BWoS outperforms BBQ by a large margin (§6). (2) For the round control in BWoS, the new round of a block is determined only by the round of its adjacent block instead of relying on global metadata, as the *version* mechanism in BBQ does. This design simplifies the round updating and reduces its overhead.

Owner-thief interference and synchronization costs. Attiya *et al.* proved that work stealing in general requires strong synchronization between the owner and thieves [49]. BWoS overcomes this issue by delegating this synchronization to the block advancement, thus removing it from the fast path. Acar *et al.* used a sequential deque with message passing to remove the owner's barrier overhead [44]. However, this design relies on explicit owner-thief communication, thus the steal operation cannot run to completion in parallel with the owner's operations. Dijk *et al.* proposed a deque-based LIFO work-stealing algorithm which splits the deque into owner and thief parts, thus reducing the owner's memory fences when they do not reach the queue split point [61]. However, the entries read by thieves cannot be reused until the whole deque is empty. Horie *et al.* proposed a similar idea, where each owner has a public queue that is accessible from other threads and a private queue that is only accessible by itself [68]. However, it requires more effort to deal with load balancing, *e.g.*, introducing global statistics metadata which causes more cache misses for the owner. In contrast, BWoS reduces the interference using techniques of block-level synchronization, and probabilistic and randomized stealing. Morrison *et al.* introduced work stealing algorithms which rely on the bounded TSO microarchitectural model, which x86 and SPARC CPUs were shown to possess [89]. Michael *et al.* reduced the thief-owner synchronization by allowing them to read the same task [87], which requires reengineering of tasks to be idempotent. BWoS exhibits correct and efficient execution on a wide range of CPU architectures without any additional requirements.

Stealing policies. Yang *et al.* gave a survey of scheduling parallel computations by work stealing [107]. Kumar *et al.* benchmarked and analyzed variations of stealing policies [76]. Mitzenmacher proposed to give the thief two choices for selecting the victim to have a better load balancing [88]. Most of the analyzed policies are size-based, and thus aim to reach the same goal as our probabilistic stealing policy—namely, better load balance. Hendler *et al.* allow thieves to steal half

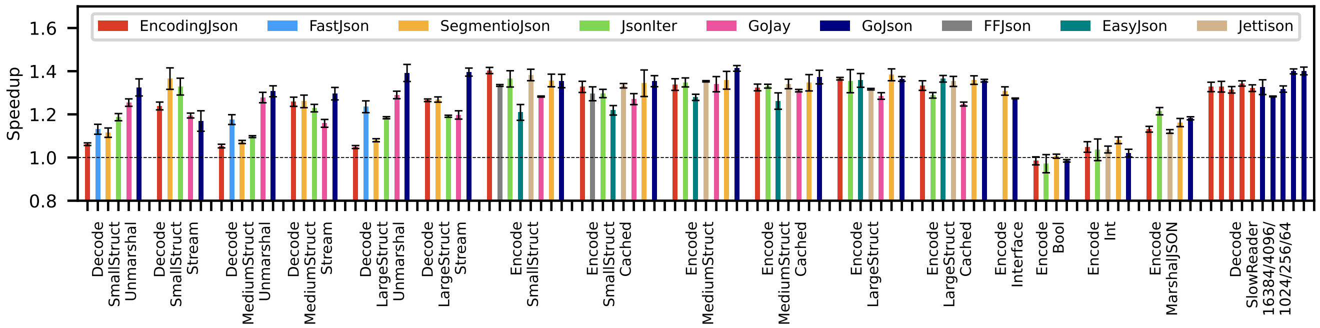


Figure 17: Speedup in the go-json library benchmark on x86 from using BWoS.

of the items in a given queue once to reduce interference [66]. BWoS supports batched stealing, but the maximum amount of data that can be stolen atomically is a block. However, the stealing policy can be configured to steal more than one block. Kumar *et al.* proposed a NUMA-aware policy for work stealing [77]. This policy is fully orthogonal to BWoS and can be combined with its probabilistic stealing policy.

Formally verified work stealing. Lê *et al.* [79] manually verified and optimized the memory barriers of Chase-Lev dequeue [53] on WMMs. Unlike the verification of BWoS which relies on model checking, manual verification is a high-effort undertaking. In the context of concurrent queues, Meta's FollyQ was verified using interactive theorem prover [105]. While this approach provides the highest levels of confidence in the design, it works only with sequentially consistent memory model, and is also a high-effort endeavor. Recently, GenMC authors have verified the ABP queue as part of evaluation of their model checker [74]. The authors of BBQ have relied on VSync to simultaneously verify and optimize the barrier for weak memory models [106]. BWoS also uses VSync for this purpose, but instead of many hand-crafted tests, which exercise the individual corner cases in BBQ, we create one comprehensive client that covers several corner cases and their interactions at once. We further verify the optimization results by adding one more thief into the verification client and checking it with GenMC.

8 Conclusion

To conclude, we explore two of our learnings from this work.

The benefit of the block-based design is manyfold. First, by replacing the global mutable metadata with block-level metadata, it is possible to eliminate the interference between the owner and the thieves that operate on different blocks. Second, by ensuring exclusive access to a block for owner's get operation through block-level synchronization, it is possible to relax most of the barriers from the operation's fast path, increasing its performance up to the theoretical upper bound. Although being unnecessary in our current algorithm, a third benefit is the verification modularity given by the block-based design, *e.g.*, allowing the verification of blocks and their composition in separate steps. Finally, the block-based design opens possibilities for holistic optimization of the data structure use, as we do with our probabilistic stealing policy.

BWoS can also be applied to GPU and hybrid CPU-GPU computations, as well as in HPC schedulers, where work stealing is common. We plan to explore this direction in the future. More generally, the BWoS design can be applied to other use cases, where the data structure is mostly accessed by a single thread, and only rarely by multiple. In this case, the decisions demonstrated in BWoS can act as design and implementation guidelines.

Verified software can be faster than unverified software. The more hardware details and tweaks are mirrored in the software, the more complex and opaque that piece of code becomes. The interaction of this complexity with concurrency and weak memory consistency is a major challenge. We believe that practical verification tools (*i.e.*, tools applied to increase confidence in correctness) are a key enabler in the development of efficient, and inevitably complex, concurrent software such as BWoS.

Future Work There are several directions for further work: We plan to contribute BWoS to more open-source projects, *e.g.*, openJDK [23, 29], and Golang, as well as investigate how to use BWoS in HPC runtimes. We also plan to better explore the performance trade-offs for BWoS: if the number of outstanding work items is smaller than the block size, BWoS can prevent stealing and thus limit the achieved parallelism. Furthermore, if the queue capacity has to be very small (due to space requirements), it may be necessary to reduce the block size and thus incur more block advancement that leads to performance drop. These situations would benefit from more exploration in the system design. In other cases, BWoS is expected to outperform existing state-of-the-art work-stealing algorithms due to its implementation of several performance-enhancing techniques.

Acknowledgments

We thank our shepherd Phillip Gibbons and the anonymous reviewers for their insightful comments.

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[29] OpenJDK. https://openjdk.org/.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[44] Acar, U. A., Charguéraud, A., and Rainey, M. Scheduling parallel programs by work stealing with private deques. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming* (2013), pp. 219–228.

[45] Alpern, B., and Schneider, F. B. Defining liveness. *Information processing letters 21*, 4 (1985), 181–185.

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

[47] Arnautov, S., Felber, P., Fetzer, C., and Trach, B. FFQ: A fast single-producer/multiple-consumer concurrent FIFO queue. In *2017 IEEE International Parallel and Distributed Processing Symposium, (IPDPS 2017)* (2017), pp. 907–916.

[48] Arora, N. S., Blumofe, R. D., and Plaxton, C. G. Thread scheduling for multiprogrammed multiprocessors. *Theory of computing systems 34*, 2 (2001), 115–144.

[49] Attiya, H., Guerraoui, R., Hendler, D., Kuznetsov, P., Michael, M. M., and Vechev, M. T. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011* (2011), T. Ball and M. Sagiv, Eds., ACM, pp. 487–498.

[50] Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. Cilk: An efficient multithreaded runtime system. *ACM SigPlan Notices 30*, 8 (1995), 207–216.

[51] Blumofe, R. D., and Leiserson, C. E. Scheduling multithreaded computations by work stealing. *J. ACM 46*, 5 (1999), 720–748.

[52] Cederman, D., and Tsigas, P. On dynamic load balancing on graphics processors. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware* (2008), pp. 57–64.

[53] Chase, D., and Lev, Y. Dynamic circular work-stealing deque. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures* (2005), pp. 21–28.

[54] Chatterjee, S., Grossman, M., Sbîrlea, A., and Sarkar, V. Dynamic task parallelism with a GPU work-stealing runtime system. In *International Workshop on Languages and Compilers for Parallel Computing* (2011), Springer, pp. 203–217.

[55] Cho, I., Saeed, A., Fried, J., Park, S. J., Alizadeh, M., and Belay, A. Overload Control for µs-scale RPCs with Breakwater. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (2020), pp. 299–314.

[56] Cong, G., Kodali, S., Krishnamoorthy, S., Lea, D., Saraswat, V., and Wen, T. Solving large, irregular graph problems using adaptive work-stealing. In *2008 37th International Conference on Parallel Processing* (2008), IEEE, pp. 536–545.

[57] Conway, M. E. A multiprocessor system design. In *Proceedings of the November 12-14, 1963, fall joint computer conference* (1963), pp. 139–146.

[58] de Lima Chehab, R. L., Paolillo, A., Behrens, D., Fu, M., Härtig, H., and Chen, H. CLOF: A compositional lock framework for multi-level NUMA systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (2021), pp. 851–865.

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

[60] Detlefs, D., Flood, C., Heller, S., and Printezis, T. Garbage-first garbage collection. In *Proceedings of the 4th international symposium on Memory management* (2004), pp. 37–48.

[61] DIJK, T. V., AND POL, J. C. Lace: non-blocking split deque for work-stealing. In *European Conference on Parallel Processing* (2014), Springer, pp. 206–217.

[62] GAVRILENKO, N., PONCE-DE LEÓN, H., FURBACH, F., HELJANKO, K., AND MEYER, R. BMC for weak memory models: Relation analysis for compact SMT encodings. In *International Conference on Computer Aided Verification* (2019), Springer, pp. 355–365.

[63] HALSTEAD JR, R. H. Implementation of Multilisp: Lisp on a multiprocessor. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming* (1984), pp. 9–17.

[64] HARRIS, T., AND KAESTLE, S. Callisto-RTS: Fine-grain parallel loops. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)* (2015), pp. 45–56.

[65] HARRIS, T., MAAS, M., AND MARATHE, V. J. Callisto: Co-scheduling parallel runtime systems. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), pp. 1–14.

[66] HENDLER, D., AND SHAVIT, N. Non-blocking steal-half work queues. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing* (2002), pp. 280–289.

[67] HERLIHY, M., AND SHAVIT, N. *The Art of Multiprocessor Programming*. Morgan Kaufmann, USA, 2011.

[68] HORIE, M., HORII, H., OGATA, K., AND ONODERA, T. Balanced double queues for GC work-stealing on weak memory models. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on Memory Management* (2018), pp. 109–119.

[69] HORIE, M., OGATA, K., TAKEUCHI, M., AND HORII, H. Scaling up parallel GC work-stealing in many-core environments. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management* (2019), pp. 27–40.

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

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

[72] KNUTH, D. E. *The Art of Computer Programming*, vol. 3. Pearson Education, 1997.

[73] KOGIAS, M., PREKAS, G., GHOSN, A., FIETZ, J., AND BUGNION, E. R2P2: Making RPCs first-class datacenter citizens. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)* (2019), pp. 863–880.

[74] KOKOLOGIANNAKIS, M., RAAD, A., AND VAFEIADIS, V. Model checking for weakly consistent libraries. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2019), PLDI 2019, Association for Computing Machinery, pp. 96–110.

[75] KOKOLOGIANNAKIS, M., AND VAFEIADIS, V. GENMC: A model checker for weak memory models. In *International Conference on Computer Aided Verification* (2021), Springer, pp. 427–440.

[76] KUMAR, S., AND SAHU, A. 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), IEEE, pp. 28–35.

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

[78] LAHAV, O., VAFEIADIS, V., KANG, J., HUR, C., AND DREYER, D. Repairing sequential consistency in C/C++11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017* (2017), A. Cohen and M. T. Vechev, Eds., ACM, pp. 618–632.

[79] LÊ, N. M., POP, A., COHEN, A., AND NARDELLI, F. Z. Correct and efficient work-stealing for weak memory models. In *18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2013* (2013), Association for Computing Machinery, pp. 69–79.

[80] LEA, D. A Java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande* (2000), pp. 36–43.

[81] LEIJEN, D., SCHULTE, W., AND BURCKHARDT, S. The design of a task parallel library. *Acm Sigplan Notices 44*, 10 (2009), 227–242.

[82] LI, J., DINH, S., KIESELBACH, K., AGRAWAL, K., GILL, C., AND LU, C. Randomized work stealing for large scale soft real-time systems. In *2016 IEEE Real-Time Systems Symposium (RTSS)* (2016), IEEE, pp. 203–214.

[83] LIU, C., SONG, P., LIU, Y., AND HAO, Q. 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), IEEE, pp. 149–152.

[84] LORCH, J. R., CHEN, Y., KAPRITSOS, M., PARNO, B., QADEER, S., SHARMA, U., WILCOX, J. R., AND ZHAO, X. Armada: Low-effort verification of high-performance concurrent programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (2020), pp. 197–210.

[85] MARTÍNEZ, M. A., FRAGUELA, B. B., AND CABALEIRO, J. C. A parallel skeleton for divide-and-conquer unbalanced and deep problems. *International Journal of Parallel Programming 49*, 6 (2021), 820–845.

[86] MCCLURE, S., OUSTERHOUT, A., SHENKER, S., AND RATNASAMY, S. Efficient scheduling policies for microsecond-scale tasks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)* (2022), pp. 1–18.

[87] MICHAEL, M. M., VECHEV, M. T., AND SARASWAT, V. A. Idempotent work stealing. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming* (2009), pp. 45–54.

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

[89] MORRISON, A., AND AFEK, Y. Fence-free work stealing on bounded TSO processors. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2014, Salt Lake City, UT, USA, March 1-5, 2014* (2014), R. Balasubramonian, A. Davis, and S. V. Adve, Eds., ACM, pp. 413–426.

[90] NORRIS, B., AND DEMSKY, B. CDSchecker: checking concurrent data structures written with C/C++ atomics. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013* (2013), A. L. Hosking, P. T. Eugster, and C. V. Lopes, Eds., ACM, pp. 131–150.

[91] NORRIS, B., AND DEMSKY, B. CDSchecker: checking concurrent data structures written with C/C++ atomics. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications* (2013), pp. 131–150.

[92] OBERHAUSER, J., CHEHAB, R. L. D. L., BEHRENS, D., FU, M., PAOLILLO, A., OBERHAUSER, L., BHAT, K., WEN, Y., CHEN, H., KIM, J., AND VAFEIADIS, V. VSync: Push-button verification and optimization for synchronization primitives on weak memory models. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2021), ASPLOS 2021, Association for Computing Machinery, pp. 530–545.

[93] OUYANG, K., SI, M., HORI, A., CHEN, Z., AND BALAJI, P. CAB-MPI: Exploring interprocess work-stealing towards balanced MPI communication. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis* (2020), IEEE, pp. 1–15.

[94] PODKOPAEV, A., LAHAV, O., AND VAFEIADIS, V. Bridging the gap between programming languages and hardware weak memory models. *Proc. ACM Program. Lang. 3*, POPL (2019), 69:1–69:31.

[95] PROKOPEC, A., ROSÀ, A., LEOPOLDSEDER, D., DUBOSCQ, G., TUMA, P., STUDENER, M., BULEJ, L., ZHENG, Y., VILLAZÓN, A., SIMON, D., ET AL. Renaissance: benchmarking suite for parallel applications on the JVM. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2019), pp. 31–47.

[96] QIAN, J., SRISA-AN, W., LI, D., JIANG, H., SETH, S., AND YANG, Y. Smartstealing: Analysis and optimization of work stealing in parallel garbage collection for Java VM. In *Proceedings of the Principles and Practices of Programming on The Java Platform*. 2015, pp. 170–181.

[97] SCHMAUS, F., PFEIFFER, N., HÖNIG, T., NOLTE, J., AND SCHRÖDER-PREIKSCHAT, W. Nowa: A wait-free continuation-stealing concurrency platform. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2021), IEEE, pp. 360–371.

[98] SCHWEIZER, H., BESTA, M., AND HOEFLER, T. Evaluating the cost of atomic operations on modern architectures. In *2015 International Conference on Parallel Architecture and Compilation (PACT)* (2015), IEEE, pp. 445–456.

[99] STEINBERGER, M., KAINZ, B., KERBL, B., HAUSWIESNER, S., KENZEL, M., AND SCHMALSTIEG, D. Softshell: dynamic scheduling on GPUs.

ACM Transactions on Graphics (TOG) 31, 6 (2012), 1–11.

[100] SUKSOMPONG, W., LEISERSON, C. E., AND SCHARDL, T. B. On the efficiency of localized work stealing. *Information Processing Letters 116*, 2 (2016), 100–106.

[101] SUO, K., RAO, J., JIANG, H., AND SRISA-AN, W. Characterizing and optimizing hotspot parallel garbage collection on multicore systems. In *Proceedings of the Thirteenth EuroSys Conference* (2018), pp. 1–15.

[102] TOSS, J. Work stealing inside GPUs.

[103] TZENG, S., PATNEY, A., AND OWENS, J. D. Task management for irregular-parallel workloads on the GPU.

[104] VENNERS, B. The Java Virtual Machine. *Java and the Java virtual machine: definition, verification, validation* (1998).

[105] VINDUM, S. F., FRUMIN, D., AND BIRKEDAL, L. Mechanized verification of a fine-grained concurrent queue from meta's folly library. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs* (2022), pp. 100–115.

[106] WANG, J., BEHRENS, D., FU, M., OBERHAUSER, L., OBERHAUSER, J., LEI, J., CHEN, G., HÄRTIG, H., AND CHEN, H. *BBQ*: A block-based bounded queue for exchanging data and profiling. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)* (2022), pp. 249–262.

[107] YANG, J., AND HE, Q. Scheduling parallel computations by work stealing: A survey. *International Journal of Parallel Programming 46*, 2 (2018), 173–197.

[108] ZHANG, Y., KUMAR, G., DUKKIPATI, N., WU, X., JHA, P., CHOWDHURY, M., AND VAHDAT, A. Aequitas: admission control for performance-critical RPCs in datacenters. In *Proceedings of the ACM SIGCOMM 2022 Conference* (2022), pp. 1–18.