usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

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

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

[1] HUAWEI

[2] TECHNISCHE UNIVERSITÄT DRESDEN

[3] MPI-SWS

[4] SHANGHAI JIAO TONG UNIVERSITY

Parallel Processing Scenarios

Parallel Processing Scenarios

Java

GO

julia

Kotlin

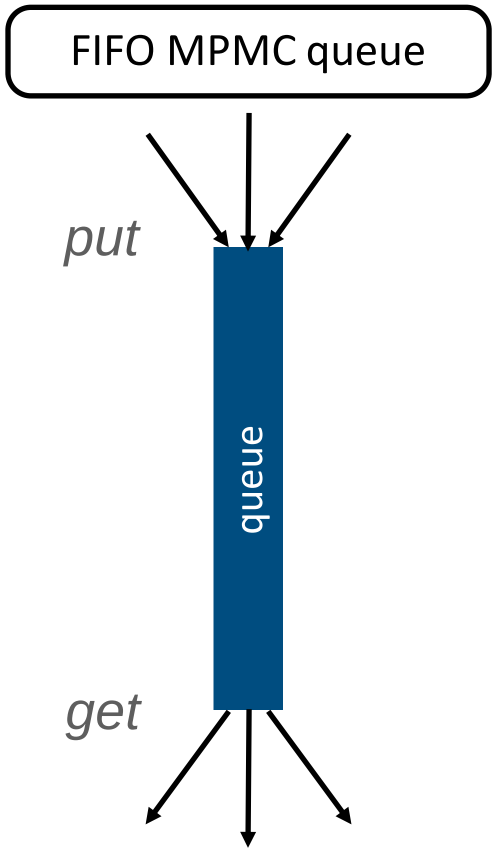
JS

Parallel Processing Scenarios

Intel® TBB runtime

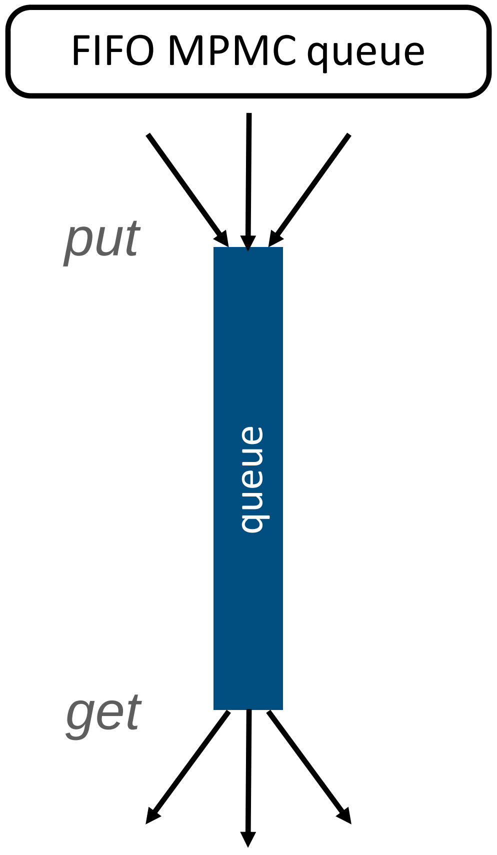
Queues for Parallel Processing

Existing Approaches

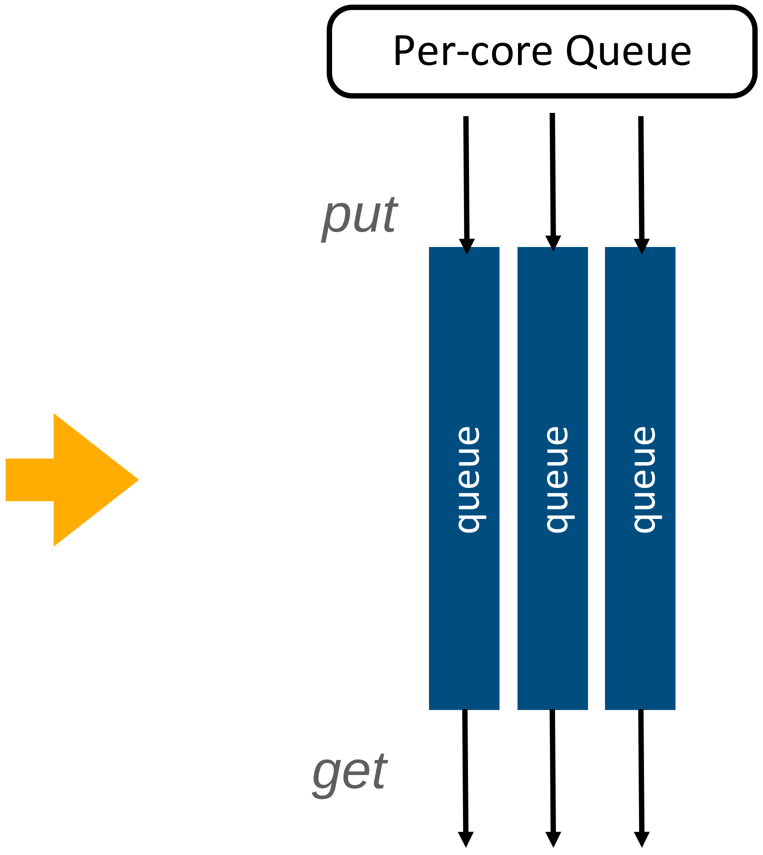


Queues for Parallel Processing

Existing Approaches

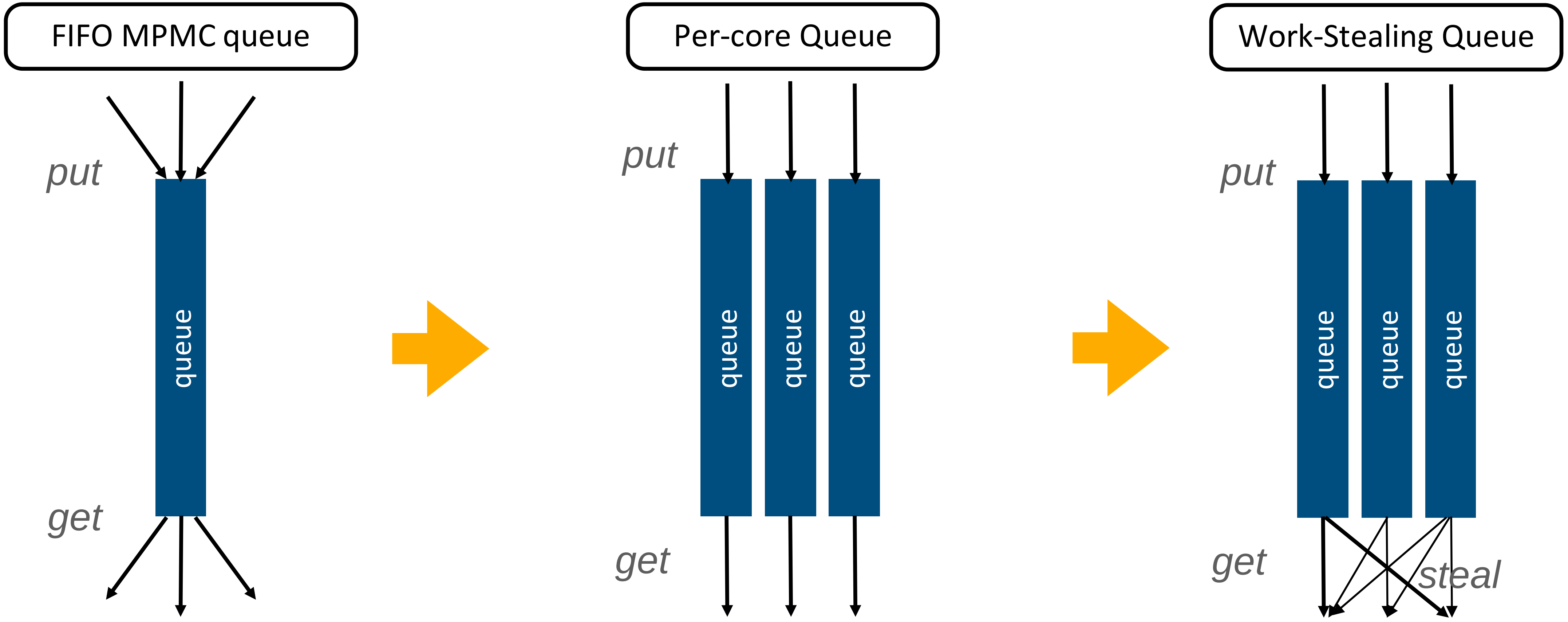


Does not scale



Queues for Parallel Processing

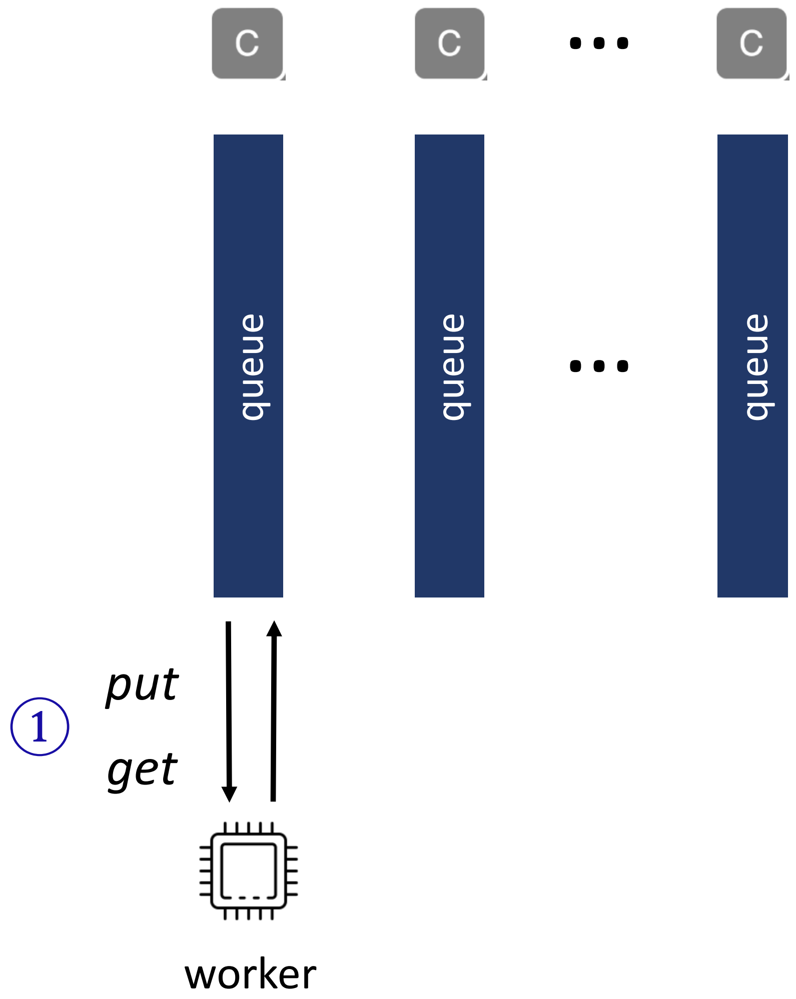
Existing Approaches



Does not scale

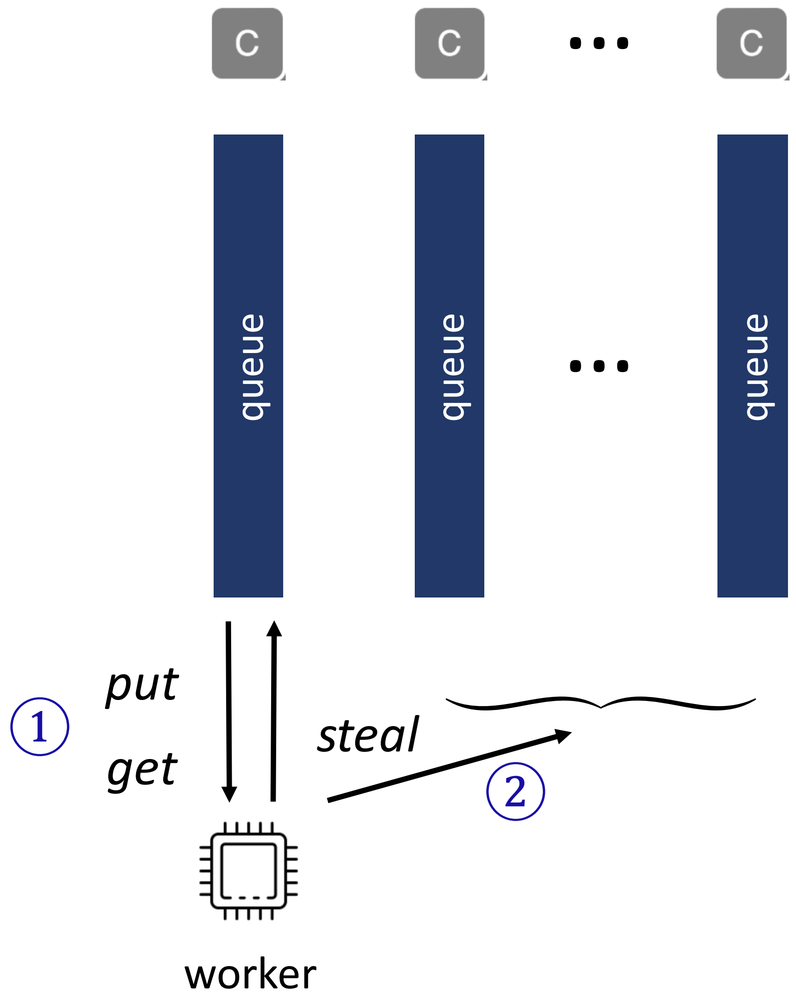
Fast but imbalanced

Work Stealing



① A worker (thread) puts on / gets from its own queue.

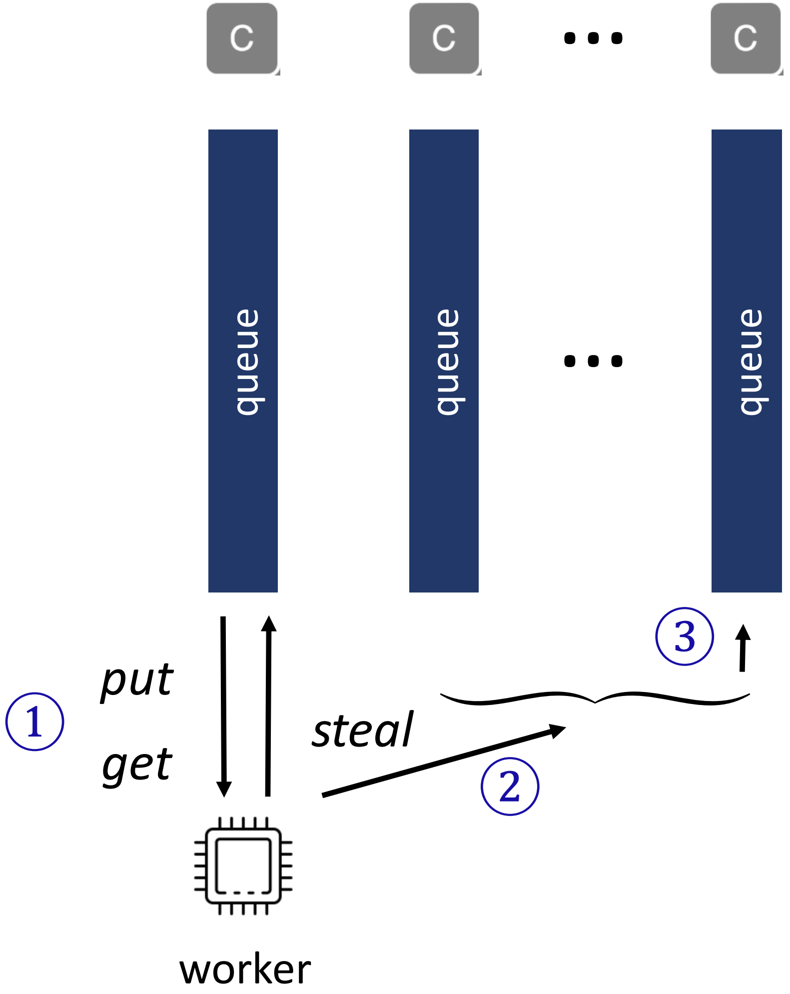
Work Stealing



1 A worker (thread) puts on / gets from its own queue.

2 When its queue is empty, it selects another queue…

Work Stealing



① A worker (thread) puts on / gets from its own queue.

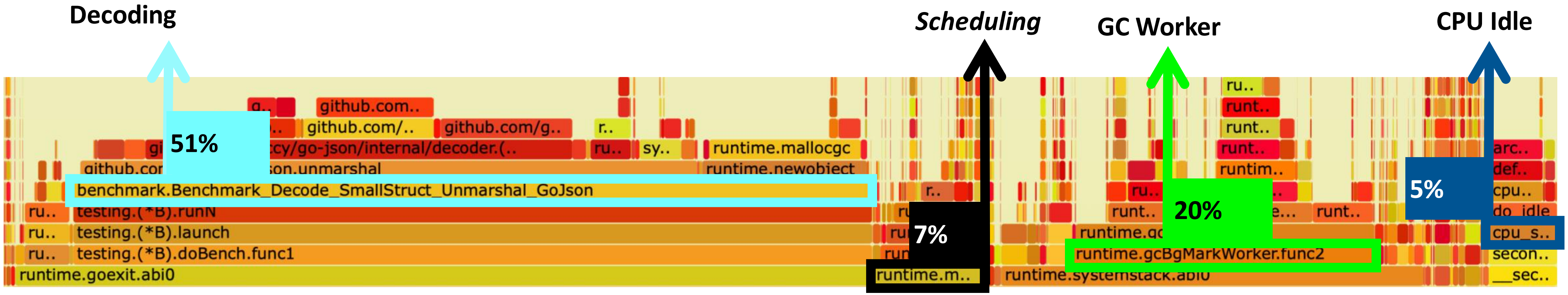
② When its queue is empty, it selects another queue…

③ and try to steal from it.

Work Stealing Becomes the Bottleneck

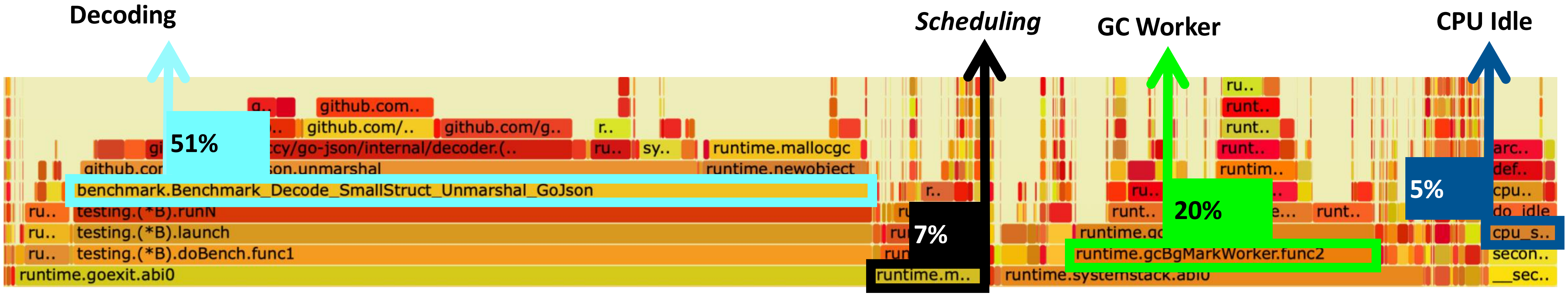
Work Stealing Becomes the Bottleneck

Example 1: GoJson Object Decoding Benchmark



Work Stealing Becomes the Bottleneck

Example 1: GoJson Object Decoding Benchmark



Example 2: Experience of Rust Tokio's author

[1]

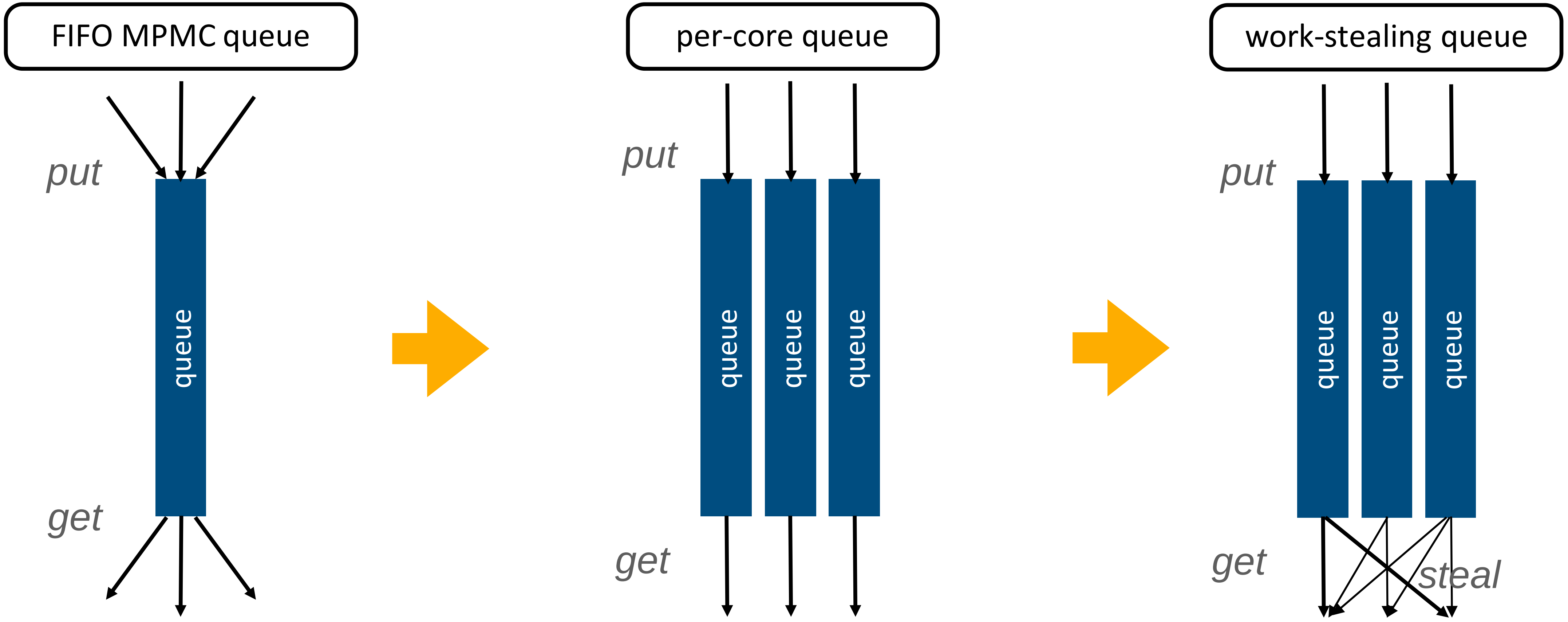
The run queue is at the heart of the scheduler. As such, it is probably the most critical component to get right. The original Tokio scheduler used crossbeam's deque implementation, which is single-producer, multi-consumer deque

contention is reduced. However, Rust's asynchronous tasks are expected to take very little time executing when popped from the run queue. In this scenario, the overhead from contending on the queue becomes significant.

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

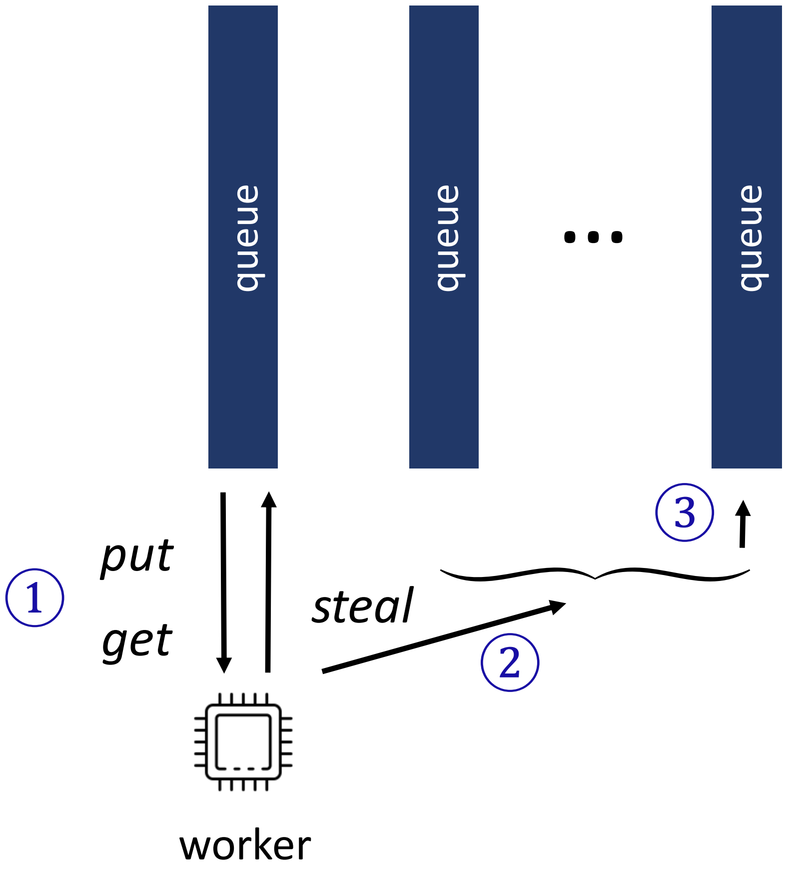
Queues for Parallel Processing

existing works



Does not scale — Fast but imbalanced — Not fast enough

Sources of the Overhead



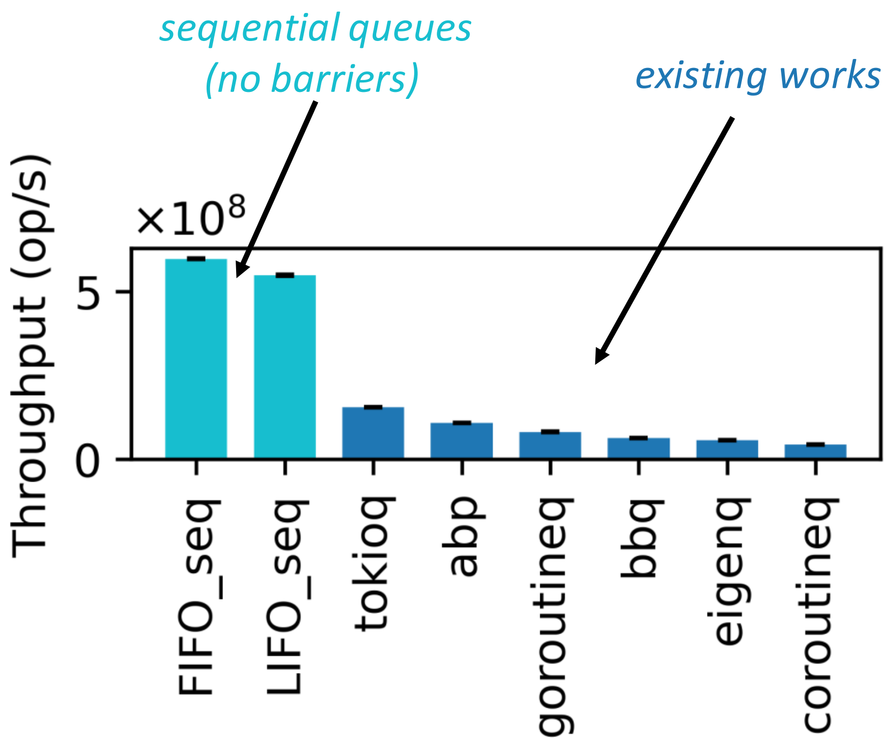
1. A worker (thread) puts on / gets from its own queue.
   FIFO / LIFO
   A) Cost of Synchronization Operations

2. When its queue is empty, it selects another queue…
   Random, best of two, NUMA-aware, Batching …
   B) Overhead due to Victim Selection *(see paper)*

3. and try to steal from it.
   C) Interference Cost with Thieves

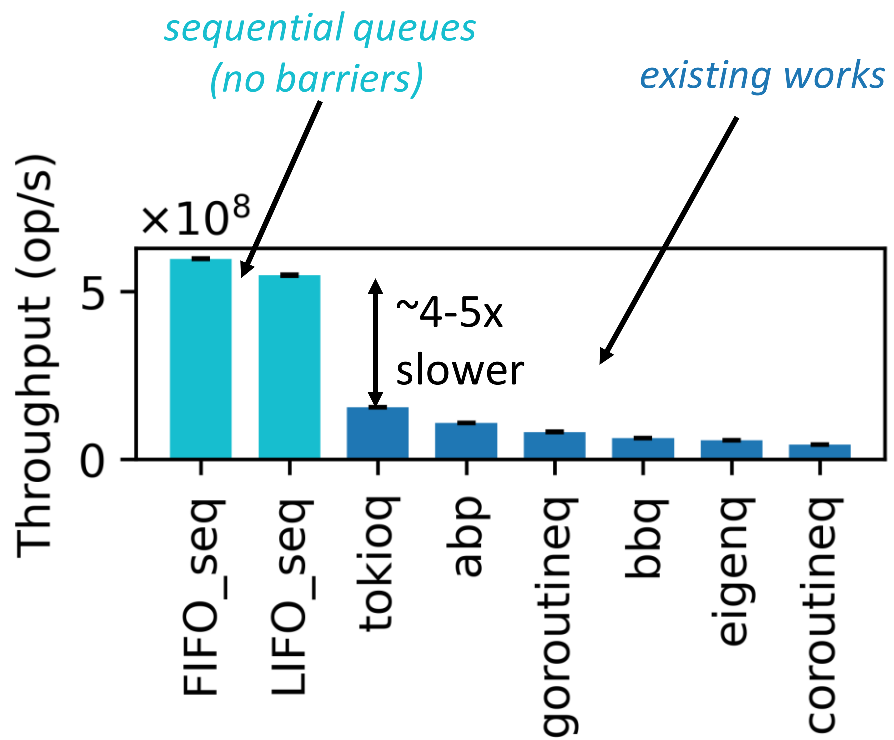
Sources of the Overhead

A) Cost of Synchronization Operations



Sources of the Overhead

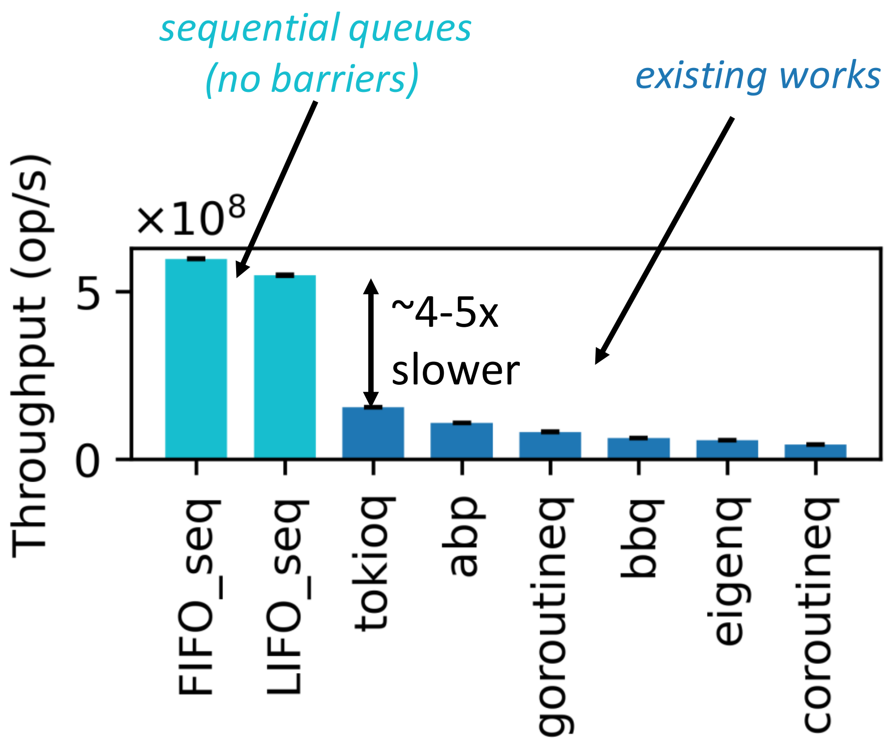
A) Cost of Synchronization Operations



- Throughput of existing works is far away from sequential queues (theoretical upper bound)

Sources of the Overhead

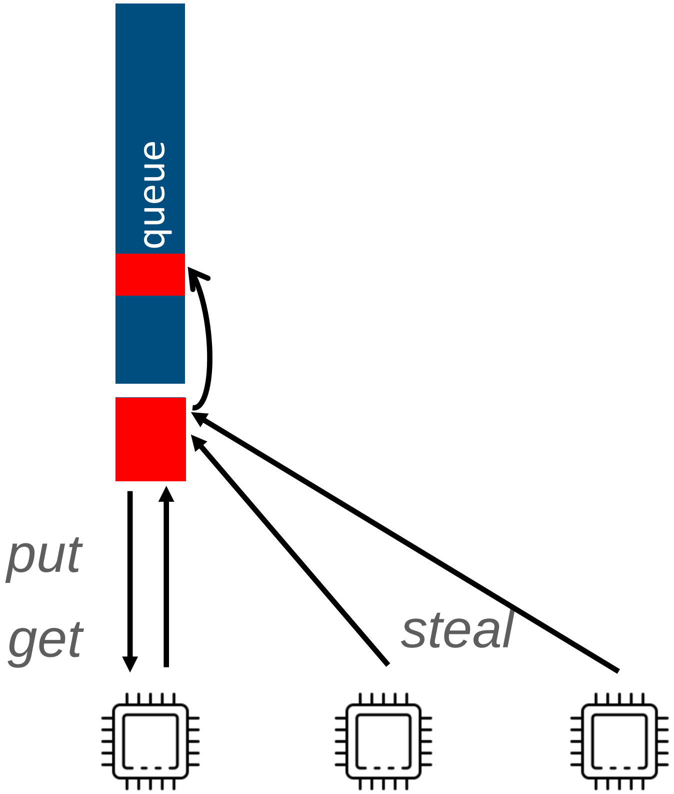
A) Cost of Synchronization Operations



- Throughput of existing works is far away from sequential queues (theoretical upper bound)
- As steals may happen at any time strong atomic barriers are introduced

Sources of the Overhead

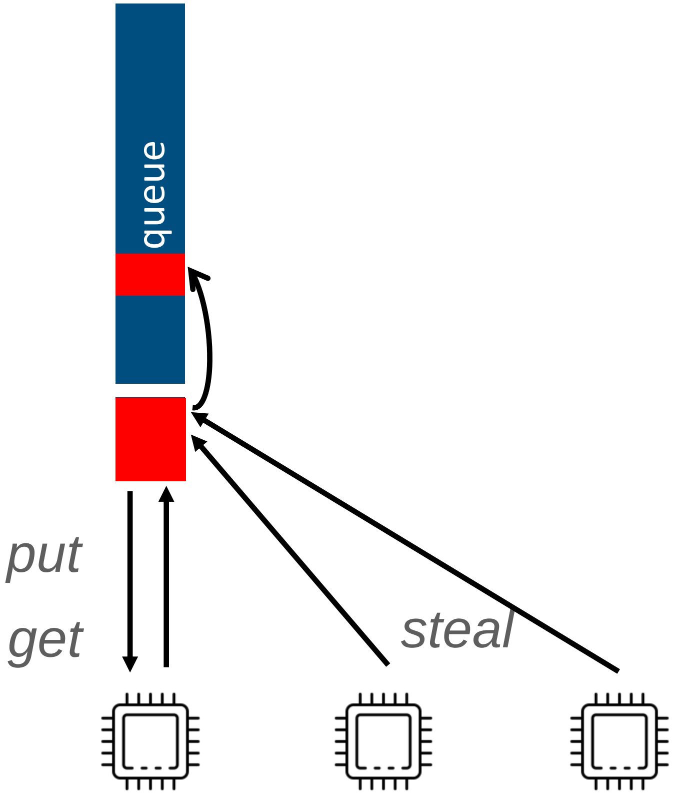
C) Cost of Interference with Thieves



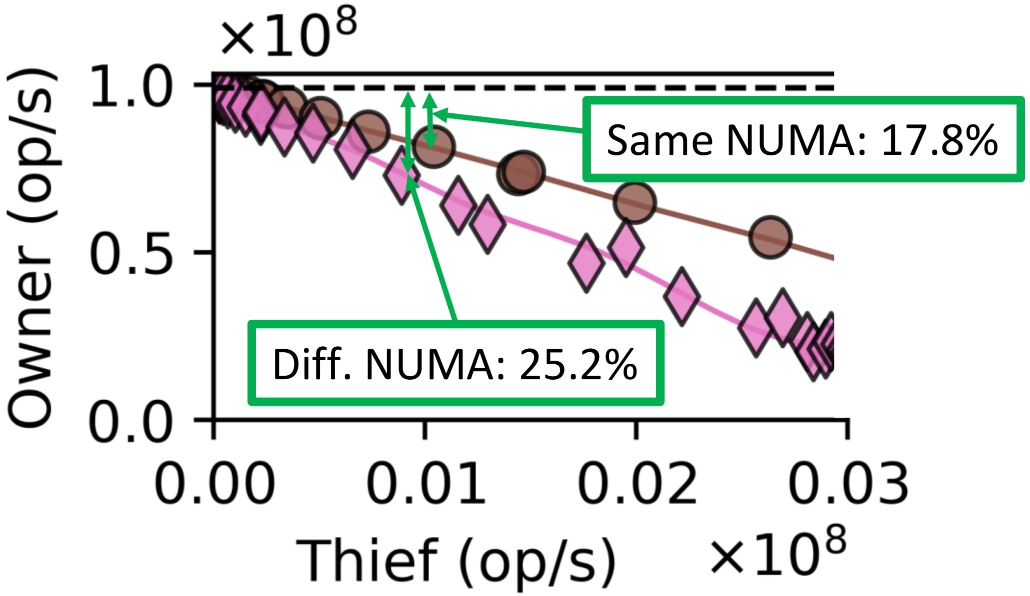
- Thieves affect the throughput of the owner

Sources of the Overhead

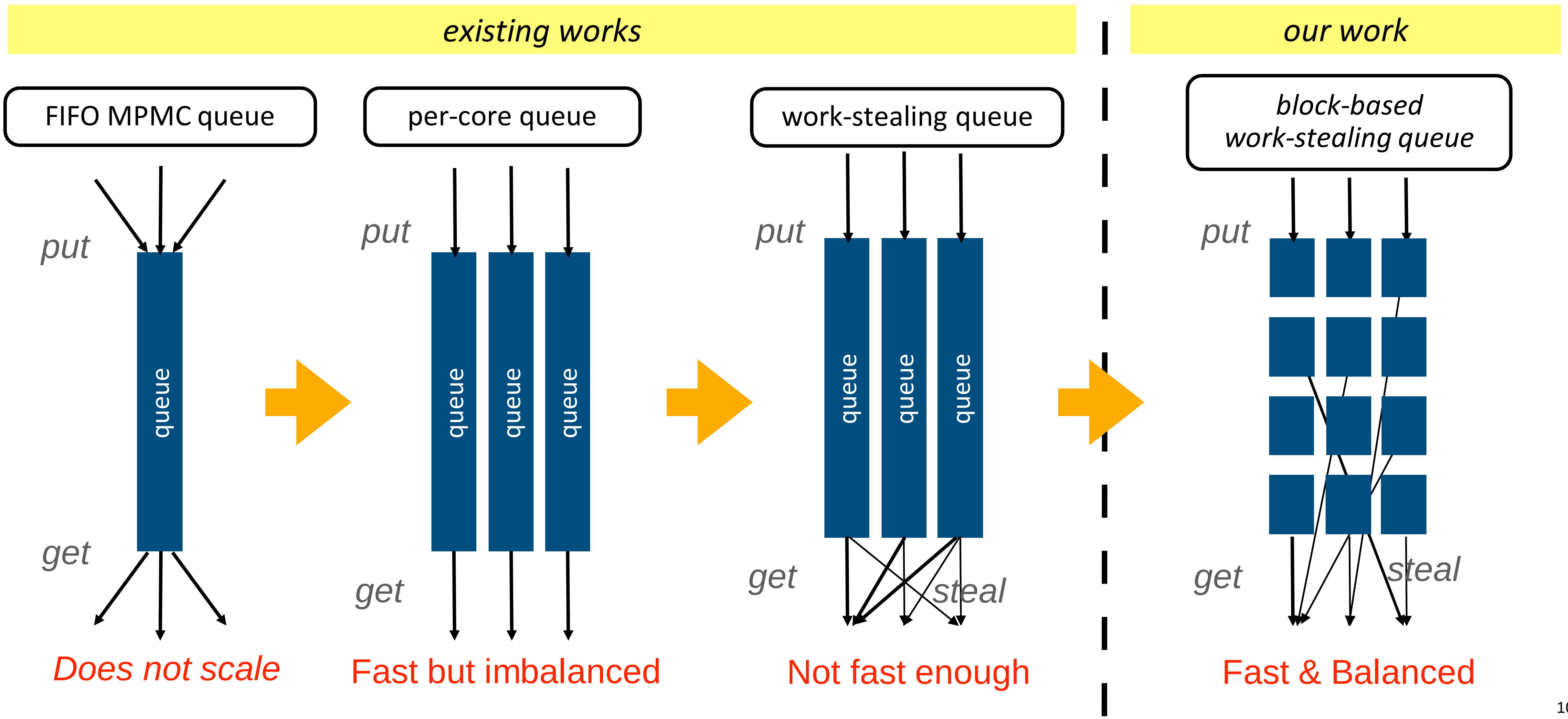
C) Cost of Interference with Thieves



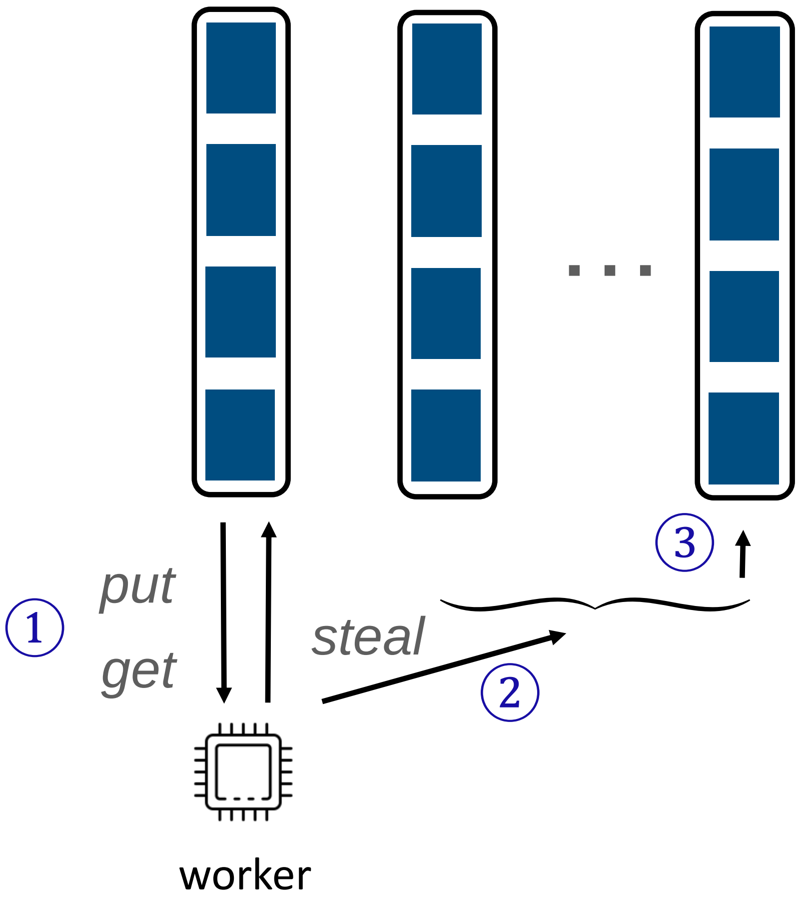
- Thieves affect the throughput of the owner
- Stealing 1% of the elements:



BWoS — Block-based Work Stealing



BWoS — Block-based Work Stealing



① A worker (thread) puts on / gets from its own queue.

FIFO / LIFO

Block-level synchronization: no barriers inside blocks

② When its queue is empty, it selects another queue…

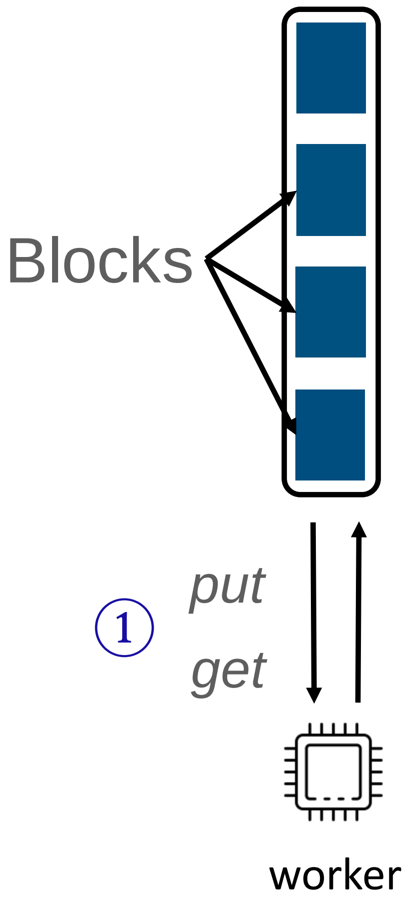
Random, best of two, NUMA-aware, Batching …

Randomized Victim Selection Policy

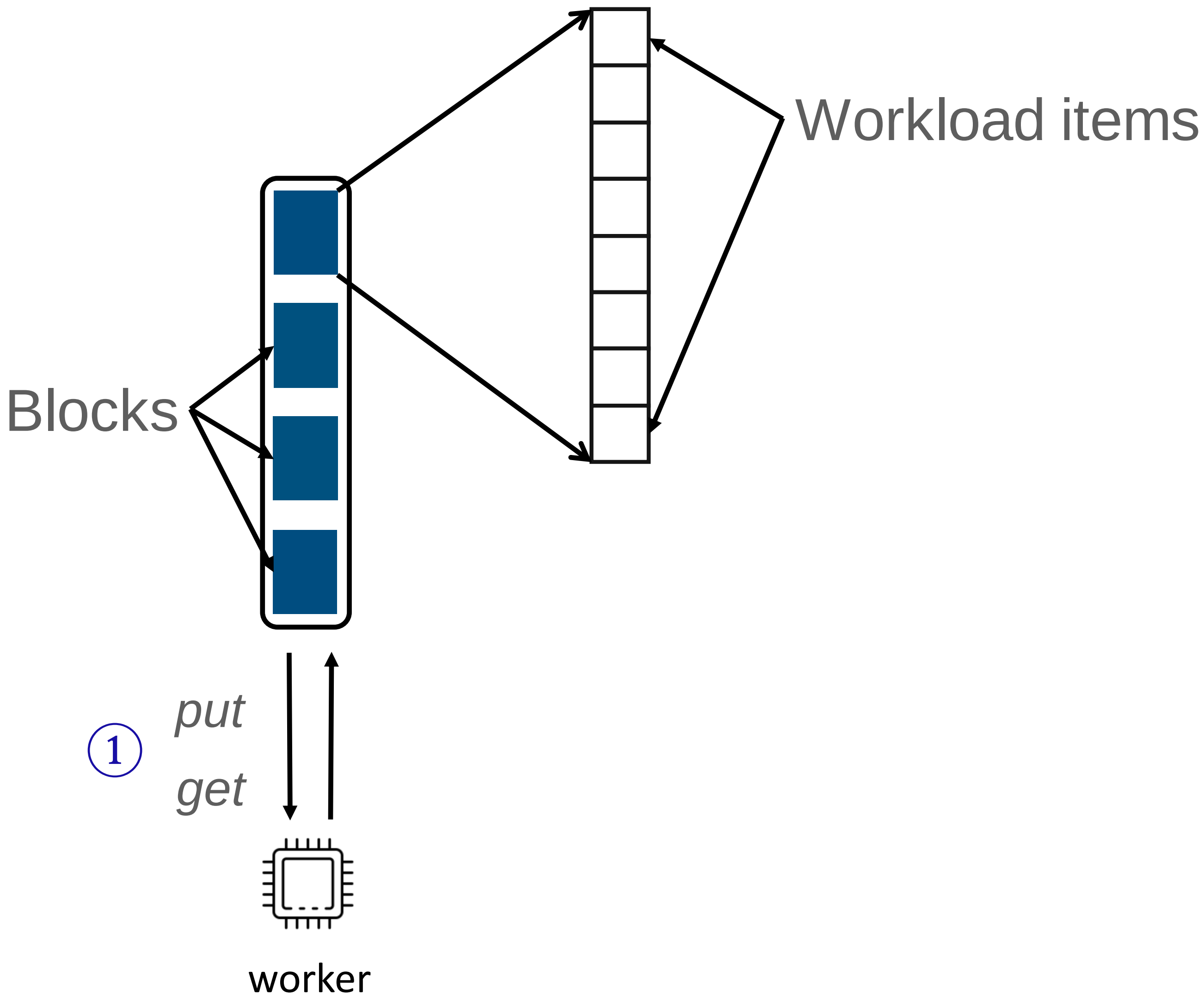
③ and try to steal from it.

No Interference when stealing from a different block

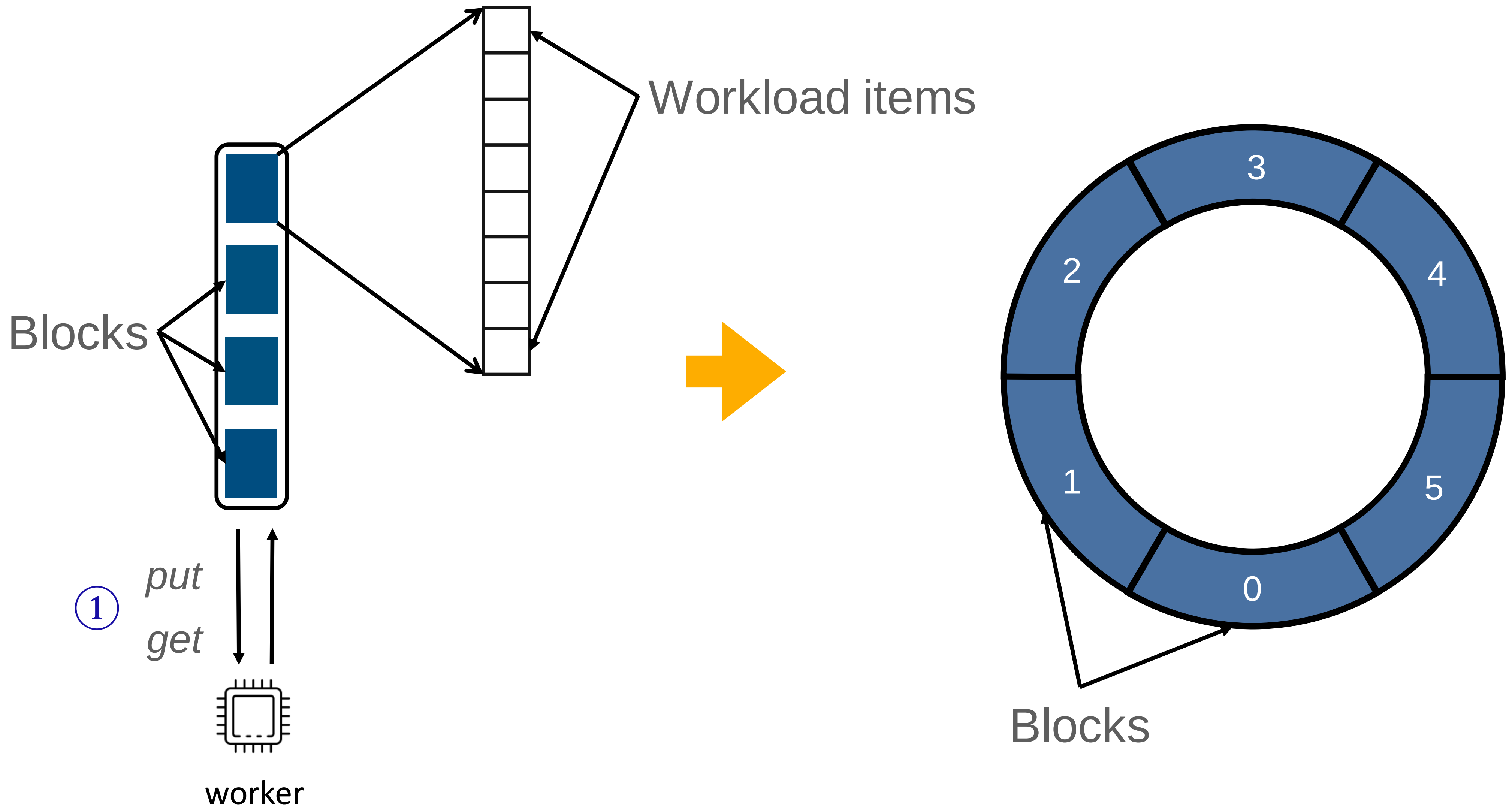
BWoS — Block-based Work Stealing



BWoS — Block-based Work Stealing



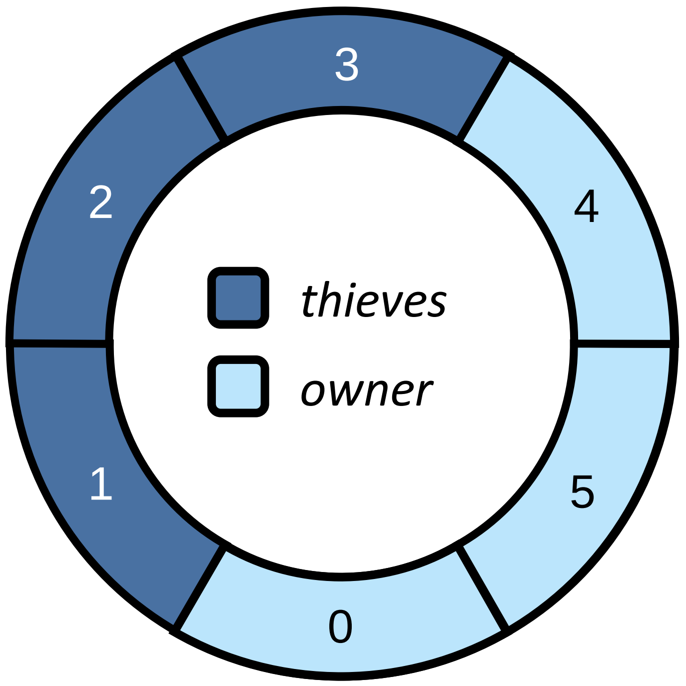
BWoS — Block-based Work Stealing



BWoS — Block-based Work Stealing

A) Cost of Synchronization Operations

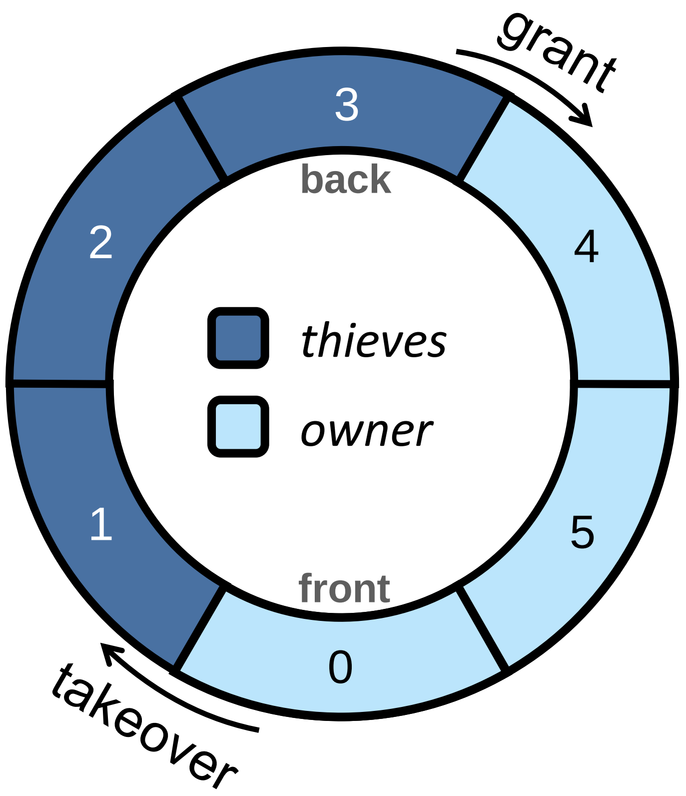
- Block-level synchronization: Each block is owned either by the owner or by the thieves



BWoS — Block-based Work Stealing

A) Cost of Synchronization Operations

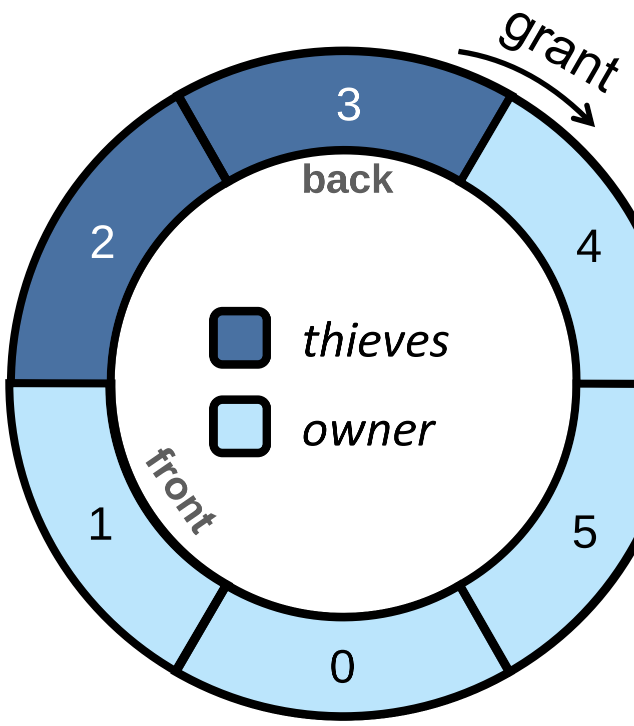
- Block-level synchronization: Each block is owned either by the owner or by the thieves
- When crossing block boundary: **use barriers** (takeover, grant)



BWoS — Block-based Work Stealing

A) Cost of Synchronization Operations

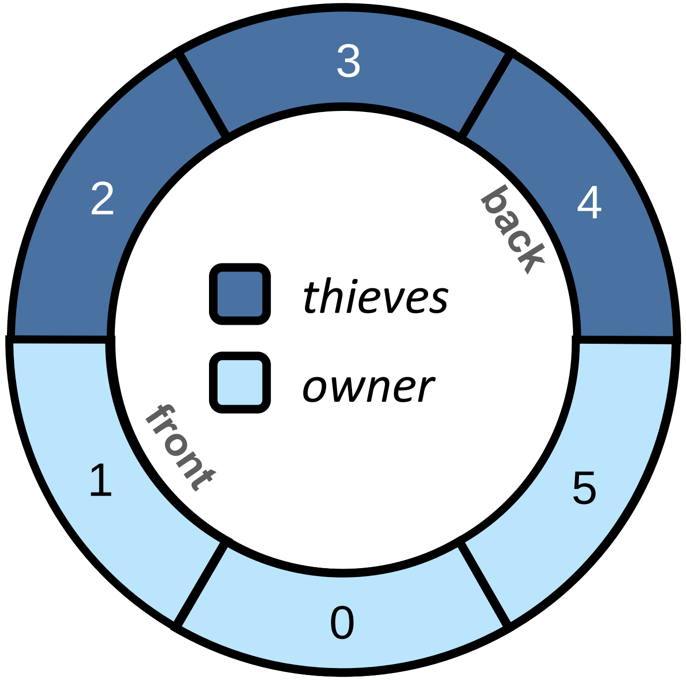
- Block-level synchronization: Each block is owned either by the owner or by the thieves
- When crossing block boundary: **use barriers** (takeover, grant)



BWoS — Block-based Work Stealing

A) Cost of Synchronization Operations

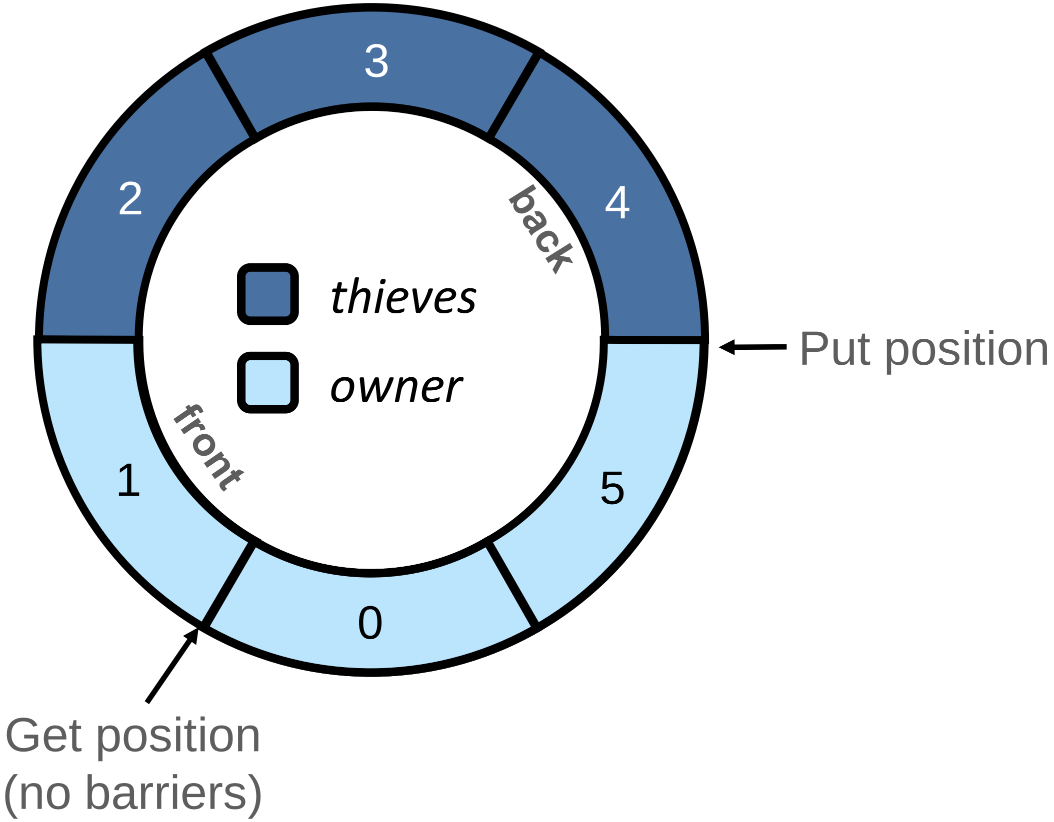
- Block-level synchronization: Each block is owned either by the owner or by the thieves
- When crossing block boundary: **use barriers** (takeover, grant)



BWoS — Block-based Work Stealing

A) Cost of Synchronization Operations

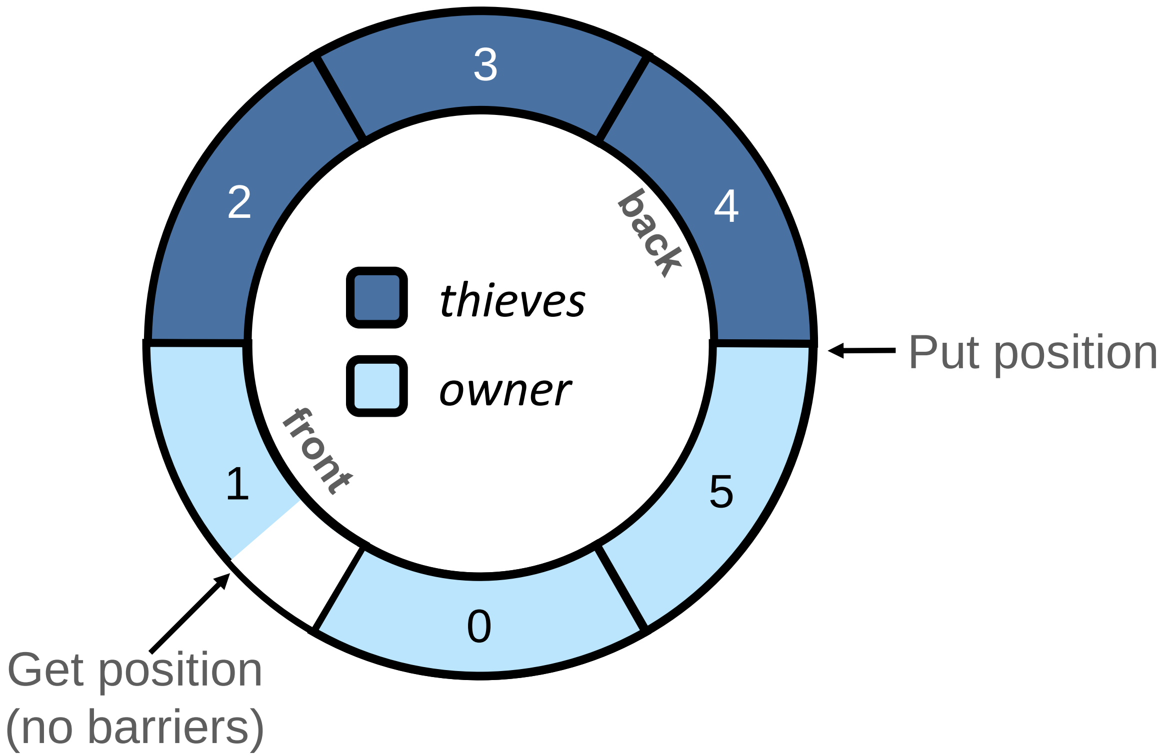
- Block-level synchronization: Each block is owned either by the owner or by the thieves
- When crossing block boundary: use barriers (takeover, grant)
- Within block boundaries: **use relaxed atomics, no barriers** (fast path)



BWoS — Block-based Work Stealing

A) Cost of Synchronization Operations

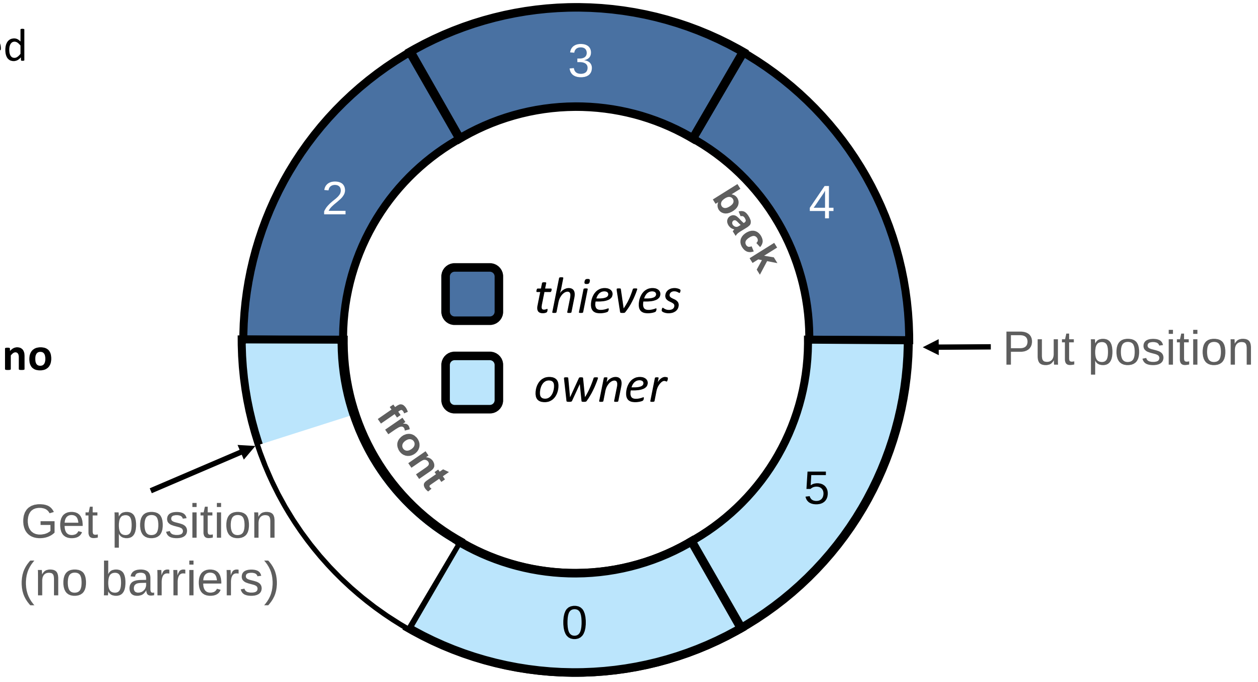
- Block-level synchronization: Each block is owned either by the owner or by the thieves
- When crossing block boundary: use barriers (takeover, grant)
- Within block boundaries: **use relaxed atomics, no barriers** (fast path)



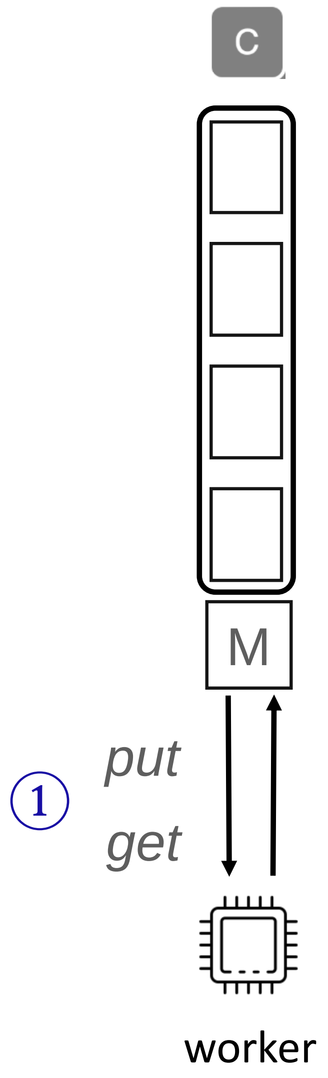
BWoS — Block-based Work Stealing

A) Cost of Synchronization Operations

- Block-level synchronization: Each block is owned either by the owner or by the thieves
- When crossing block boundary: use barriers (takeover, grant)
- Within block boundaries: **use relaxed atomics, no barriers** (fast path)

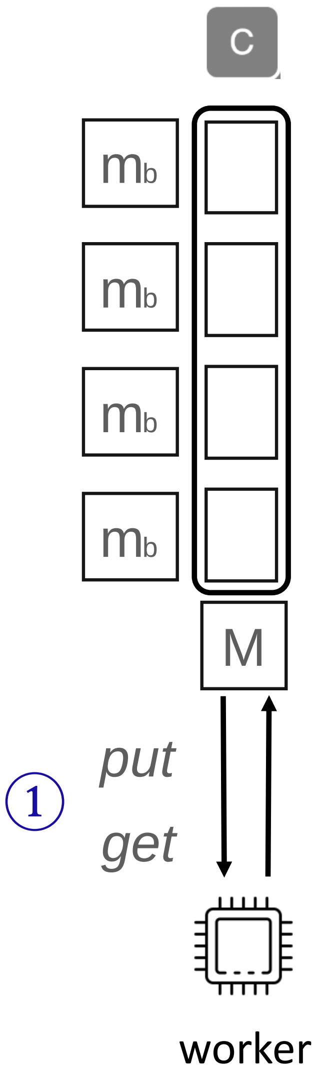


BWoS — Block-based Work Stealing



B) Overhead due to Victim Selection

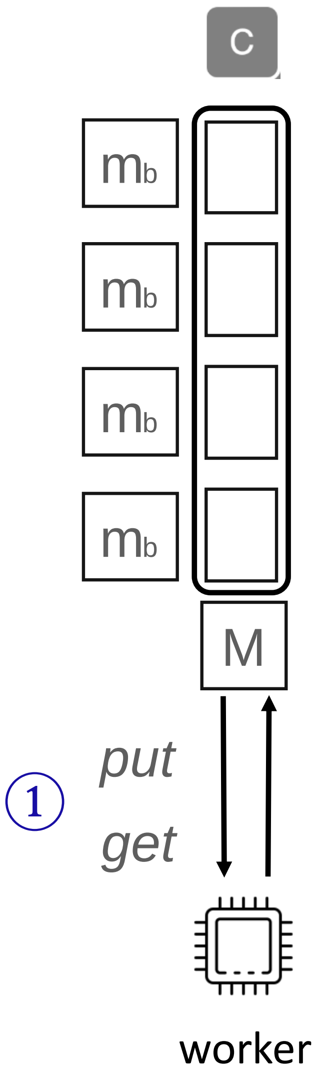
BWoS — Block-based Work Stealing



B) Overhead due to Victim Selection

- Each block has a dedicated metadata instance

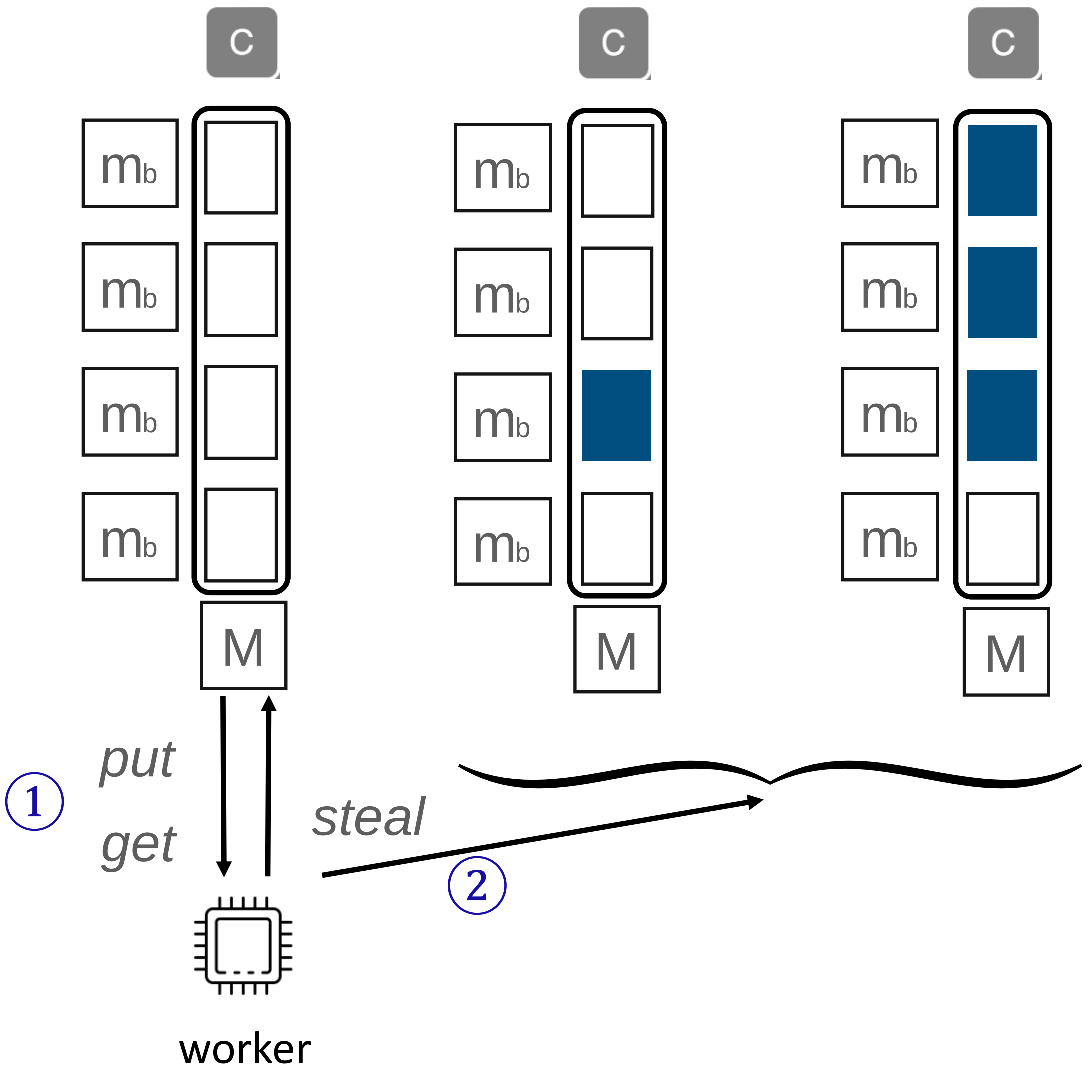
BWoS — Block-based Work Stealing



B) Overhead due to Victim Selection

- Each block has a dedicated metadata instance
- Novel probabilistic stealing policy:
  Use sampling to estimate Size/Capacity.
- Thieves read only block-level metadata, and steal from longer queues with higher probability

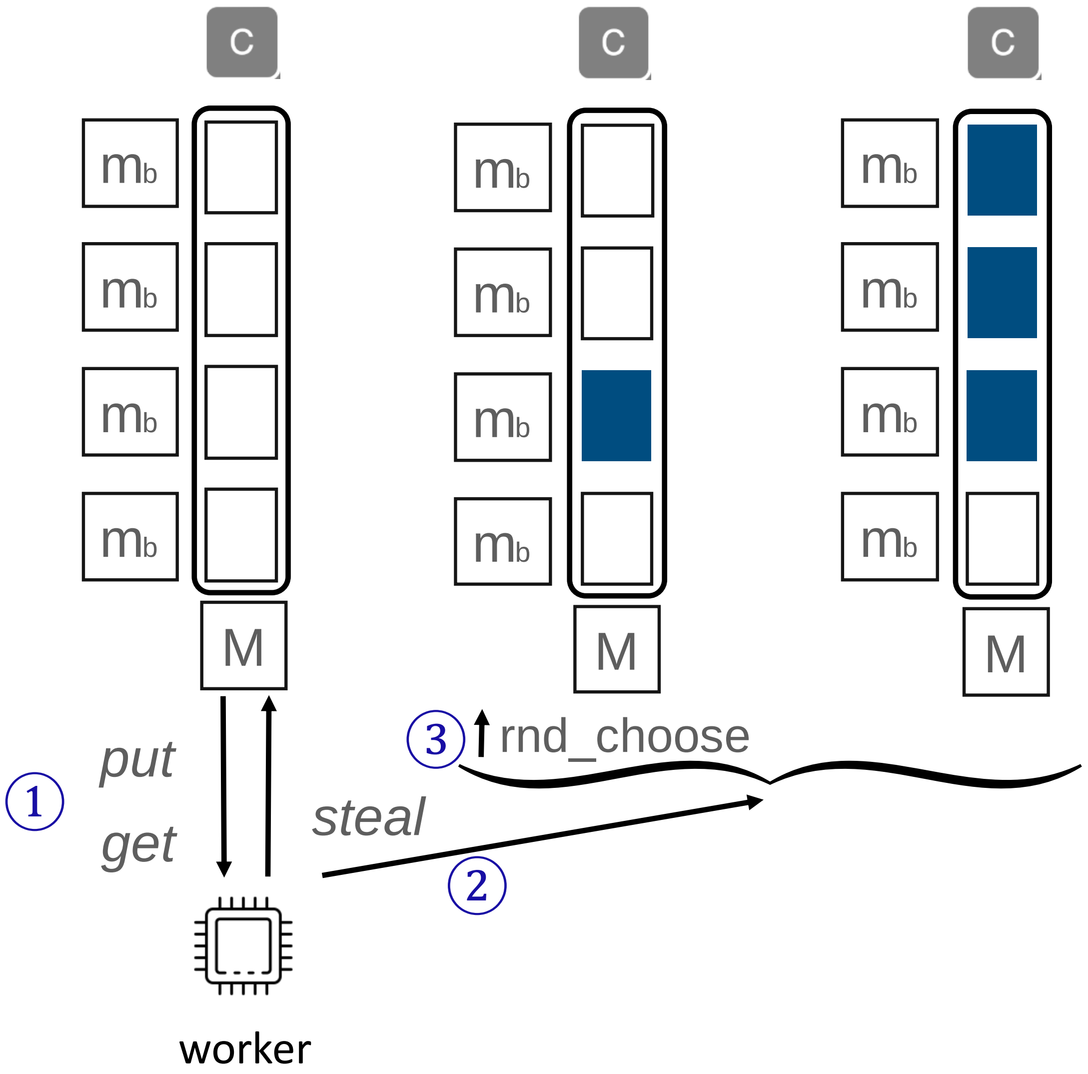
BWoS — Block-based Work Stealing



B) Overhead due to Victim Selection

- Each block has a dedicated metadata instance
- Novel probabilistic stealing policy:
  Use sampling to estimate Size/Capacity.
- Thieves read only block-level metadata, and steal from longer queues with higher probability

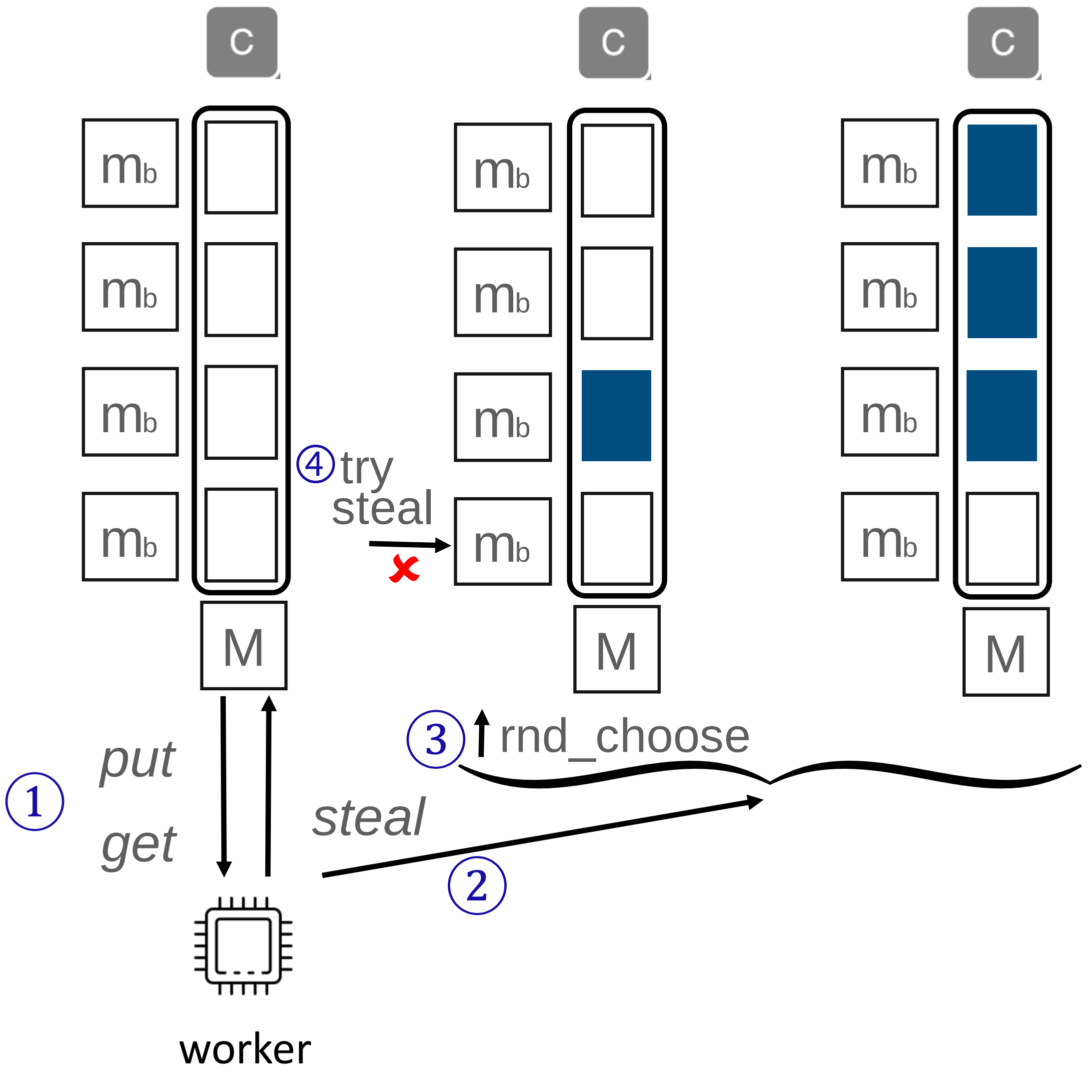
BWoS — Block-based Work Stealing



B) Overhead due to Victim Selection

- Each block has a dedicated metadata instance
- Novel probabilistic stealing policy:
  Use sampling to estimate Size/Capacity.
- Thieves read only block-level metadata, and steal from longer queues with higher probability

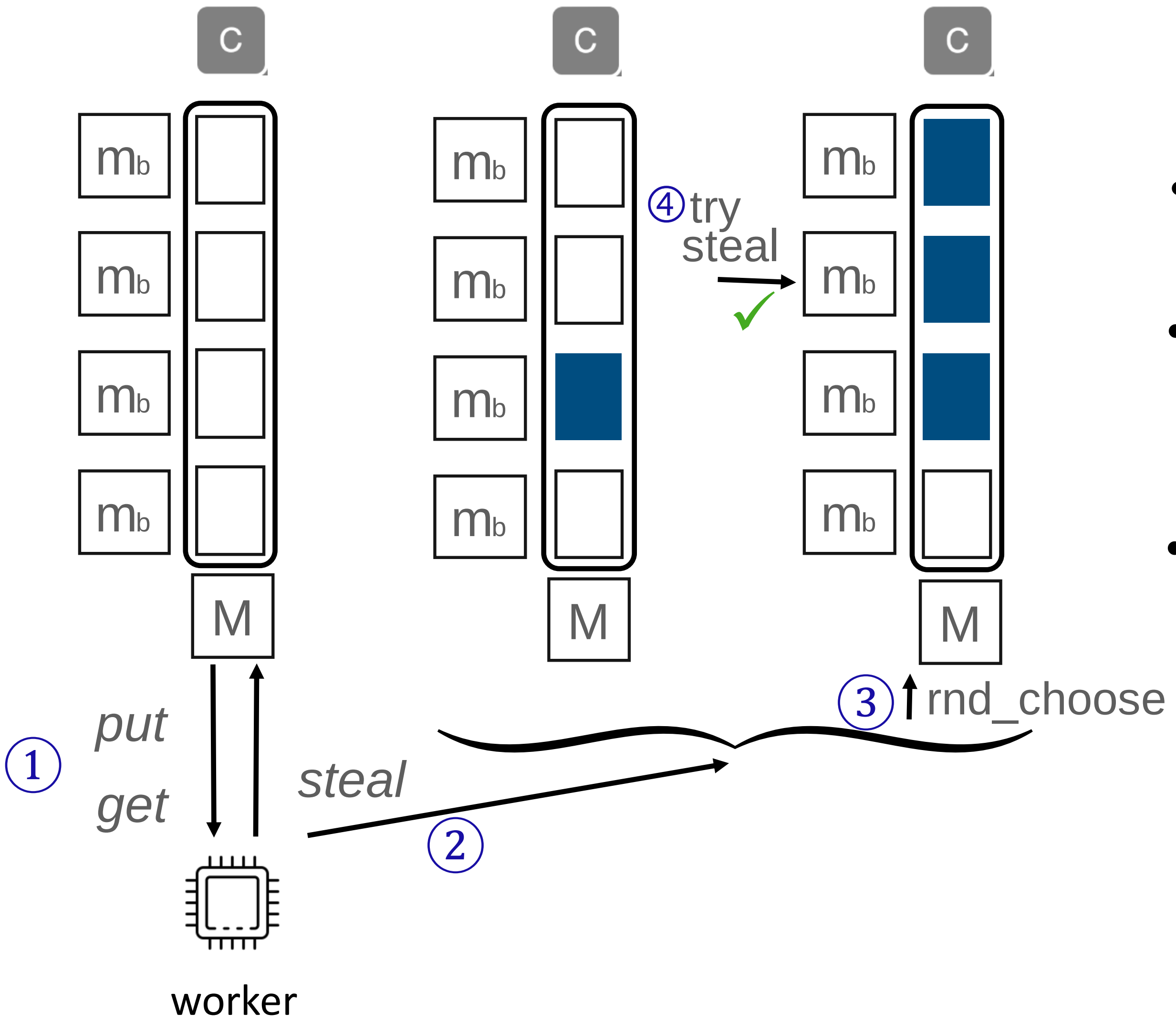
BWoS — Block-based Work Stealing



B) Overhead due to Victim Selection

- Each block has a dedicated metadata instance
- Novel probabilistic stealing policy:
  Use sampling to estimate Size/Capacity.
- Thieves read only block-level metadata, and steal from longer queues with higher probability

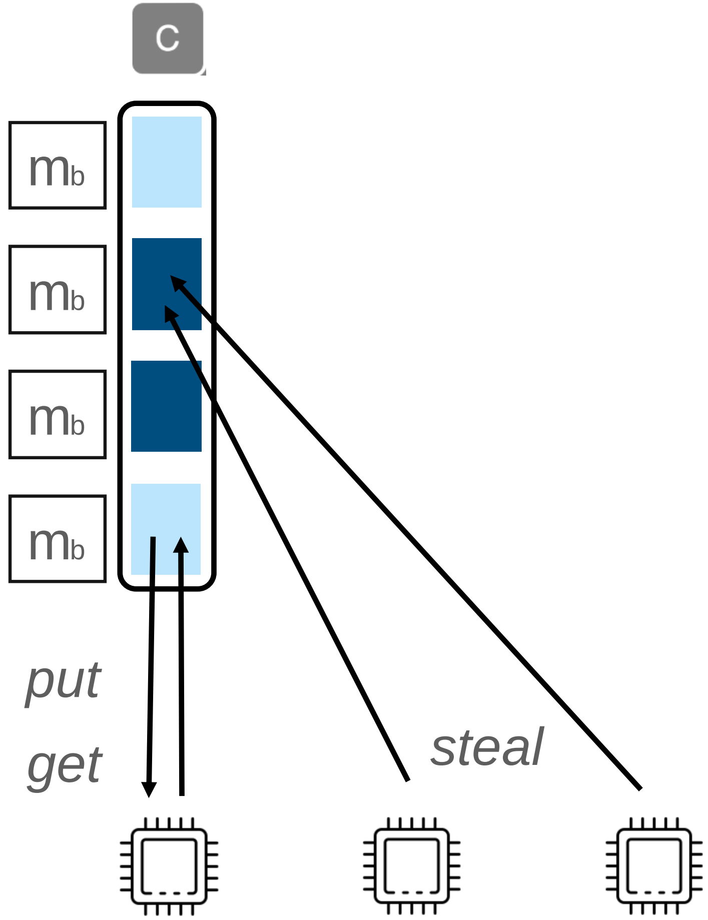
BWoS — Block-based Work Stealing



B) Overhead due to Victim Selection

- Each block has a dedicated metadata instance
- Novel probabilistic stealing policy:
  Use sampling to estimate Size/Capacity.
- Thieves read only block-level metadata, and steal from longer queues with higher probability

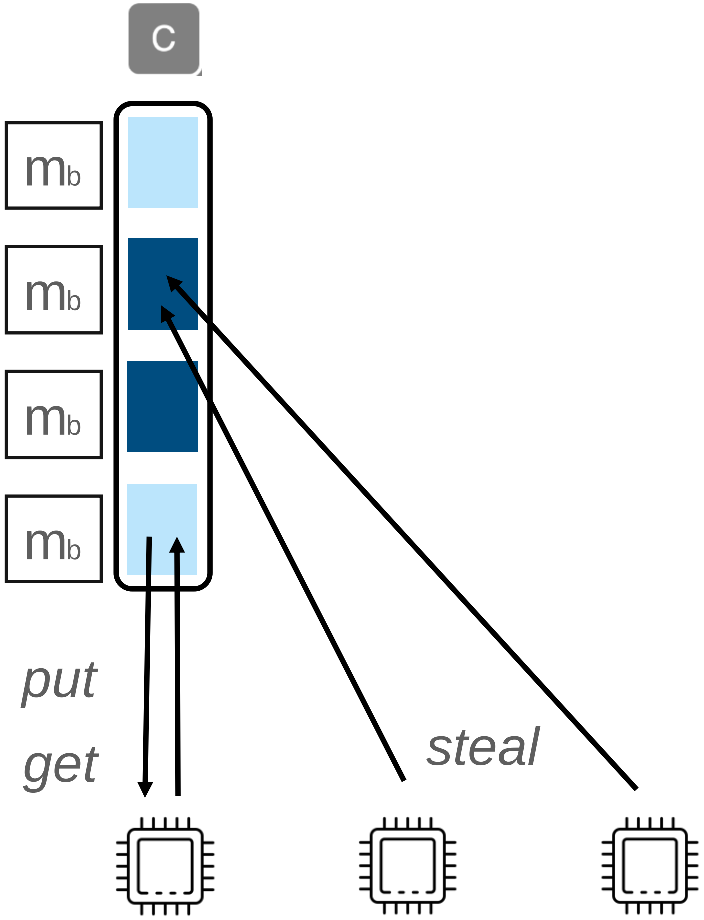
BWoS — Block-based Work Stealing



C) Cost of Interference with Thieves

- Fixed by Block-Level Synchronization and Randomized Stealing

BWoS — Block-based Work Stealing

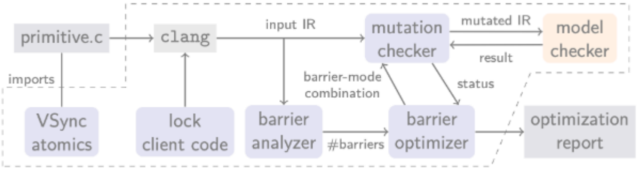


C) Cost of Interference with Thieves

- Fixed by Block-Level Synchronization and Randomized Stealing
- Thieves and the owner update different metadata, thus interference is reduced
- Thieves and the owner are likely to operate on different blocks

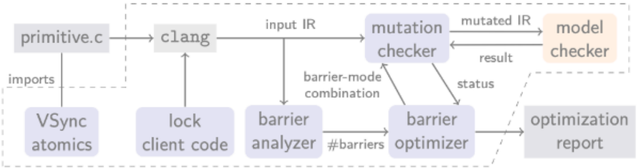
BWoS — Verification and Optimization

VSync Framework

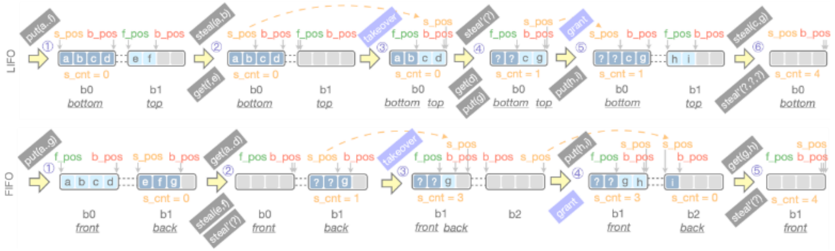


BWoS — Verification and Optimization

VSync Framework



Algorithms in C



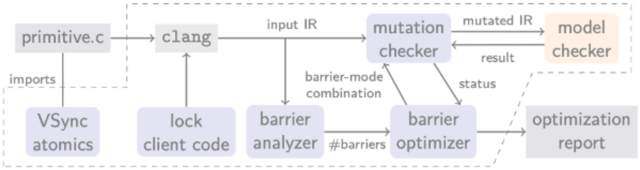
BWoS — Verification and Optimization

```
class stat {
 u64 sum = 0, buf = 1;
 void put(queue<u64> q){
   if (q.put(buf))
     sum += buf;
     buf <<= 1;
 }
 bool get(queue<u64> q){
   data = q.get(buf);
   if (data != null) {
     sum += data;
     return true;
   }
   return false;
 }
 void steal(queue<u64> q){
   data = q.steal(buf);
   if (data != null)
     sum += data;
  }
}
stat f, b, s1, s2;
queue<u64> q; // 2 * 2
T0: b.put(q)*3; f.get(q)*2;
    b.put(q)*4; f.get(q)*3;
    b.put(q)*5; f.get(q)*4;
T1: s1.steal(q);
T2: s2.steal(q)*2;
T3: while (f.get(q));
   assert (b.sum == f.sum +
               s1.sum + s2.sum);
(T0 ‖ T1 ‖ T2) ; T3
```

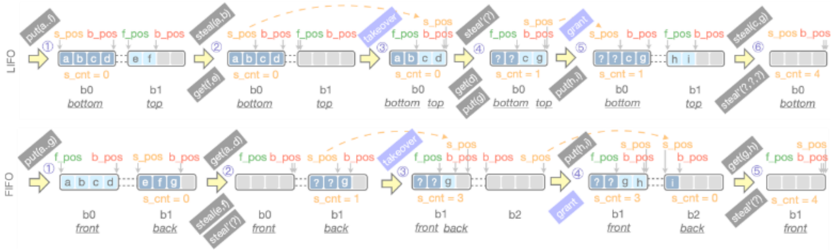
Client Code

trigger edge cases with assertions

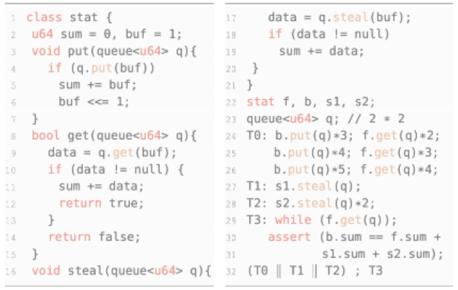
VSync Framework



Algorithms in C



BWoS — Verification and Optimization

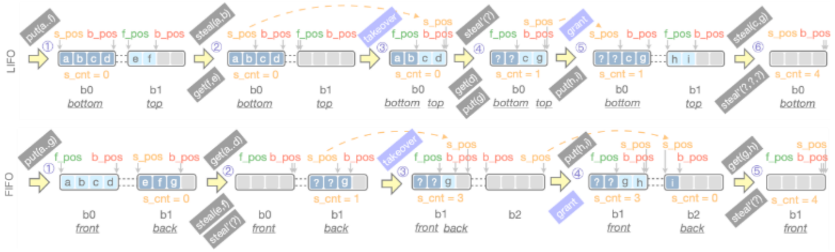


```
class stat {
 u64 sum = 0, buf = 1;
 void put(queue<u64> q){
   if (q.put(buf))
     sum += buf;
     buf <<= 1;
 }
 bool get(queue<u64> q){
   data = q.get(buf);
   if (data != null) {
     sum += data;
     return true;
   }
   return false;
 }
 void steal(queue<u64> q){
   data = q.steal(buf);
   if (data != null)
     sum += data;
 }
}
stat f, b, s1, s2;
queue<u64> q; // 2 * 2
T0: b.put(q)*3; f.get(q)*2;
    b.put(q)*4; f.get(q)*3;
    b.put(q)*5; f.get(q)*4;
T1: s1.steal(q);
T2: s2.steal(q)*2;
T3: while (f.get(q));
    assert (b.sum == f.sum +
        s1.sum + s2.sum);
(T0 ∥ T1 ∥ T2) ; T3
```

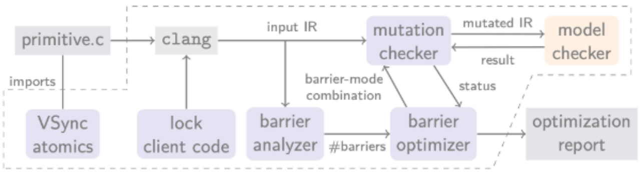
Client Code

trigger edge cases with assertions

Algorithms in C



VSync Framework



Verification

Properties:

- Memory safety
- Data race freedom
- Loop termination
- Consistency

WMM Optimization

	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*

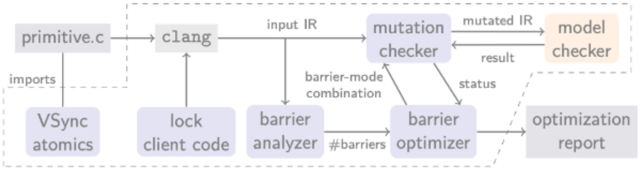
BWoS — Verification and Optimization

```
class stat {
 u64 sum = 0, buf = 1;
 void put(queue<u64> q){
   if (q.put(buf))
     sum += buf;
     buf <<= 1;
 }
 bool get(queue<u64> q){
   data = q.get(buf);
   if (data != null) {
     sum += data;
     return true;
   }
   return false;
 }
 void steal(queue<u64> q){
   data = q.steal(buf);
   if (data != null)
    sum += data;
  }
}
stat f, b, s1, s2;
queue<u64> q; // 2 * 2
T0: b.put(q)*3; f.get(q)*2;
     b.put(q)*4; f.get(q)*3;
     b.put(q)*5; f.get(q)*4;
T1: s1.steal(q);
T2: s2.steal(q)*2;
T3: while (f.get(q));
   assert (b.sum == f.sum +
              s1.sum + s2.sum);
(T0 ‖ T1 ‖ T2) ; T3
```

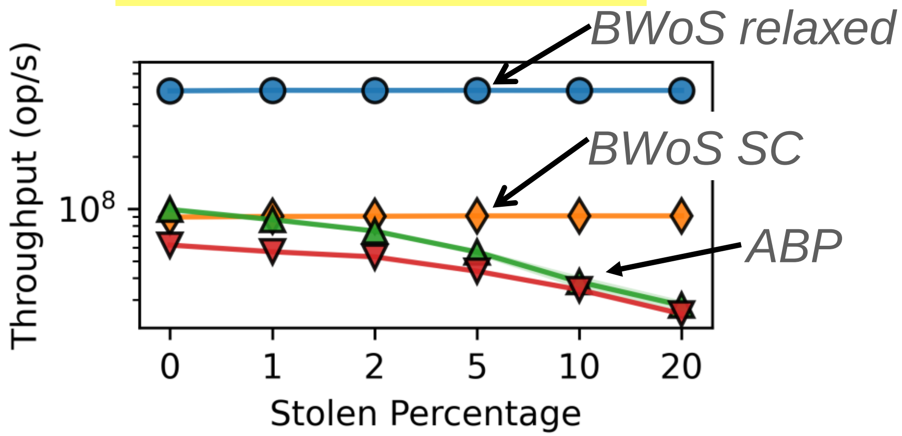
Client Code

trigger edge cases with assertions

VSync Framework



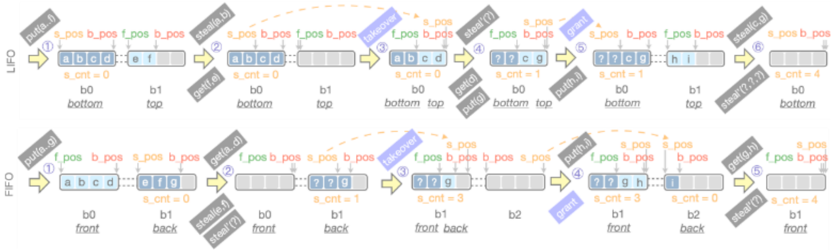
WMM Optimization Results



WMM Optimization

	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*

Algorithms in C



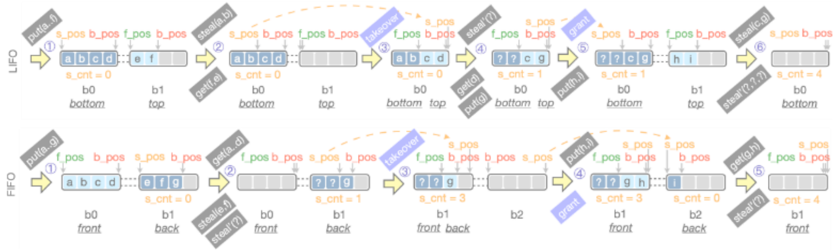
BWoS — Verification and Optimization

```
class stat {
 u64 sum = 0, buf = 1;
 void put(queue<u64> q){
   if (q.put(buf))
     sum += buf;
     buf <<= 1;
 }
 bool get(queue<u64> q){
   data = q.get(buf);
   if (data != null) {
     sum += data;
     return true;
   }
   return false;
 }
 void steal(queue<u64> q){
   data = q.steal(buf);
   if (data != null)
     sum += data;
  }
}
stat f, b, s1, s2;
queue<u64> q; // 2 * 2
T0: b.put(q)*3; f.get(q)*2;
    b.put(q)*4; f.get(q)*3;
    b.put(q)*5; f.get(q)*4;
T1: s1.steal(q);
T2: s2.steal(q)*2;
T3: while (f.get(q));
   assert (b.sum == f.sum +
           s1.sum + s2.sum);
(T0 ‖ T1 ‖ T2) ; T3
```

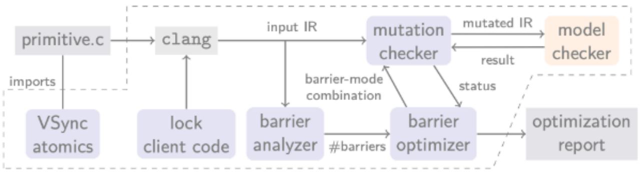
Client Code

trigger edge cases with assertions

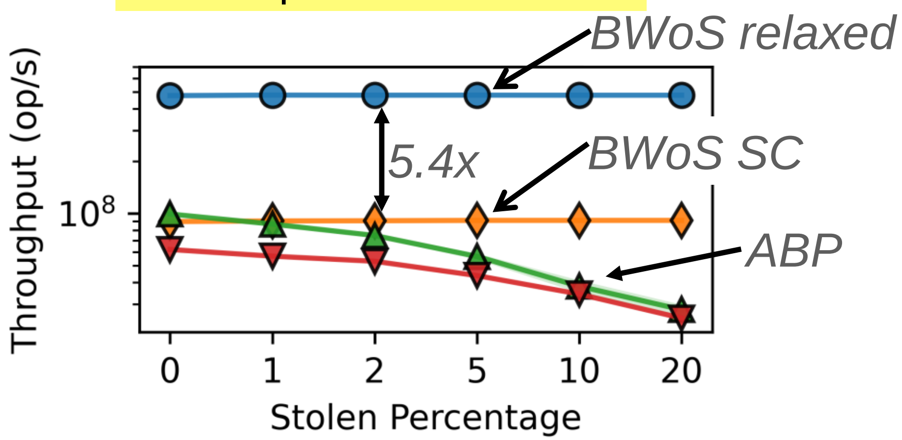
Algorithms in C



VSync Framework



WMM Optimization Results



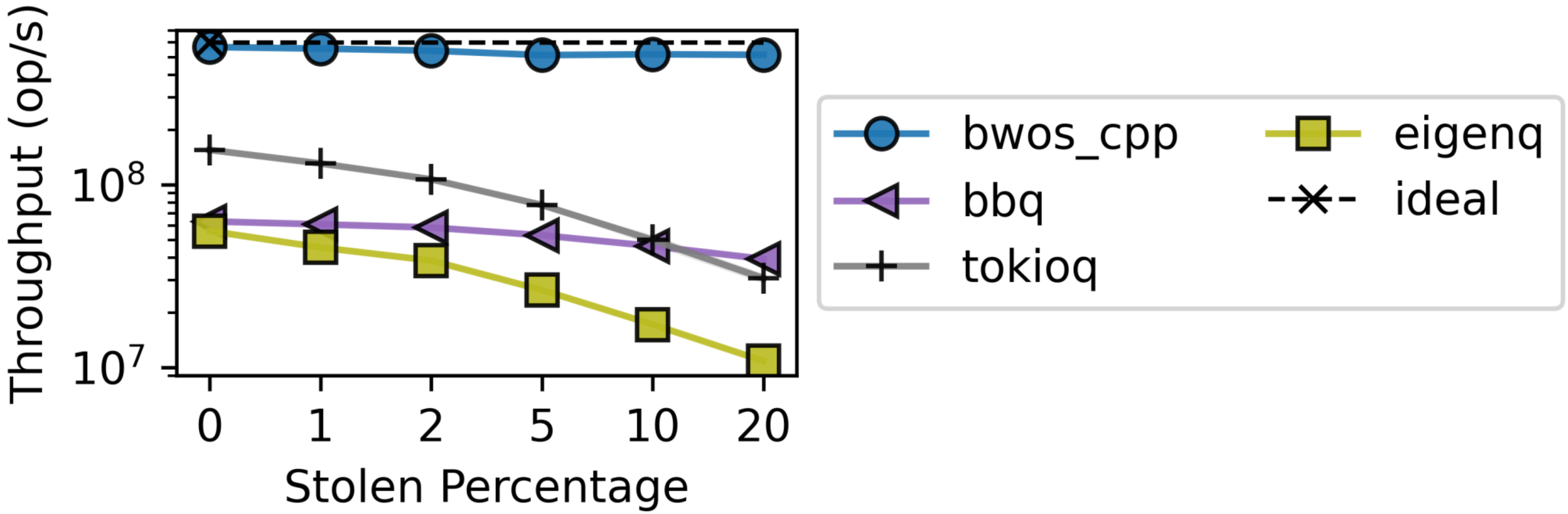
WMM Optimization

	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*

Micro-benchmark Results

Compared against state-of-the-art work-stealing queues

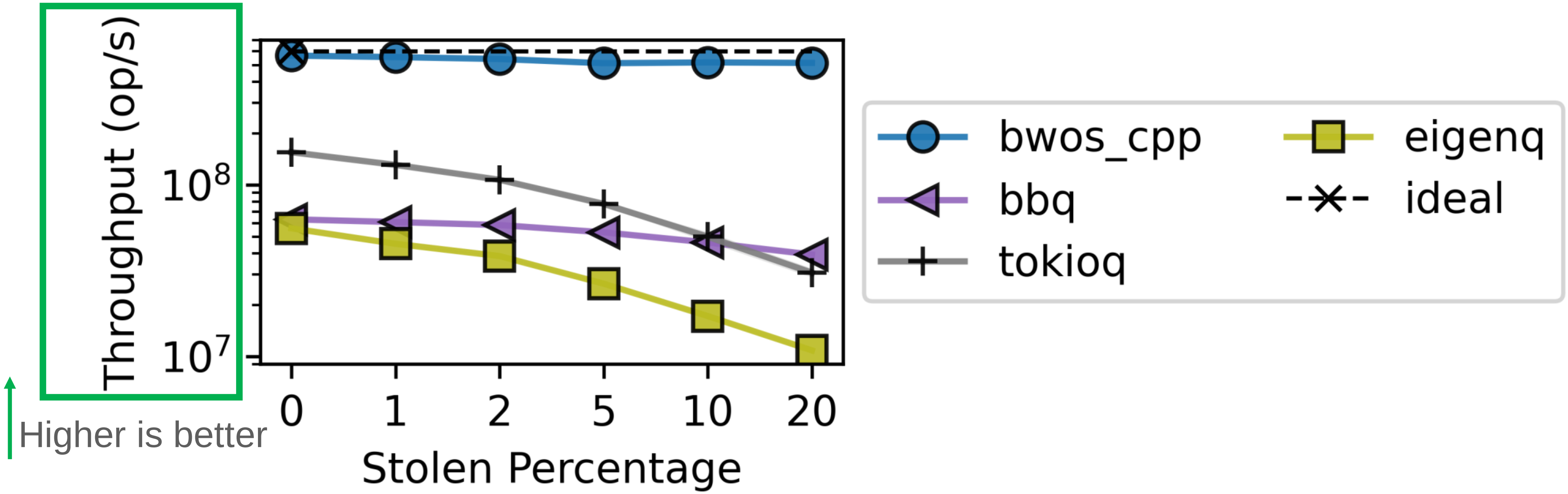
Each queue has a capacity of 8k entries, with 8-byte data items; BWoS is configured to have 8 blocks.



Micro-benchmark Results

Compared against state-of-the-art work-stealing queues

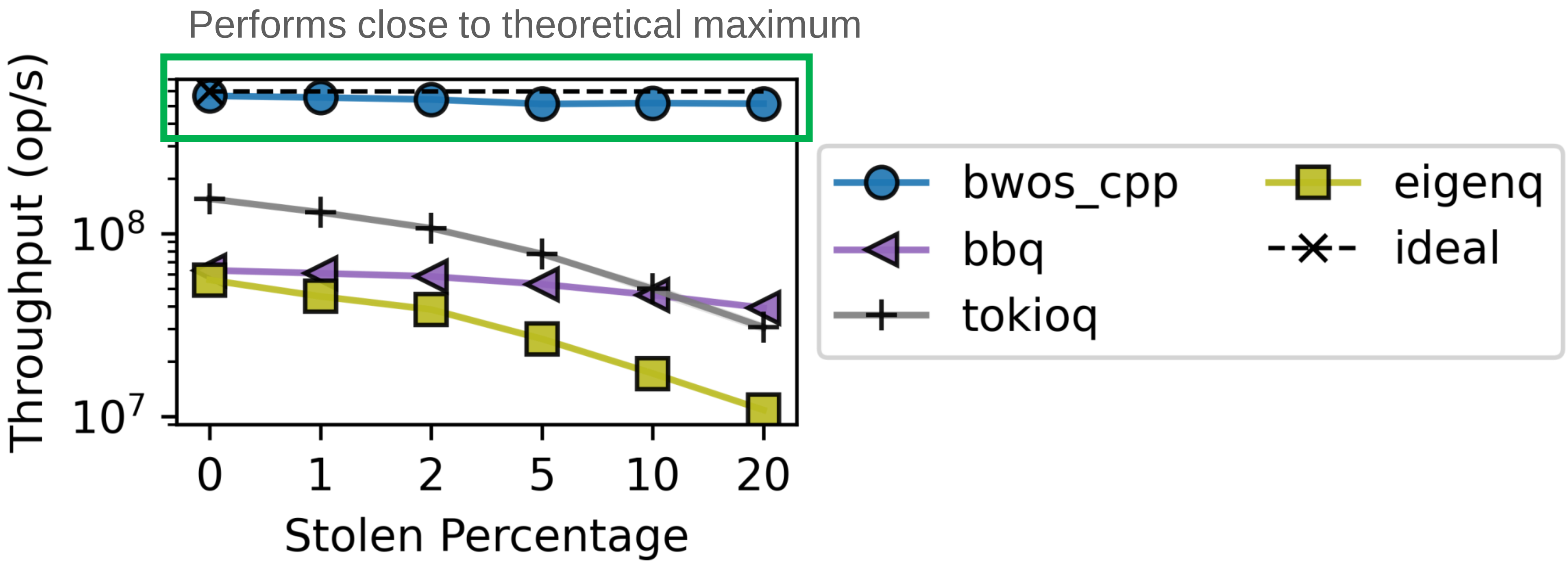
Each queue has a capacity of 8k entries, with 8-byte data items; BWoS is configured to have 8 blocks.



Micro-benchmark Results

Compared against state-of-the-art work-stealing queues

Each queue has a capacity of 8k entries, with 8-byte data items; BWoS is configured to have 8 blocks.

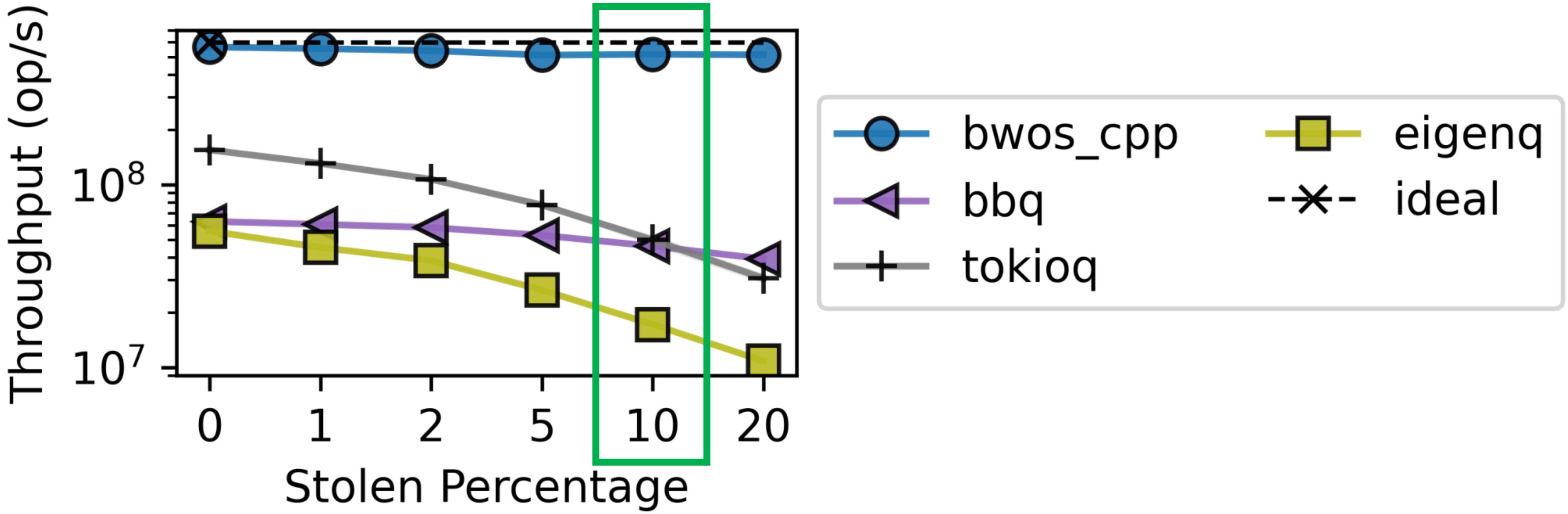


Micro-benchmark Results

Compared against state-of-the-art work-stealing queues

Each queue has a capacity of 8k entries, with 8-byte data items; BWoS is configured to have 8 blocks.

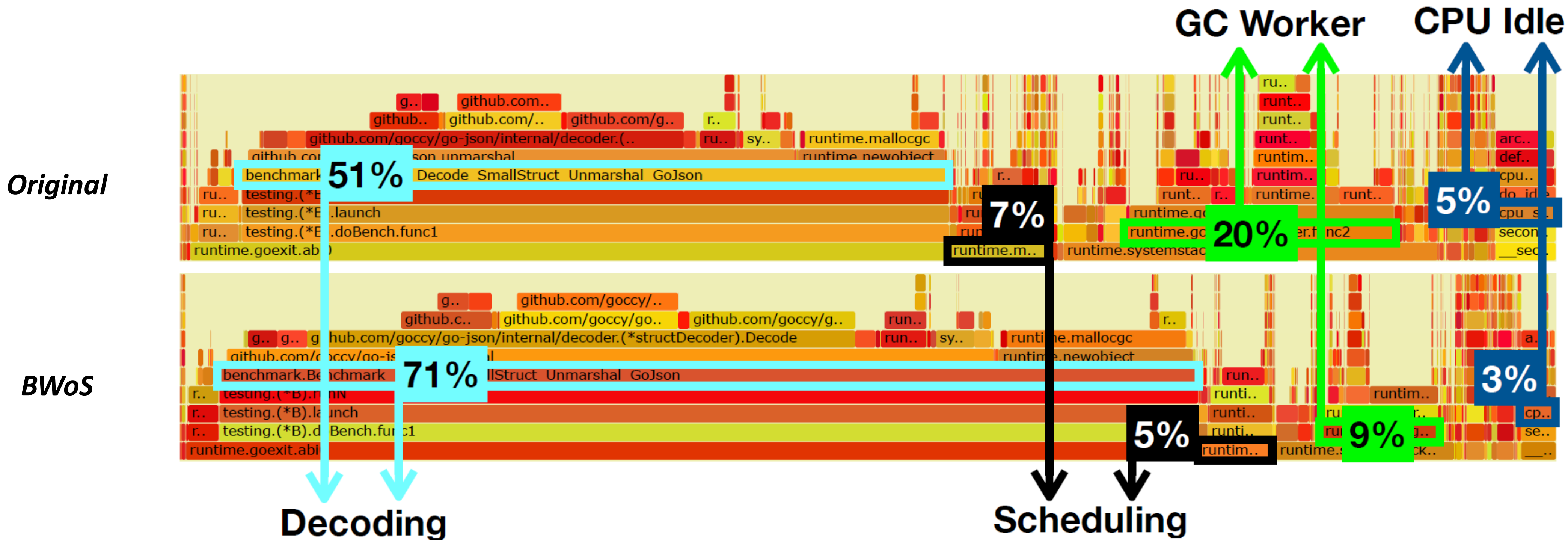
10% stolen: up 30x throughput



Summary: BWoS outperforms state-of-the-art queues by
1.6x – 10x without thieves, 1.6x - 30x with thieves.

BWoS in Go’s Runtime

GoJson Object Decoding Benchmark

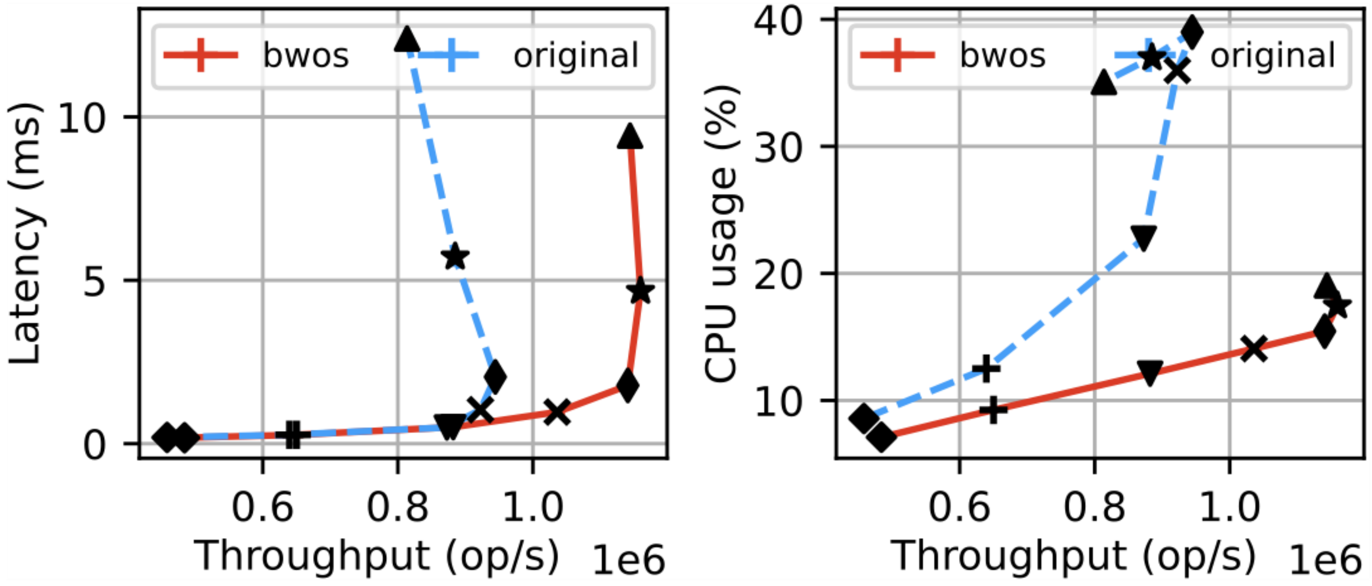


Summary: GoJson benchmarks experience 28.2% speedup on average for Arm (see paper).
BWoS improves performance of real-world computational workloads.

BWoS in Rust Tokio Runtime

We replace the run queue in Rust Tokio with FIFO BWoS

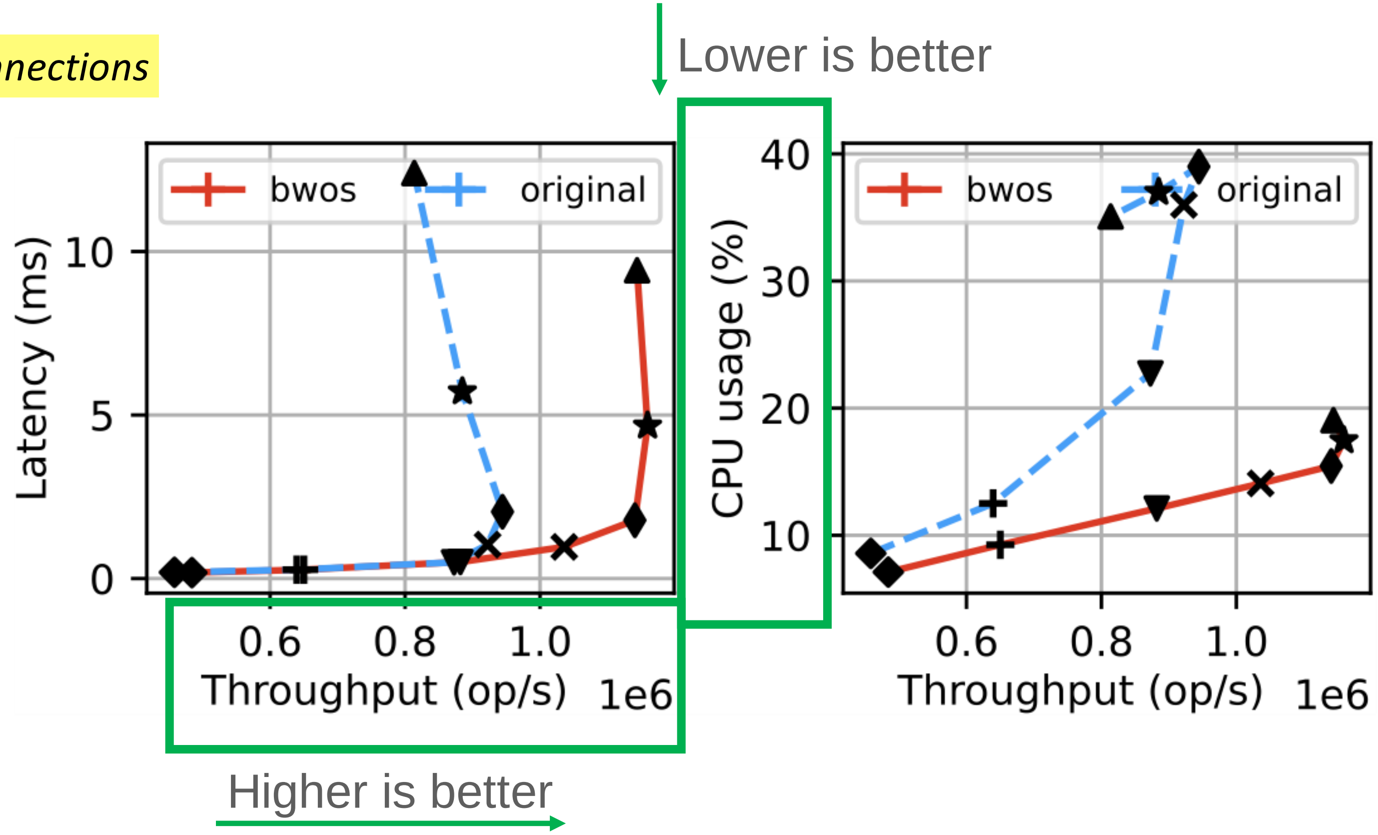
Hyper HTTP server, 1k connections



BWoS in Rust Tokio Runtime

We replace the run queue in Rust Tokio with FIFO BWoS

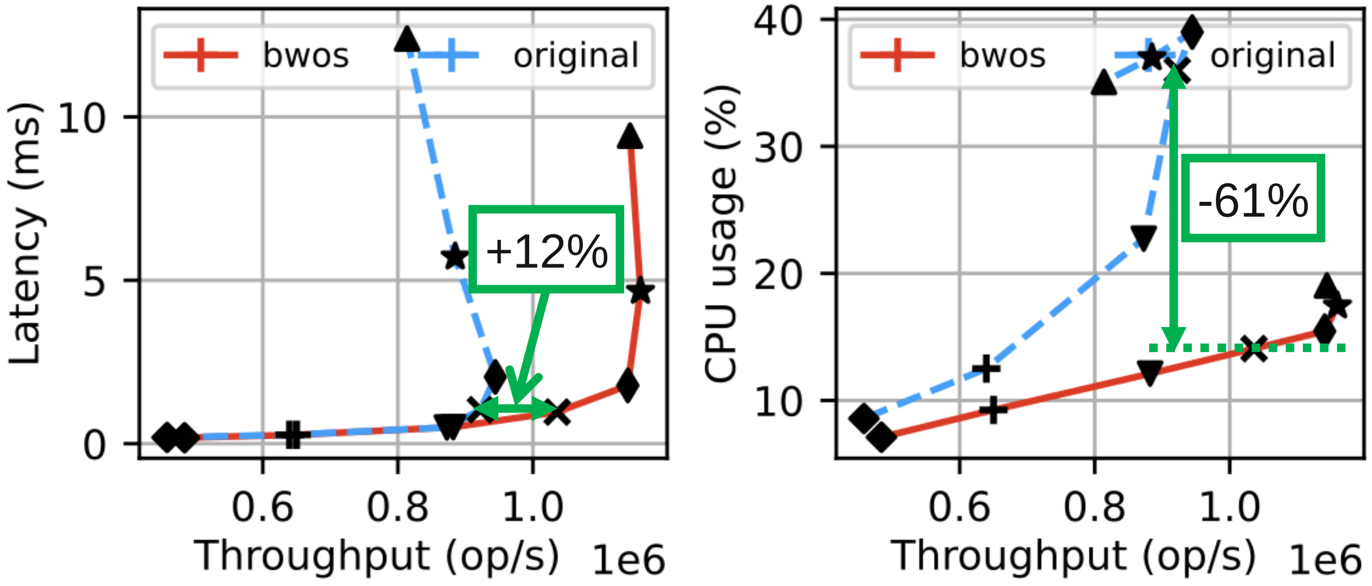
Hyper HTTP server, 1k connections



BWoS in Rust Tokio Runtime

We replace the run queue in Rust Tokio with FIFO BWoS

Hyper HTTP server, 1k connections



Summary: BWoS increases throughput by 12% with 7% lower latency and 61% lower CPU utilization.
BWoS improves performance of real-world IO servers.

We have published our changes for the Tokio runtime: https://github.com/tokio-rs/tokio/pull/5283

Summary

- The benefit of the block-based design is manyfold, and can be applied in many concurrent algorithms:
  - BWoS: Work Stealing (this work)
  - BBQ: Producer-Consumer Queues (ATC’22)

Summary

- The benefit of the block-based design is manyfold, and can be applied in many concurrent algorithms:
  - BWoS: Work Stealing (this work)
  - BBQ: Producer-Consumer Queues (ATC’22)

- Verified software can be faster than unverified software.

Summary

- The benefit of the block-based design is manyfold, and can be applied in many concurrent algorithms:
    - BWoS: Work Stealing (this work)
    - BBQ: Producer-Consumer Queues (ATC'22)

- Verified software can be faster than unverified software.

Thanks!