Blocking versus Non-Blocking Shared-Memory Multicore Synchronization: Programmability, Scalability and Performance

Shinhyung Yang[1], Seongho Jeong[1], Byunguk Min[1], Yeonsoo Kim[1], Bernd Burgstaller[1] and Johann Blieberger[2]

[1]Department of Computer Science, Yonsei University, Korea
[2]Institute of Computer Engineering, Automation Systems Group, TU Wien, Austria

June 13, 2019

Table of Contents

Synchronization

- mutual exclusion locks

Synchronization

- mutual exclusion locks
- Ada's protected objects (POs)

Synchronization

- mutual exclusion locks
- Ada's protected objects (POs)
- Entries and procedures of a PO execute one after another

Synchronization

- mutual exclusion locks
- Ada's protected objects (POs)
- Entries and procedures of a PO execute one after another
- makes it straight-forward for programmers to reason about updates to the shared data encapsulated by a PO

Synchronization

- mutual exclusion locks
- Ada's protected objects (POs)
- Entries and procedures of a PO execute one after another
- makes it straight-forward for programmers to reason about updates to the shared data encapsulated by a PO
- mutual-exclusion property of (highly-contended) **locks** stands in the way to **scalability of parallel programs** on many-core architectures

Synchronization

- mutual exclusion locks
- Ada's protected objects (POs)
- Entries and procedures of a PO execute one after another
- makes it straight-forward for programmers to reason about updates to the shared data encapsulated by a PO
- mutual-exclusion property of (highly-contended) **locks** stands in the way to **scalability of parallel programs** on many-core architectures
- locks do not allow **progress guarantees**, because a task may fail inside a critical section, e.g., by entering an endless loop, and thereby prevent other tasks from accessing shared data

Synchronization

- mutual exclusion locks
- Ada's protected objects (POs)
- Entries and procedures of a PO execute one after another
- makes it straight-forward for programmers to reason about updates to the shared data encapsulated by a PO
- mutual-exclusion property of (highly-contended) **locks** stands in the way to **scalability of parallel programs** on many-core architectures
- locks do not allow **progress guarantees**, because a task may fail inside a critical section, e.g., by entering an endless loop, and thereby prevent other tasks from accessing shared data
- $\Rightarrow$ allow method calls to **overlap** in time

Lock Elision

Protecting shared-data with POs

- A coarsed-grained lock is prone to task serialization
- A fine-grained locking is error-prone and complex
- Lock elision reduces serialization with lock-based code

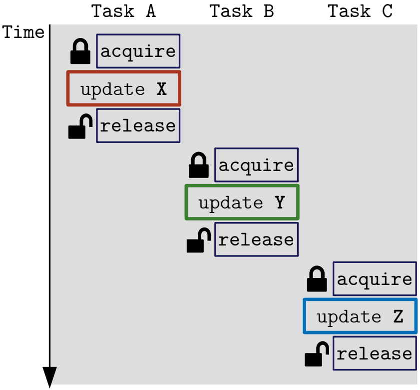


Figure : Task serialization with locks

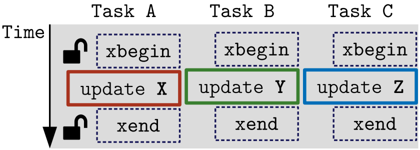


Figure : Lock elision with *Intel TSX*

```
function xbegin return uint32;
procedure xend;
function xtest return uint32;
procedure xabort;
```

Listing 1: TSX lock elision intrinsics

Adapting GNU Ada Run-time Library (GNARL)

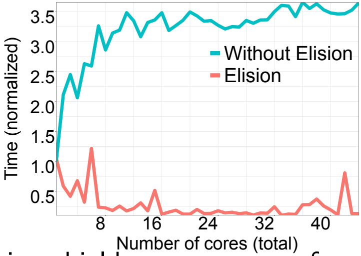
- GNARL employs one POSIX lock per PO for synchronization
- Lock elision is incorporated into Write_Lock

```
procedure Write_Lock         -- GNARL lock acquisition procedure
  result := Try_Elision      -- Attempt lock elision
  if result = fail then      -- If failed:
    acquire PO.lock          --   fall-back to acquire POSIX lock
  end if
  return
end Write_Lock
```

Adapted Write_Lock

1. Invoke Try_Elision
2. If failed:
   - Fall-back to default routine to acquire POSIX lock

Summary: Lock-elision of POs



Ex: hash table

- Pros:
  - PO lock-elision shields programmers from non-blocking synchronization problem
  - high scalability in case of low probability of data-conflicts (e.g., hash-table)
- Cons:
  - requires HW support to be efficient. Not mainstream yet (e.g., not available on ARM platform)
  - Intel TSX requires fallback-path (lock)
  - Intel TSX capacity overflows with large amount of shared data (e.g., linked lists)
    - not generally applicable to all types of data structures

Sequential Consistency

- allow method calls to **overlap** in time

Sequential Consistency

- allow method calls to **overlap** in time
- synchronization on a **finer granularity** within a method's code, via atomic *read-modify-write* (RMW) operations

Sequential Consistency

- allow method calls to **overlap** in time
- synchronization on a **finer granularity** within a method's code, via atomic *read-modify-write* (RMW) operations
- atomic operations are provided either by the CPU's instruction set architecture (ISA), or the language run-time (with the help of the CPU's ISA)

Sequential Consistency

- allow method calls to **overlap** in time
- synchronization on a **finer granularity** within a method's code, via atomic *read-modify-write* (RMW) operations
- atomic operations are provided either by the CPU's instruction set architecture (ISA), or the language run-time (with the help of the CPU's ISA)
- e.g., CAS compare&swap operation

Sequential Consistency

- allow method calls to **overlap** in time
- synchronization on a **finer granularity** within a method's code, via atomic *read-modify-write* (RMW) operations
- atomic operations are provided either by the CPU's instruction set architecture (ISA), or the language run-time (with the help of the CPU's ISA)
- e.g., CAS compare&swap operation
- **sequential consistency** ensures that method calls act as if they occurred in a **sequential, total order** that is **consistent with the program order** of each participating task

Non-blocking Synchronization Techniques

- difficult to implement

Non-blocking Synchronization Techniques

- difficult to implement
- the design of non-blocking data structures is an area of active research

Non-blocking Synchronization Techniques

- difficult to implement
- the design of non-blocking data structures is an area of active research
- a programming language must provide a strict memory model

Lock-free Synchronization – Example

```
-- Initial values:
Flag := False;
Data := 0;
```

```
-- Task 1:
Data := 1;
Flag := True;
```

Lock-free Synchronization – Example

```
-- Initial values:
Flag := False;
Data := 0;
```

```
-- Task 1:
Data := 1;
Flag := True;
```

```
-- Task 2:
loop
  R1 := Flag;
  exit when R1;
end loop;
R2 := Data;
```

Lock-free Synchronization – Example

```
1 -- Initial values:
2 Flag := False;
3 Data := 0;
```

```
1 -- Task 1:
2 Data := 1;
3 Flag := True;
```

```
1 -- Task 2:
2 loop
3   R1 := Flag;
4   exit when R1;
5 end loop;
6 R2 := Data;
```

store–store re-ordering of the assignments in lines 2 and 3 of Task 1

$\Rightarrow$ reading `R2 = 0` in Line 6 of Task 2.

Lock-free Synchronization – Example

```
-- Initial values:
Flag := False;
Data := 0;
```

```
1 -- Task 1:
2 Data := 1;
3 Flag := True;
```

```
1 -- Task 2:
2 loop
3   R1 := Flag;
4   exit when R1;
5 end loop;
6 R2 := Data;
```

store–store re-ordering of the assignments in lines 2 and 3 of Task 1

⇒ reading `R2 = 0` in Line 6 of Task 2.

```
Data : Integer with Volatile; -- Ada2012
Flag : Boolean with Atomic; -- Ada2012
```

Ada's Volatile Variables

- guarantee that all tasks agree on the same order of updates

Ada's Volatile Variables

- guarantee that all tasks agree on the same order of updates
- $\Rightarrow$ sequentially consistent

Ada's Volatile Variables

- guarantee that all tasks agree on the same order of updates
- $\Rightarrow$ sequentially consistent
- however: relaxing SC for the sake of performance on contemporary CPU architectures

Concurrent Objects

J. Blieberger, B. Burgstaller, *Safe Non-blocking Synchronization in Ada 202x*, Ada-Europe'18, 2018, p. 53–69

Concurrent Objects

J. Blieberger, B. Burgstaller, *Safe Non-blocking Synchronization in Ada 202x*, Ada-Europe'18, 2018, p. 53–69

- support for weak memory model

Concurrent Objects

J. Blieberger, B. Burgstaller, *Safe Non-blocking Synchronization in Ada 202x*, Ada-Europe'18, 2018, p. 53–69

- support for weak memory model
- for non-blocking synchronization

Concurrent Objects

J. Blieberger, B. Burgstaller, *Safe Non-blocking Synchronization in Ada 202x*, Ada-Europe'18, 2018, p. 53–69

- support for weak memory model
- for non-blocking synchronization
- for synchronization on a finer granularity (RMW operations)
- encapsulation of non-blocking synchronization by high-level language construct
    - what protected objects (POs) do for blocking synchronization

Example – Generic Release-Acquire Object (1/2)

```
generic
  type Data is private;
package Generic_Release_Acquire is

  concurrent RA
  is
    procedure Write (d: Data);
    entry Get (D: out Data);
  private
    Ready: Boolean := false with Synchronized,
      Memory_Order_Read => Acquire,
      Memory_Order_Write => Release;
    Da: Data;
  end RA;

end Generic_Release_Acquire;
```

Example – Generic Release-Acquire Object (2/2)

```
package body Generic_Release_Acquire is

  concurrent body RA is

    procedure Write (D: Data) is
    begin
      Da := D;
      Ready := true;
    end Write:

    entry Get (D: out Data)
      until Ready is
      -- spin-lock until released, i.e., Ready = true;
      -- only sync. variables and constants allowed
      -- in guard expression
    begin
      D := Da;
    end Get;
  end RA;

end Generic_Release_Acquire;
```

Example Lock-free Stack (1/2)

```
subtype Data is Integer;

type List;
type List_P is access List;
type List is
  record
    D: Data;
    Next: List_P;
  end record;

Empty: exception;

concurrent Lock_Free_Stack
is
  entry Push(D: Data);
  entry Pop(D: out Data);
private
  Head: List_P with Read_Modify_Write,
    Memory_Order_Read => Relaxed,
    Memory_Order_Write_Success => Release,
    Memory_Order_Write_Failure => Relaxed;
end Lock_Free_Stack;
```

Example Lock-free Stack (2/2)

```
concurrent body Lock_Free_Stack is
  entry Push (D: Data)
      until Head = Head'OLD is
    New_Node: List_P := new List;
  begin
    New_Node.all := (D => D, Next => Head);
    Head := New_Node; -- RMW
  end Push;

  entry Pop(D: out Data)
      until Head = Head'OLD is
    Old_Head: List_P;
  begin
    Old_Head := Head;
    if Old_Head /= null then
      Head := Old_Head.Next; -- RMW
      D := Old_head.D;
    else
      raise Empty;
    end if;
  end Pop;
end Lock_Free_Stack;
```

Benchmark Configuration

- **Platform 1:** 2 CPU Intel Xeon E5-2697 v3 system
  - 14 x86_64 cores per CPU
- **Platform 2:** 4 CPU AWS Graviton on Amazon AWS
  - 4 ARMv8 cores per CPU
- Scalability experiment policies:
  - One Ada task assigned per core
  - Cores of a CPU populated consecutively
  - Once all cores of a CPU are populated, the next CPU receives tasks
- Tasks run synchronization-constructs in **tight-loop**
  - Incurs high contention
  - Brings out the best in each synchronization-construct :)

COstack (1/2)

- **COstack**: Non-blocking stack using `C++11 atomic` library with weaker memory model
- Function `bool node.compare_exchange_strong(type *expected, type *desired)`
  - **Success**: If the `node` has not been changed by other threads, then the `node` is atomically changed to `desired`, and returns `true`
  - **Fail**: If the value of `node` has changed by other threads, then the `expected` is atomically changed to `node`, and returns `false`
- **Push**: RMW spins in the `while` loop until `head` is changed to `new_node` and returns `true`

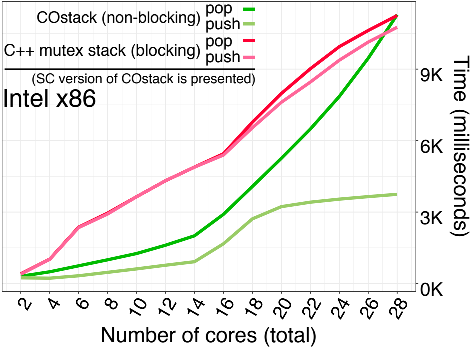
```
void push(T const &data) {
  node *const new_node = new node(data);
  new_node->next = head.load(std::memory_order_acquire);
  while (!head.compare_exchange_strong(new_node->next,
      new_node)) ; // RMW
}
```

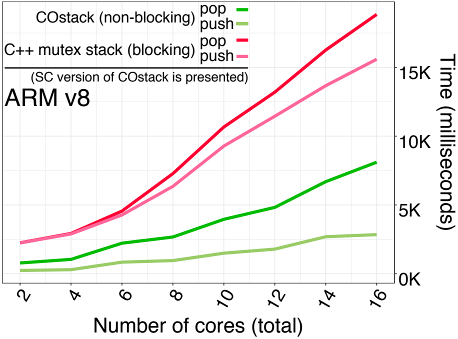
COstack (2/2)

- **Pop**: `old_head` is in `while` loop until `old_head` is updated to the latest `head` variable.

```
void pop() {
  node *old_head = head.load();
  do {
    node *temp;
    do {
      temp = old_head;
      old_head = head.load();
    } while (old_head != temp);
  } while (old_head &&
      !head.compare_exchange_strong(old_head, old_head->next))
          ;
}
```

Non-blocking vs. Blocking Stacks





- non-blocking COstack mimics Concurrent Objects proposed for Ada202x
- actually implemented in C++
- COstack performs better than blocking C++ mutex stack
- more performance gains observed on ARMv8

Locks: AdaPO-lock vs. POSIX & C++ Mutexes

- PO-lock using PO's monitor-style synchronization:

```
protected body Lock is
  procedure CriticalSection is
  begin
    null; -- Critical section code here...
  end CriticalSection;
end Lock;
```

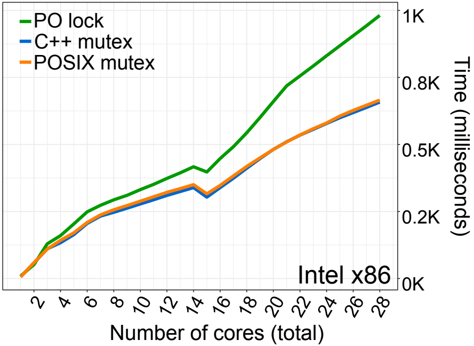
- POSIX mutex consisting of lock-unlock primitives:

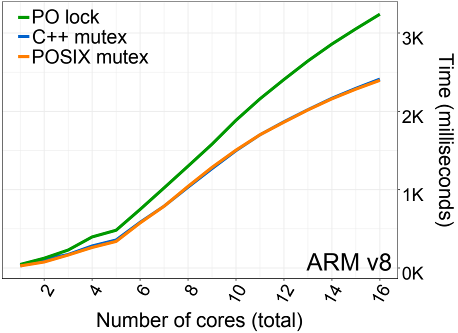
```
pthread_mutex_t mutex;

pthread_mutex_lock(&mutex);
  // Critical section code here...
pthread_mutex_unlock(&mutex);
```

- C++ mutex:

```
std::mutex mtx;

mtx.lock();
  // Critical section code here...
mtx.unlock();
```

Performance: PO-lock, POSIX & C++ Mutexes





- All three locks provide mutual exclusion but no fairness
  - (The egg-shell model applies only to entries)
- PO adds noticeable performance-overhead compared to "plain" mutexes
  - Programmer-supplied PO configuration mechanism could help to avoid this overhead

Ada_TAS_MPMC (Multi-Producer-Multi-Consumer Queue) (1/2)

- GCC atomic intrinsics are used for synchronization instead of protected objects
- `type __sync_lock_test_and_set_4 (type *ptr, type value)`
  ⇒ It atomically writes `value` into `*ptr`, and return the previous contents of `*ptr`. Size of `*ptr` and `value` is 4 byte each.

```
function TAS_4(
   current: access Unsigned;
   Newval: Unsigned) return Unsigned;
   pragma Import (Intrinsic, TAS_4,
       "__sync_lock_test_and_set_4");
```

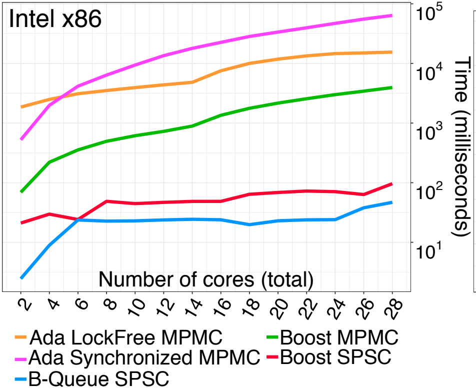
- `__sync_lock_release_4 (type *ptr)`
  ⇒ It releases the lock acquired by `__sync_lock_test_and_set`. It atomically writes constant 0 to `*ptr`

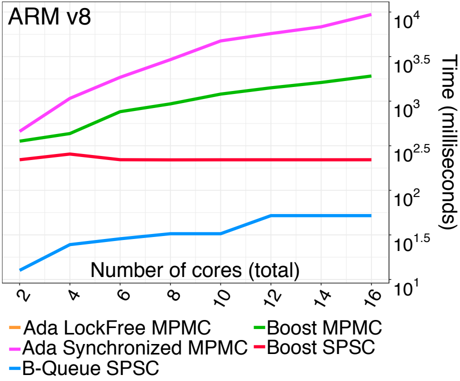
Ada_TAS_MPMC (2/2)

- **Lock & Unlock**: `Q.Locked` is initialized as 0. By `TAS_4` on `Q.Locked`, lock is acquired. $\Rightarrow$ By writing 0 to `Q.Locked`, lock is released.
- **Producer**: It enqueues to a queue `Q` for `iter` times.
- **Consumer**: It dequeues element from `Q` to `Data` for `iter` times. If `Q` is empty, release the lock, and redo inner loop.

```ada
Q: My_USQ.Queue;
task body Producer is
begin
  for I in 1..iter loop
    while TAS_4(Q.Locked'Access,1)=1 loop -- Acquire
      null;
    end loop;

    Q.Enqueue(New_Item => i);

    Lock_Release_4(Q.Locked'Access);       -- Release
  end loop;
```

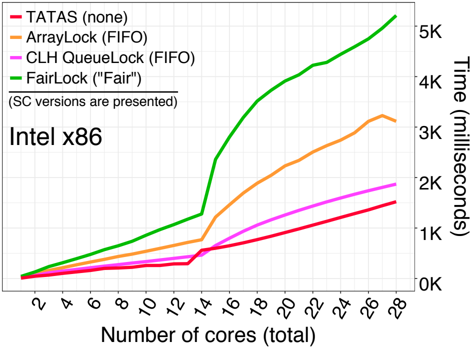
Non-blocking vs. Blocking Queues

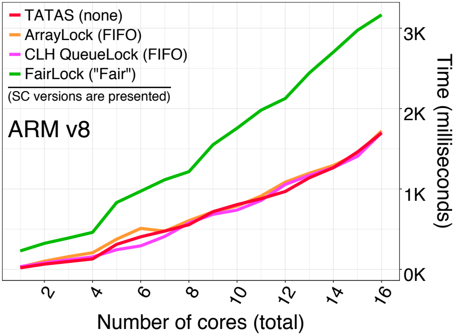




- SPSC-queues: lower sync. overhead, higher performance
- Boost MPMC and Ada LockFree MPMC constitute identical concurrent data-structures (Algorithm of Michael & Scott)
- Blocking MPMCs (Ada Synchronized, Ada TAS): clear performance disadvantage

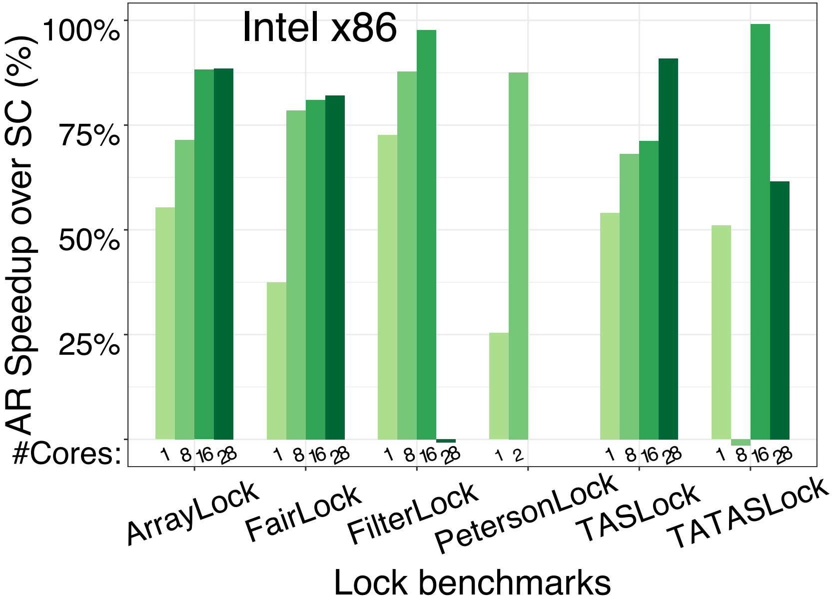
Non-blocking Locks and Progress Guarantees



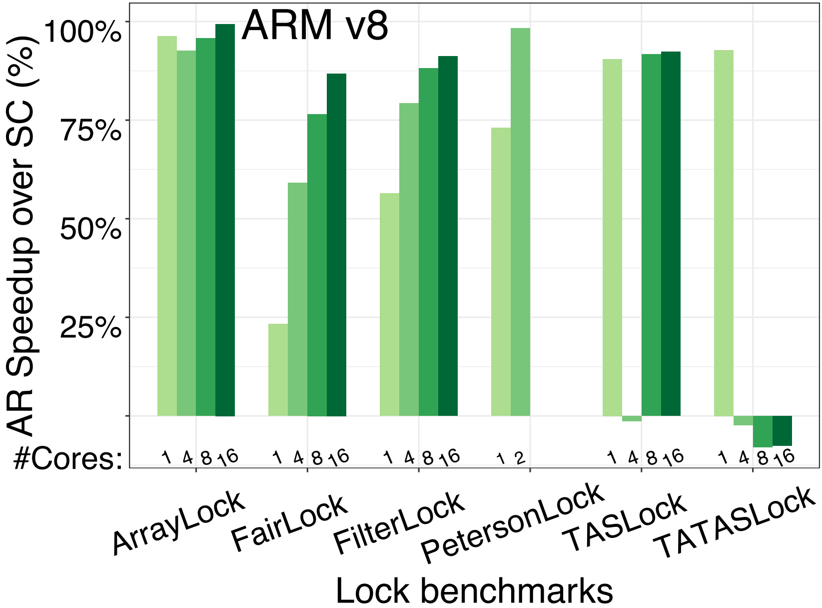


- TATAS-lock permits **starvation**
  - Optimizes throughput of task-ensemble as a whole
- CLH QueueLock and ArrayLock: **FIFO progress-guarantee**
- FairLock: no task can enter critical section twice while another task is waiting
  - Weaker guarantee than FIFO

Performance gain ratio: AR over SC (1/3)



Performance gain ratio: AR over SC (2/3)



Performance gain ratio: AR over SC (3/3)

- took considerable time and a lot of discussions to convert from SC to AR version

Performance gain ratio: AR over SC (3/3)

- took considerable time and a lot of discussions to convert from SC to AR version
- the same or even more to relax acquire and release operations (here we got real performance gains)

Performance gain ratio: AR over SC (3/3)

- took considerable time and a lot of discussions to convert from SC to AR version
- the same or even more to relax acquire and release operations (here we got real performance gains)
- some C++ memory fences turned out to be insufficient

Conclusions and Future Work (1/2)

- Ada's protected objects have advantages:

Conclusions and Future Work (1/2)

- Ada's protected objects have advantages: (1) safe

Conclusions and Future Work (1/2)

- Ada's protected objects have advantages: (1) safe (2) easy to use

Conclusions and Future Work (1/2)

- Ada's protected objects have advantages: (1) safe (2) easy to use (3) easy to comprehend.

Conclusions and Future Work (1/2)

- Ada's protected objects have advantages: (1) safe (2) easy to use (3) easy to comprehend.
- If performance is a major concern, lock-free implementations outperform the blocking approach.

Conclusions and Future Work (1/2)

- Ada's protected objects have advantages: (1) safe (2) easy to use (3) easy to comprehend.
- If performance is a major concern, lock-free implementations outperform the blocking approach.
- To gain higher performance for Ada 2012 $\Rightarrow$ intrinsics or machine code insertions.

Conclusions and Future Work (2/2)

- Simply replacing PO locks with spin loops is not enough.

Conclusions and Future Work (2/2)

- Simply replacing PO locks with spin loops is not enough.
- Concurrent execution inside the methods is better.

Conclusions and Future Work (2/2)

- Simply replacing PO locks with spin loops is not enough.
- Concurrent execution inside the methods is better.
- For Ada 2012 programs $\Rightarrow$ calls to intrinsics.

Conclusions and Future Work (2/2)

- Simply replacing PO locks with spin loops is not enough.
- Concurrent execution inside the methods is better.
- For Ada 2012 programs $\Rightarrow$ calls to intrinsics.
- $\Rightarrow$ adding *concurrent objects (COs)* and a strict memory consistency model to Ada 202x.

Conclusions and Future Work (2/2)

- Simply replacing PO locks with spin loops is not enough.
- Concurrent execution inside the methods is better.
- For Ada 2012 programs $\Rightarrow$ calls to intrinsics.
- $\Rightarrow$ adding *concurrent objects (COs)* and a strict memory consistency model to Ada 202x.
- Future container libraries for Ada 202x should also include lock-free and wait-free data-structures.

Conclusions and Future Work (2/2)

- Simply replacing PO locks with spin loops is not enough.
- Concurrent execution inside the methods is better.
- For Ada 2012 programs $\Rightarrow$ calls to intrinsics.
- $\Rightarrow$ adding *concurrent objects (COs)* and a strict memory consistency model to Ada 202x.
- Future container libraries for Ada 202x should also include lock-free and wait-free data-structures.
- Set up benchmarks for reuse.

Conclusions and Future Work (2/2)

- Simply replacing PO locks with spin loops is not enough.
- Concurrent execution inside the methods is better.
- For Ada 2012 programs $\Rightarrow$ calls to intrinsics.
- $\Rightarrow$ adding *concurrent objects (COs)* and a strict memory consistency model to Ada 202x.
- Future container libraries for Ada 202x should also include lock-free and wait-free data-structures.
- Set up benchmarks for reuse.
- Prototype implementation for COs.