Design-space evaluation for non-blocking synchronization in Ada: lock elision of protected objects, concurrent objects, and low-level atomics

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

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

Technical Report TR-0004

Embedded Systems Languages and Compilers Lab
Department of Computer Science
Yonsei University

February 10, 2020

Abstract

The current ISO/IEC standard of the Ada programming language does not support non-blocking synchronization. The restriction to locks conflicts with Ada's design goals for program *safety* and *efficiency*, because (1) task failure inside a critical section may incur deadlock, and (2) locks stand in the way of scaling parallel programs on multicore architectures. Increased autonomy of software systems and advances in embedded multicore platforms make non-blocking synchronization a desirable feature for Ada, which is traditionally employed for safety-critical embedded applications in the automotive and aerospace domains.

We propose two techniques to support non-blocking synchronization in Ada: (1) Lock elision of Ada's Hoare-style monitor synchronization construct (called "protected object") allows method calls of the monitor to overlap in time; inter-thread data conflicts are resolved by underlying hardware transactional memory. (2) Concurrent objects constitute a novel programming primitive to encapsulate the complexity of non-blocking synchronization in a language-level construct. We investigate the use of an alternative, low-level API that employs atomic read–modify–write operations in the style of C++11, in conjunction with relaxed memory consistency models.

We conduct an extensive experimental evaluation on an x86- and an ARM v8 multicore platform to explore the trade-offs of the proposed designs with respect to programmability, scalability and performance; and evaluate the performance improvements achievable with relaxed memory consistency models. We include a comparison with state-of-the-art blocking synchronization constructs.

This paper extends work published at the Ada-Europe 2017 and 2018 conferences [51, 25]: we have revised and extended the definition of concurrent objects. We employ concurrent objects with the design of a non-blocking stack based on hazard pointers, which we compare to low-level atomics in scalability, performance, and programmability. We provide quantitative data on the performance of blocking and non-blocking synchronization constructs and performance gains through relaxed memory consistency models.

1 Introduction

Shared-memory multicore architectures have become the de-facto standard for modern computer systems, ranging from resource-constraint embedded and mobile devices to desktops, servers, and Cloud- and high-performance computers [41, 38, 67].

Multi-threaded software requires synchronization to protect data shared among multiple threads. Locks allow the transformation of a block of code into a critical section, which can only be executed by one thread at a time. Employing locks to achieve mutual exclusion is well-understood and the most prevalent form of synchronization. However, because threads serialize to gain access to shared data, locks negatively impact performance and hamper scalability on multicore architectures [70].

A fundamental limitation of lock-based synchronization is the impossibility of progress guarantees: a thread that fails inside a critical section—e.g., by entering an endless loop—will prevent other threads from accessing shared data. Page faults, context switches, and cache misses inside a critical section may delay the progress of an entire thread ensemble for a potentially unbounded amount of time. So lock-based synchronization allows system-states where a thread is unable to make progress without the cooperation of its peers. A lock is potentially *blocking*, irrespective of whether the lock implementation is using busy-waiting or suspension by the underlying OS.

Given this fundamental limitation of mutual exclusion locks, it is desirable not to use code-blocks as critical sections, and instead to confine synchronization to hardware instruction-granularity. Using atomic *read–modify–write* (RMW) operations eliminates the possibility of software failure inside a critical section, because that section is then reduced to a single CPU instruction. These atomic RMW operations cannot fail (crash, in the sense of division-by-zero), but rather return an error value if a synchronization operation must be retried. So it becomes possible to produce progress guarantees, which are unattainable with locks. In particular, a method is *non-blocking*, if a thread's pending invocation is never required to wait for another thread's pending invocation to complete [43].

As an example of a non-blocking increment of a shared counter, consider Figure 1(b), adapted from [70] and [71]. The shared variable `Ctr` is read (line 2), incremented (line 3), and written back (line 4), if the value of `Ctr` has not changed since it was read in line 2. Otherwise, the increment operation is

```
1  lock(L)
2  Ctr := Ctr + 1
3  unlock(L)
```

(a)

```
1  loop
2    R1 := Ctr;
3    R2 := INC R1
4    R2 := CAS(Ctr, R1, R2)
5    exit when R2;
6  end loop;
```

(b)

Figure 1: Blocking (a) and non-blocking (b) update of shared counter `Ctr` in pseudocode.

retried. The `CAS` instruction is an atomic RMW operation that conditionally updates a memory location (`Ctr`) with a new value (`R2`) if the memory location contains the expected old value (`R1`).

Note that the `lock()` and `unlock()` operations in Figure 1(a) will be implemented in a way that avoids the reordering of memory operations from inside the critical section with preceding and subsequent memory operations. Such transformations, attempted by a compiler or the CPU itself, may otherwise break the semantics of a multi-threaded program. A simple example would be a compiler that caches variable `Ctr` in a register (where other threads cannot observe it) rather than writing it back to memory after the increment. In contrast, the assembly-like pseudocode in Figure 1(b) contains no such precautions—depending on the semantics of the underlying CPU's `CAS` operation, additional ordering constraints expressed by memory fences may be required (see Section 2 for details).

The atomic RMW operations of a CPU and their semantics with respect to the reordering of memory operations are hardware-specific and subtle. They are referred to as the CPU's *memory consistency model*. To abstract from the diversity of hardware memory consistency models such as x86, ARM, MIPS, and RISC-V available in the market today, a programming language must provide a language-level memory model. In conjunction with non-blocking synchronization constructs, such a memory model allows programmers to use non-blocking synchronization in a hardware-agnostic way. The C++11 standard ([50, 84]) has already specified a strict memory model that was later adopted for C11, OpenCL 2.0, and X10 [86].

However, Ada differs from C++ in important aspects—particularly in the use of Hoare-style monitors (called "protected objects") for blocking synchronization (C++ uses mutexes). The abstract nature of monitors allows design of non-blocking synchronization constructs on a higher abstraction level than the atomic RMW operations provided by the C++11 standard. Because non-blocking synchronization is conceptually difficult when exposed to the programmer, raising the abstraction level can be expected to improve programmability and reduce programming errors.

The only other language-level memory consistency model available today is for the Java programming language [58]. For safety and security of its sandboxed execution environment, Java attempts to bound the semantics of programs with synchronization errors. In the remainder of the paper we do not consider this approach, because it does not align with Ada's approach to regard the semantics of programs with synchronization errors as undefined (as an *erroneous execution*, by [76, 9.10§11]).

The current ISO/IEC standard of the Ada programming language [76] does not support non-blocking synchronization. Support for progress guarantees through non-blocking synchronization will benefit Ada's use with safety-critical embedded applications, aligned with Ada's design goal for program safety. Improved scalability on multicore architectures enhances language efficiency, another major design goal of Ada [76, Introduction].

So the contribution of this paper is the design and evaluation of non-blocking

synchronization techniques for Ada on three abstraction levels:

1. We introduce a *lock elision* mechanism for *monitors* in the context of Ada-protected objects; and we employ Intel's transactional synchronization extensions (TSX) as the underlying hardware transactional memory to resolve inter-thread data conflicts. This is the first approach to employ hardware transactional memory for lock elision with monitors.
2. We propose *concurrent objects* as a high-level programming primitive to encapsulate the complexity of non-blocking synchronization in a language-level construct.
3. We investigate the use of atomic RMW operations in the style of C++11, in conjunction with relaxed memory consistency models; exposing atomic RMW operations at language level will enable programmers to gain fine-grained control over the synchronization problem, including the memory consistency model.
4. We relax several recent state-of-the-art lock and queue synchronization primitives from sequential consistency to acquire-release consistency, to explore the performance advantage of low-level atomic operations and relaxed memory consistency in Ada and C++.
5. We conduct an extensive experimental evaluation on a 28-core x86- and a 16-core ARM v8 multicore platform to explore the trade-offs of the proposed designs with respect to programmability, scalability and performance. We include a comparison with state-of-the-art blocking synchronization constructs.
6. For reproducibility, we have open-sourced all software artifacts in a GitHub repository [8].

The remainder of this paper is organized as follows. Section 2 discusses relevant background-material on memory consistency and non-blocking synchronization. Section 3 introduces lock elision for protected objects. Section 4 presents our concurrent object non-blocking synchronization construct. Section 5 presents our benchmark-implementations of state-of-the-art non-blocking and blocking synchronization constructs. Section 6 contains our experimental results. We discuss the related work in Section 7 and draw conclusions in Section 8.

2 Background

2.1 Memory consistency

To motivate the need for a strict memory model, consider the producer-consumer synchronization example in Figure 2(a), adapted from [74] and [24]. The programmer's intention is to communicate the value of variable `Data` (the payload) from Task 1 to Task 2. It is the programmer's implicit assumption that variable

```
-- Initial values:
Flag : Boolean := False;
Data : Integer := 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;
```

(a)

```
1  Data : Integer with Volatile;
2  Flag : Boolean with Atomic;
```

(b)

Figure 2: (a) Producer-consumer synchronization in pseudocode: Task 1 writes the `Data` variable and then signals Task 2 by setting the `Flag` variable. Task 2 is spinning on the `Flag` variable (lines 2 to 5) and then reads the `Data` variable. Dotted lines represent memory fences. (b) Labeling to enforce sequential consistency in Ada 2012.

`Data` will be written *before* variable `Flag` in Task 1, and that Task 2 will observe `Flag = True` and will then read the payload (`Data`).

The memory semantics that programmers imply for multi-threaded programs alludes to sequential consistency (SC) as defined by L. Lamport in [56]: "A multiprocessor system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."

However, both out-of-order processors and optimizing compilers relax SC for performance reasons (they only maintain data-dependencies within a thread). So without precautions that enforce a sequentially consistent execution, a compiler or CPU may break the intended synchronization in Figure 2 by reordering memory operations that will result in reading `R2 = 0` in line 6 of Task 2; e.g., a store–store reordering of the assignments in lines 2 and 3 of Task 1 will have this effect. Alternatively, a load–load reordering of the read operations in lines 3 and 6 of Task 2 will have the same effect.

CPU ISAs provide *synchronization instructions*, which prohibit the CPU (and the compiler) from reordering memory operations. The most common synchronization instruction is a *memory fence* ("fence" for short). A fence forces all *preceding* memory operations that precede it in program order to be committed to the memory hierarchy before any *subsequent* memory operation in program order can start. It is common for further CPU instructions to have memory-ordering semantics; e.g., on the x86, the CAS instruction (used in Figure 1) constitutes a fence. Besides memory order operations, CPU ISAs provide synchronization instructions to enforce the *atomicity* of memory accesses (CAS is an example of an atomic RMW operation).

The dotted red lines in Figure 2 represent fences, which have been positioned to prohibit those store–store and load–load reorderings. The x86 architecture performs neither store–store nor load–load reordering [46]; so on the x86, the stated fences are unnecessary. Architectures with weaker ordering constraints

may require both or either of those fences to enforce SC. On any architecture that does not guarantee atomicity of load/store operations on type `Boolean`, the compiler will have to resort to atomic synchronization instructions with variable `Flag` instead.

Definition 1. Memory model: *a memory consistency model (or memory model, for short) defines the set of values a read operation is allowed to return [14].*

This terse definition becomes clear when re-considering Figure 2: on x86, the absence of store–store and load–load reordering does not permit to read `R2 = 0` in line 6 of Task 2. Conversely a CPU architecture with a weaker memory consistency model may yield `R2 = 0` in the absence of the stated fences.

2.2 SC-for-DRF

To facilitate programmers' intuition, it would be ideal if all read/write operations of a program's threads were sequentially consistent. But the hardware memory models provided by contemporary CPU architectures relax SC in various ways for the sake of performance [15, 39, 74]. Enforcing SC on any such architecture may incur a noticeable performance penalty (see our experimental evaluation in Section 6).

The workable middle ground between programmer intuition (SC) and performance (relaxed hardware memory models) has been established with SC for data race-free programs (SC-for-DRF) [16, 70]. SC-for-DRF is a language-level memory model that serves as a contract between programmer and language implementation. SC-for-DRF distinguishes between ordinary and synchronized access. To enforce synchronized access, a variable must be labeled by the programmer, e.g., by keyword `atomic` in C++11.

Informally, a program contains a data race if two tasks access the same memory location, at least one of them is a write, at least one of them is an ordinary access, and they are not *ordered* with respect to each other according to the ordering relations stated in Section 2.2.1 below.

SC-for-DRF requires a programmer to ensure a program is free of data races. In turn, the language implementation will guarantee SC for all executions of this program. To do so, a compiler will use a CPU's synchronization instructions (see Section 2.1) to enforce SC on that CPU's hardware memory model. Compiler optimizations must ensure that reordering of operations is performed in such a way that the semantics of the memory model are not violated. Reordering in the CPU is done with respect to the CPU's relaxed hardware memory model, constrained by the ordering semantics of the synchronization instructions inserted by the compiler.

2.2.1 Enforcing order on memory operations

Informally, the acquisition of a lock will be ordered after its most recent release; so access to shared data within a critical section protected by a lock is ordered.

A variable value read by a synchronized read-access will be ordered after the synchronized write-access that wrote that value.

The following ordering relations originated from the DRF [16] and properly labeled [39] hardware memory models. For space considerations we give only an overview of these relations. They were formalized for the memory model of C++ in [28], to which we refer for details.

Ordering of operations across threads is possible only via synchronized access. The *synchronizes-with* relation states that a variable value read by a synchronized read-access B will be ordered after the synchronized write-access A that wrote this value: write operation A synchronizes-with read operation B.

The *happens-before* relation states that if operation A occurs before operation B in program order, then operation A is ordered before operation B. The happens-before relation is transitive and combines with the synchronizes-with relation: if operation A happens-before operation B, and operation B synchronizes-with operation C, then operation A happens-before operation C.

Returning to Figure 2(a): in the SC-for-DRF memory model, variable `flag` must be labeled for synchronized access. Then the synchronized write-access in line 3 of Task 1 synchronizes-with the synchronized read-access in line 3 of Task 2. When this is combined with the happens-before relation, we can establish that writing variable `Data` in line 2 of Task 1 happens-before reading it in line 6 of Task 2.

2.2.2 Relaxed memory models

SC enforces a total order on all synchronized accesses, which may be an exceedingly strong constraint in practice. As an example, consider two pairs of threads employing producer-consumer synchronization within each pair. Each producer access must happen-before the corresponding consumer access, but there is no need to order the data exchange between the two producer-consumer pairs wrt. each other (i.e., they are unrelated).

The "acquire-release" memory model and the "relaxed" memory model are relaxations of the "sequentially consistent" memory model. They require concessions from the programmer to weaken SC, which increases opportunities for the CPU to reorder memory operations. Altogether, the following three memory models can then be used to label a synchronized access.

Sequentially consistent ordering: all threads see the same, total order of synchronized accesses labeled "sequentially consistent". A sequentially consistent write of a value to a variable synchronizes-with a sequentially consistent read of this value from the same variable.

A sequentially consistent memory operation needs a fence before and after it to prohibit reordering of adjacent memory operations across it. This may incur considerable performance overhead, even on platforms like the x86, which is close-to-SC (see our experimental evaluation in Section 6).

Relaxed ordering: accesses labeled as "relaxed" do not engage in synchronizes-with relationships. So a thread X reading a value written by a "re-

laxed" write operation in thread Y cannot infer any ordering with respect to other operations in thread Y.

A relaxed memory operation places no ordering constraint wrt. its surrounding memory operations; so no fence is required either before or after the relaxed operation. On a given core, a relaxed read operation of variable V must return the most recent value written to V, but there is no global order on the accesses to V that cores would agree on. So a core may serve a read operation on variable V from its on-core write buffer, while other cores still read an outdated value of V from the memory hierarchy.

Relaxed memory operations are useful for polling a flag-variable at no ordering cost, and perform a sequentially consistent memory operation once the flag contains the desired value (e.g., the TATAS lock in Section 5.2). Relaxed memory operations can be used in conjunction with explicit (programmer-supplied) memory fences (e.g., the Taubenfeld lock in Section 5.2).

Acquire-release ordering: the ordering guarantees are stronger than with relaxed ordering. A read operation (called read-acquire) can be labeled `acquire`—a write operation (called write-release) by label `release`. Synchronization between write-release and read-acquire is pairwise between the thread that issues the write-release and the thread(s) that issue the read-acquire operation (a write-release synchronizes-with a read-acquire). The principal use case for this type of synchronization is the communication of payload-data between threads, with the help of a flag variable (e.g., Figure 2(a)).

A write-release is a write operation with a fence before it (in program order), to prohibit reordering of the write-release with prior (in program order) read and write operations. A write-release requires no fence after it and so is cheaper than a sequentially consistent operation. A read-acquire is a read operation with a fence after it (in program order) to prohibit reordering of the read-acquire with subsequent memory operations. A read-acquire needs no preceding fence and so is cheaper than a sequentially consistent operation.

Table 1 summarizes the ordering guarantees of these three memory models.

2.3 Memory ordering in Ada 2012

Ada 2012 allows labeling of variables by aspect `volatile`, as depicted in Figure 2(b). The intention for volatile variables in Ada 2012 was to guarantee that all tasks agree on the same order of updates [76, C.6§16/3]. (All memory accesses to `volatile` variables are sequentially consistent.) But as pointed out in [25], this approach is insufficient as a memory model, because (1) no guarantees are provided for un-labeled variables, and (2) it is impractical to label all shared data as `volatile`.

Table 1: Memory order and ordering constraints for compiler and CPU

Memory order	**Involved threads**	**Ordering constraints (for compiler and CPU)**
relaxed	1	No inter-thread ordering constraints
acquire-release	write-release: 1 read-acquire: $1 \leq x \leq N$	(1) Read and write operations before the `write-release` (in program order) happen-before it. (2) The `write-release` that writes a particular value to variable V synchronizes-with the `read-acquire` that reads this particular value from variable V. (3) The `read-acquire` happens-before subsequent (in program order) read and write operations.
sequentially consistent	all	(1) All ordering constraints of the acquire-release memory order apply. (2) A total order on all sequentially consistent memory operations exists.

2.4 Blocking and non-blocking synchronization

We recapitulate definitions of important terms from [43]:

Blocking: A method implementation is blocking if the delay of any one thread can delay others.

Non-blocking: A method implementation is non-blocking if the delay of a thread cannot delay others.

Obstruction-free: A method is obstruction-free if, from any point after which it executes in isolation, it finishes in a finite number of steps.

Lock-free: A method is lock-free if it guarantees that infinitely often *some* method call finishes in a finite number of steps.

Wait-free: A method is wait-free if it guarantees that *every* call finishes its execution in a finite number of steps.

The definitions are ordered in the sense that a wait-free method is lock-free, a lock-free method is obstruction-free, and an obstruction-free method is non-blocking; but not vice versa. Note that the term "blocking" in the definition is different from the OS literature: a spin-loop may block a thread from making progress according to the definition. (The literature sometimes uses the term "lock-free" to denote absence of locks.)

3 Lock elision for protected objects

We use lock elision [65] to introduce non-blocking synchronization with protected objects (POs) in Ada. Lock elision is a technique to replace coarse-grained locks by optimistic concurrent execution. This transformation is transparent to programmers who use the blocking synchronization mechanism of Ada POs, because the language runtime system attempts parallel execution of critical sections (which are then called transactional regions).

The key insight with lock elision is that many dynamic data sharing patterns among threads do not conflict and so do not require the acquisition of a lock; e.g., a concurrent hash-map [12] contains multiple key-value pairs. Two threads updating different keys will not conflict and hence do not require serialization (locking). Serialization is needed only among threads updating the same key's value.

Lock elision requires hardware support to be efficient. Recent CPU architectures from Intel, IBM, and Sun/Oracle provide hardware transactional memory (TM) extensions [47, 81, 35], which allow a processor to dynamically detect inter-thread data conflicts. (Transactional memory was originally proposed in 1993 by Herlihy and Moss [42]. A categorization of hardware TM systems is given in [37].)

We use Intel TSX [47] as the underlying hardware mechanism. Lock elision with Intel TSX until now has been attempted only with mutual exclusion locks in C and C++ [85, 54]. In contrast, Ada's POs [76] implement the monitor concept [44]. The PO synchronization mechanism goes beyond "plain" mutual exclusion, because POs provide protected functions, procedures, and entries. Protected functions do not update shared data, and hence multiple protected functions of a PO may execute in parallel. Protected procedures and entries require mutual exclusion. Protected entries provide programmed guards for conditional synchronization.

With lock elision, a thread will *speculatively* execute a critical section (a transactional region) without acquiring the associated lock (the lock is said to be elided). In the absence of inter-thread data conflicts, the memory updates (write operations) of the thread are committed to memory. If a data conflict with another thread is detected, speculative execution of the transactional region is aborted and the thread's write operations are *not* committed to memory. The failed thread must then re-execute the transactional region. With lock elision, programmers are granted the convenience of using coarse-grained locks, which will exhibit the scalability of fine-grained locking in the absence of inter-thread data conflicts.

To detect data conflicts and ensure an atomic commit of a thread's memory updates, a read-set and a write-set are maintained for a transactional region. The read-set consists of addresses the thread reads from within the transactional region; the write-set consists of addresses written to within the transactional region. The updates to the write-set will be committed atomically to memory in the absence of data conflicts (see Def. 2), and otherwise discarded.

```
1 function xbegin return uint32 is
2   ret : uint32 := XBEGIN_STARTED;
3 begin
4   Asm(".byte 0xc7,0xf8 ; .long 0",
5   Outputs => uint32'Asm_Output ("+a", ret),
6   Clobber => "memory", Volatile => True);
7   return ret;
8 end xbegin;
```

Listing 1: Ada implementation for TSX instruction `xbegin`.

Definition 2. Data Conflict. *Assume a thread executing a transactional region. A data conflict occurs if another thread either reads a location that is part of the transactional region's write-set, or writes a location that is a part of either the read- or write-set of the transactional region (adapted from [47]).*

To elide locks transparently from protected functions and procedures, we adapt the GNU Ada runtime library (see Section 3.1). We investigate two lock elision schemes for protected entries in Section 3.2 and discuss the general applicability of lock elision with POs in Section 3.4. The experimental evaluation is presented in Section 6.1.

3.1 Lock elision with the GNU Ada runtime library (GNARL)

To access the Intel TSX instruction set extensions from Ada code, we created an Ada package with a procedural interface to each TSX instruction. The specification of this package has been described in [51] and later open-sourced [8].

We employ inline assembly to emit bytes corresponding to Intel TSX instructions. Listing 1 depicts our implementation for the `xbegin` instruction. The remaining TSX instructions are implemented in a similar manner.

The Ada 2012 RM [76, Chapter 9.5.1(5)] states that execution of a protected procedure needs exclusive read-write access to the PO. Speculative execution of critical sections with elided locks does not meet this requirement. Rather, serialization is achieved by re-executing a critical section in case of a transactional abort [54, 70].

GNARL employs one lock per PO for synchronization. We perform lock elision of such locks as follows.

1. Check if the lock is free. If not, wait until the lock has been released by the task that is holding it.

2. Once the lock is free, the task starts its transaction without actual lock acquisition, and executes the critical section (the body of a protected function or procedure).

3. If the task proceeds through the critical section without encountering a data conflict, it commits the updates in its write-set to memory (to become visible to other tasks).

Note that during this entire process, the lock appears to be free to all tasks.

Whenever a transaction aborts, control is transferred back to the `xbegin` instruction. In case of an abort, the x86 processor sets a specific bit in the EAX register to signal the type of conflict that caused the abort. This value is used to decide whether to retry the transaction or fall back to the locking code.

There is no guarantee from the processor that a transaction will eventually succeed. So the fall-back path with the conventional lock-based code is required to prevent infinite aborts. If any single task proceeds with the lock-based code, all other tasks competing for access to the PO must wait (serialize) until the lock has been released (alike the conventional GNARL implementation).

Listing 2 depicts the PO lock elision scheme we added to GNARL. Procedure `Write_Lock` is called from inside GNARL before entering a PO's critical section. In our implementation, `Write_Lock` calls procedure `Try_Elision` to attempt lock elision. The transaction retry count is initialized in line 14, before the start of the transaction. Line 16 (`xbegin`) marks the start of the transaction. If the PO's lock is found open, `Try_Elision` returns in transactional mode (line 20); otherwise a task will abort the transaction (in line 22). The abort will transfer control to line 16 and from there to line 31, where the task will wait until the lock becomes available. If `retry` exceeds `MAX_RETRY`, procedure `Try_Elision` terminates the retry-loop and returns "fail" (line 35). As a result, procedure `Write_Lock` will acquire the PO's lock (line 4) and return in non-transactional mode. Note that line 4 can only be reached in non-transactional mode.

The GNARL function for exiting a critical section is conceptually much simpler and has been omitted due to space constraints.

We need read-access to the PO's lock to detect data conflicts. If task A is inside a transaction and task B acquires the lock, task A must abort its transaction. This can be achieved by keeping the PO's lock in the read-set of task A's transaction. But with the lock library underneath GNARL, it is not possible to read a lock without modifying it. As a work-around we introduced a shadow-lock variable per PO lock (as outlined in [6]). The shadow-lock is of type integer, and we added it to the protected object package inside GNARL. The original PO lock is used with the lock-based code, but not with transactions. The shadow-lock is the one kept in the read-set of a transaction. We set the shadow-lock if a task acquires the PO's lock. To ensure atomicity, we use GCC's atomic built-in functions to access the shadow-lock.

To improve performance we apply the following three adjustments. First, we employ the x86 `pause` instruction inside the busy-waiting loop used by a task to wait for a PO lock to be released. The `pause` instruction is a hint to the CPU to limit speculative execution, which increases the speed at which the release of the lock is detected [5]. This optimization yielded a considerable performance improvement. Second, a task attempts transactional execution several times before falling back to the lock. As shown in Listing 2 (lines 10 and 15), a task

```
1  procedure Write_Lock          -- GNARL lock acquisition procedure
2    result := Try_Elision       -- Added: attempt to elide lock
3    if result = fail then
4      acquire PO.lock           -- Lock elision failed -> acquire lock
5    end if
6    return                      -- Proceed to critical section, lock is
7  end Write_Lock                -- either elided or acquired.
8
9  -- GNARL extension for lock elision:
10 MAX_RETRY : constant Natural := ...  -- Tuning-knob 1
11 BACKOFF   : constant Natural := ...  -- Tuning-knob 2
12
13 function Try_Elision return Result_Type
14   retry = 0
15   while retry < MAX_RETRY loop          -- Attempt elision multiple times
16     state := xbegin     -- Start transaction; resume at abort or conflict
17     if state = XBEGIN_STARTED then
18       -- From here we execute in transactional mode
19       if PO.lock = open then
20         return success  -- Report that elision succeeded
21       else              -- Another task is holding lock:
22         xabort          --   Abort transaction (-> resume at line 16)
23       end if
24     else if state = XABORT_CAPACITY or state = XABORT_RETRY then
25       return fail       -- Report that elision failed
26     else                -- Transaction failed, but might succeed next time
27        if state = XABORT_CONFLICT then
28          -- Data conflict: defer to competing tasks:
29          Exponential_Backoff ((2**retry)*BACKOFF)
30        end if
31        wait until PO.lock = open -- Data conflict or xabort
32        retry := retry + 1
33     end if
34   end loop
35   return fail           -- Report that elision failed
36 end Try_Elision
```

Listing 2: Elision of a PO lock in GNARL (in pseudocode). `Try_Elision` is called from procedure `Write_Lock` before entering a critical section. The call returns either inside a transaction (line 20) or to acquire the PO's lock (lines 25 and 35). Only in the first case will the lock be elided. The GNARL function for exiting a critical section is conceptually much simpler and has been omitted due to space constraints.

tries up to `MAX_RETRY` times to execute a critical section as a transaction. Last, depending on the conflict type, a task may already fall back to the PO lock before reaching the `MAX_RETRY` limit. On a transaction abort the CPU provides a status code. If the `XABORT_RETRY`-flag is not set in the status code, the transaction will not succeed in a retry either (e.g., if the task attempts a system-call inside the critical section). If the `XABORT_CAPACITY`-flag is set, the transactional memory capacity reached its limit. When we detect one of these flags, we fall back to the conventional lock without further retries. This is a heuristic: depending on control flow, a task might refrain from executing a system call during the retry, or the transactional memory capacity limit was exceeded because of another task inside a transaction on the same processor core. (This may occur with processors that support hyperthreads.) In line 29 of Listing 2, a task that

encountered a data conflict will wait to allow competing tasks to finish their transactions before retrying. Procedure `Exponential_Backoff` contains a busy-waiting loop, with iteration count specified by the procedure's argument. The waiting time increases with every unsuccessful retry ("exponential backoff"). This measure avoids situations where tasks keep mutually aborting each other's transactions without overall progress.

Lock elision may not always improve performance. For example, critical sections that routinely lead to a capacity overflow will always fall back to the lock. Concrete values for the maximum number of attempted lock elisions and calibration of the busy-waiting loop (lines 10 and 11 in Listing 2) differ across benchmarks and HW platforms. Thus we put under programmer control these tuning parameters and the decision whether to elide a PO lock. Alternatively, GNARL itself may be extended with dynamic profiling capabilities to take decisions at runtime.

3.2 Lock elision for protected entries

The Ada 2012 RM [76, Chapter 9.5.3(16)] states that queued entry calls with an open barrier take precedence over all other protected operations of the PO (known as the "eggshell model" [66]). This requirement avoids starvation of queued tasks. Nevertheless, RM Clause 9.5.3(16) restricts the degree of parallelism obtainable with lock elision, because it requires task-serialization irrespective of inter-task data conflicts. For many parallel workloads, freedom from starvation is not a concern (latency or throughput are). For such workloads (e.g., the "stencil" and "map" programming patterns from [60]), the amount of work is constant and known *a priori*. The order in which tasks enter a critical region is immaterial, and it is impossible to starve a task because the amount of work is bounded.

Permissive lock elision. One possible elision-scheme for protected entries is to waive Clause (16), at least in response to a programmer-supplied PO type annotation such as a pragma. Protected functions, procedures, and entries will then execute in parallel in any order, subject only to serialization due to inter-task data conflicts.

Restrictive lock elision. A more restrictive scheme will provide a mode-switch from elided to non-elided serialized execution as soon as an entry call enqueues at a closed barrier. Such semantics can be achieved by an `is_queued`-flag, added to the read-set of every PO transaction. The task that is about to enqueue will write to the `is_queued`-flag, aborting all ongoing transactions. Procedure `Try_Elision` from Listing 2 must be adapted such that no transactional execution is attempted if the `is_queued`-flag is set. The flag will be cleared and the PO switched back to elided mode once all entry queues are empty.

The core part of the Ada language does not specify the order in which entry queues will be served. (The real-time systems annex of the Ada RM addresses (implementation-dependent) queuing policies, which we did not consider for this paper.) If a first-come-first-served fairness property is not needed, parallelism

can be leveraged with the restrictive scheme by allowing the tasks in the front position of each queue to proceed to the critical section in parallel.

3.3 Programmer control of PO lock elision

To give the programmer control for lock elision, we propose for future Ada versions an aspect `Lock_Elision` that may be applied to protected objects and protected types. The aspect has three parameters:

`Mode:` which has two possible values, `Permissive` and `Restricted`, according to the schemes described in Section 3.2.

`Max_Retry:` a positive number to define the number of maximum retries before lock elision falls back to the standard lock-based synchronization (cf. Section 3.1). This variable is a tuning knob; from our experiments, the value of this variable must be larger than the number of participating tasks to ensure competitive performance. On our 44-core test platform, we set this value to 200 (so a task would try 200× to perform a protected function or procedure in transactional mode before falling back to the lock).

`Backoff:` a positive number to affect procedure `Exponential_Backoff` (cf. Section 3.1). In our lock-elided GNARL implementation, `Backoff` was set to 10. This setup was used with all benchmarks in Section 6.1.

If the aspect `Lock_Elision` is not applied to a protected object, standard lock-based synchronization is performed for this PO.

3.4 General applicability of lock elision with POs

Our lock elision scheme for Ada POs allows the programmer to use the well-understood semantics of monitors and rely on the language runtime system to attempt parallel execution of transactional regions. Clearly this scheme offers high programmability, because it shields the programmer from the entire non-blocking synchronization problem. As shown in Section 6.1, we found that lock elision of POs achieves high scalability and performance. But several restrictions prevent this scheme from being generally applicable:

1. Intel TSX needs a fall-back path, which by necessity will use the underlying lock of the PO. So for the fall-back path, progress cannot be guaranteed. To maintain progress guarantees, an implementation may fall back to software-transactional memory, but this yields lower performance (we did not consider software-transactional memory for our implementation).

2. Capacity constraints of a CPU's read- and write-set restrict its ability to elide a coarse-grained lock. For example, in our experiments we found that our linked-list implementation inevitably caused TSX capacity aborts if more than 100 list-nodes had to be traversed within a single PO call (e.g., in the `lookup()` procedure). Without re-factoring the linked-list

implementation to use more fine-grained locking, elision with hardware TM cannot be applied.

3. As outlined in the introduction of this section, the optimistic parallelization applied with lock elision is only effective with inter-task data sharing patterns that have a low conflict rate. Concurrent hash-maps show this property, but we found elision to perform well with less-obvious cases, such as k-means clustering and even the Dining Philosopher's problem (see our experimental evaluation in Section 6.1). The adversary case constitutes a small, highly contended amount of shared data in the PO; examples include shared counters, and read- and write-pointers of bounded queues.

Because of these restrictions, lock elision of POs cannot be applied in the general case. Rather, a critical assessment is necessary to decide whether lock elision will be effective for a given use case. Such an assessment can be made by the programmer if the programming language gives the means to explicitly enable/disable elision. Static program analysis may assist the programmer with this decision. Alternatively, the decision can be taken by the runtime system through dynamic program analysis.

We propose concurrent objects (Section 4) as an alternative synchronization construct for those cases where lock elision of POs is not possible. Concurrent objects are on the same abstraction level as POs, but they nevertheless require involvement of the programmer to solve the non-blocking synchronization problem.

4 Concurrent objects and low-level atomics

To mitigate the complexity associated with non-blocking synchronization, we propose concurrent objects (COs) as a novel high-level language construct. COs resemble Ada's POs, which are Hoare-style monitors [44] used for implementing blocking multi-threaded shared memory systems in Ada. Unlike mutually exclusive POs, the entities of a CO execute in parallel, due to a fine-grained, optimistic synchronization mechanism framed by the semantics of concurrent entry execution. The programmer is only required to label shared data accesses in the code of concurrent entries. For this purpose we extend Ada's memory model with synchronized types to support the expression of memory ordering operations at a sufficient level of detail.

Ada is already well-positioned for the extension, by a strict memory model in conjunction with support for non-blocking synchronization, because it provides tasks as part of the language specification. This rules out inconsistencies that may result from threading functionality available through add-on libraries [27].

Ada 83 was the first widely used high-level programming language to provide first-class support for shared-memory programming [14]. The approach taken with Ada 83 and later language revisions was to require legal programs to be free from synchronization errors, as with SC-for-DRF (see Section 2). In the

same manner, our approach with COs uses SC-for-DRF semantics. A program with a data-race constitutes an *erroneous execution*, by [76, 9.10§11].

Our approach to introducing COs as a safe and efficient non-blocking synchronization mechanism for Ada consists of the following parts.

Synchronized types: We extend Ada's memory model by introducing synchronized types, which allow the expression of memory-ordering operations consistently and at a sufficient level of detail. Memory-ordering operations are expressed through aspects and attributes. Language support for spin loop synchronization via synchronized variables is proposed (Section 4.1).

Synchronization semantics: We introduce the fine-grained, optimistic synchronization mechanism of COs (Section 4.2).

Concurrent data structure example: An implementation of hazard pointers is provided together with an application of them—a lock-free stack (Section 4.3).

Low-level atomics: We provide an alternative, low-level API on synchronized types, giving programmers full control over the implementation of non-blocking synchronization semantics (Section 4.4).

We illustrate the use of concurrent objects and the alternative, low-level API by several examples. To the best of our knowledge, this is the first approach to provide a non-blocking synchronization construct as a first-class citizen of a high-level programming language.

4.1 Synchronized types

On programming-language level, to express a program free of data-races requires a means to enforce an order on memory operations. Ada's POs serve this purpose by blocking synchronization. For non-blocking synchronization we propose *synchronized variables*, which allow the programmer to label variables to mandate synchronized access (see Section 2.2). Concurrent objects employ synchronized types to synchronize access to shared data.

Synchronized variables are similar to C++'s atomic variables. But the use of synchronized variables is restricted to the lexical scope (data part) of a CO.

A read-access to a synchronized variable may be labeled with attribute `Concurrent_Read`, write accesses with attribute `Concurrent_Write`. Both attributes have a parameter `Memory_Order` to specify the memory order of access. (If the operations are not labeled, the default values given below apply.) In case of read accesses, `Memory_Order` can be either `Sequentially_Consistent`, `Acquire`, or `Relaxed`. The default value is `Sequentially_Consistent`. For write accesses the allowed values are `Sequentially_Consistent`, `Release`, and `Relaxed`. The default value is again `Sequentially_Consistent`.

For example, assigning the value of synchronized variable `Y` to synchronized variable `X` is given like

```
X'Concurrent_Write(Memory_Order => Release) :=
  Y'Concurrent_Read(Memory_Order => Acquire);
```

We propose aspects for specifying variable-specific default-values for the attributes described above. So when declaring synchronized variables, the default values for read and write accesses can be given by aspects `Memory_Order_Read` and `Memory_Order_Write`. The values allowed are the same as those given above for read and write accesses. If these memory model aspects are specified with the declaration of a synchronized variable, the attributes `Concurrent_Read` and `Concurrent_Write` may be omitted for actual read and write accesses to this variable. The previous example can be rewritten as follows.

```
X: integer with Synchronized, Memory_Order_Write => Release;
Y: integer with Synchronized, Memory_Order_Read => Acquire;
...
X := Y;
```

However, these attributes can still be used to override the default values for particular accesses.

4.2 CO semantic definition

Concurrent objects have a specification part and an implementation part. A specification part looks like

```
concurrent <Name> is
   -- Entries, procedures, and functions declared here
private
   -- data part: local variables (synchronized, read_modify_write, ...)
   -- private entries, procedures, and functions
end <Name>;
```

The implementation part (body) of a concurrent object contains the implementation of public and private entries, procedures, and functions.

All operations of concurrent objects can be executed in parallel. Synchronized variables must be used for synchronizing the executing operations. Entries have Boolean-valued guards. The Boolean expressions for such guards may contain only synchronized variables declared in the data part of the concurrent object, and constants. Calling an entry results in either immediate execution of the entry's body, if the guard evaluates to `true`, or spinning until the guard eventually evaluates to `true`. We call such a spin loop *sync loop*. An example for an entry specification of a concurrent object is

```
entry E(
   -- parameters
   );
```

A corresponding entry implementation looks like

```
entry E(
   -- parameters
   ) until <Cond> is
begin
```

```
   -- code
end E;
```

Keyword `until` is used for guards of CO's entries, whereas `when` is used for the guards of POs; this highlights the semantic differences between POs and COs.

Aspect `Synchronized_Components` relates to aspect `Synchronized` in the same way as `Atomic_Components` relates to `Atomic` in Ada 2012.

If a variable inside the data part of a CO has the aspect `Read_Modify_Write`, this implies that the variable is synchronized. Write access to an RMW variable is an RMW access. RMW access is done via the attribute `Concurrent_Exchange`. This attribute has two parameters. The first, `Memory_Order_Success`, specifies the memory order for a successful write. The second, `Memory_Order_Failure`, specifies the memory order if the write access fails. Variable-specific default-values can be specified for RMW variables in a similar fashion as for synchronized variables. As an example, we consider the following specification of a lock-free stack.

```
1  concurrent Lock_Free_Stack
2  is
3    entry Push(D: Data);
4    entry Pop(D: out Data);
5  private
6    Head: List_P with Read_Modify_Write,
7      Memory_Order_Read => Relaxed,
8      Memory_Order_Write_Success => Release,
9      Memory_Order_Write_Failure => Relaxed;
10 end Lock_Free_Stack;
```

Attribute `Memory_Order_Success` is either `Sequentially_Consistent`, `Acquire`, `Release`, or `Relaxed`. The allowed values for `Memory_Order_Failure` are `Sequentially_Consistent`, `Acquire`, and `Relaxed`. The default value for both is `Sequentially_Consistent`.

For read access to an RMW variable, attribute `Concurrent_Read` has to be used. The parameter `Memory_Order` must be provided by the programmer. Its value is one of `Sequentially_Consistent`, `Acquire`, `Relaxed`. The default value is `Sequentially_Consistent`.

Again, aspects for variable-specific default-values of the attributes described above may be provided with the declaration of an RMW variable. The aspects are `Memory_Order_Read` for read-access, and `Memory_Order_Write_Success` and `Memory_Order_Write_Failure` for write-access. The allowed values are as stated above.

The guards of entries may have the form `X = X'Old` when `X` denotes an RMW variable of the concurrent object.

If during the execution of an entry's body an RMW operation is encountered, that operation might succeed immediately, in which case execution proceeds after the operation in the normal way. If the operation fails, the whole execution of the entry is restarted (*implicit sync loop*). In particular, only those statements being data-dependent on the RMW variable are re-executed. Statements not being data-dependent on the RMW variables are executed only on the first try.

We propose a new state for Ada tasks to ease correct scheduling and dispatching for threads synchronizing via synchronized or RMW variables. If a thread is in a sync loop, the thread state changes to "in_sync_loop". Sync loops can only appear in the lexical scope of COs; hence, they can be detected easily

by the compiler and cannot be mistaken with regular loops. In this fashion the runtime system can guarantee that not all available CPUs (cores) are busy executing threads in state "in_sync_loop". So at least one thread is certain to make progress, and finally all synchronized or RMW variables are released.

After leaving a sync loop, the thread state changes back to "runnable". For more examples of such spin loops see [25, 26].

Tools performing (static) analysis of multi-threaded shared memory systems can also leverage this knowledge about sync loops.[1]

4.3 Example: lock-free stack using hazard pointers

Hazard pointers have been introduced in [61]. We present an implementation of hazard pointers, based on our proposed language extension to Ada. In Listing 3 we show only the major parts of the implementation; the remaining parts are similar to the code given in [61].

```
1  generic
2    type Thread_ID is (<>);
3    No_Of_HPs_per_Thread: positive;
4    type Item is limited private;
5    type Pointer is access item;
6    with procedure Free(P: Pointer);
7    Threshold: positive := 2;
8
9  package Hazard_Pointers is
10
11   subtype Pointer_ID is positive range 1..No_Of_HPs_per_Thread;
12   type HP_Array is array(Thread_ID, Pointer_ID) of Pointer with
         Read_Modify_Write_Components;
13
14   concurrent HP is
15     entry Assign_And_Register_HP(
16       Th_ID: Thread_ID;
17       P_ID: Pointer_ID;
18       Source_Concurrent_Function: access concurrent function return
             Pointer;
19       Target_Pointer: out Pointer);
20         -- Safely assigns result of concurrent function
21         -- Source_Concurrent_Function to Target_Pointer and registers
22         -- Target_Pointer as being a hazard pointer
23     entry Assign_And_Register_HP(
24       Th_ID: Thread_ID;
25       P_ID: Pointer_ID;
26       Source_Function: access function return Pointer;
27       Target_Pointer: out Pointer);
28         -- Safely assigns result of function
29         -- Source_Function to Target_Pointer and registers
30         -- Target_Pointer as being a hazard pointer
31     procedure Retire(
32       Th_ID: Thread_ID;
33       P_ID: Pointer_ID;
34       P: Pointer);
35         -- Free hazard pointer P unless referred by other threads for
36         -- arbitrary reuse. Otherwise retain until next retire.
37   private
38     HPs: HP_Array;
39   end HP;
40
```

[1] Such a tool could for example try to prove that the weak memory model is used correctly or that the implemented algorithm is not only lock-free, but even wait-free.

```
41 end Hazard_Pointers;
```

```
1  package body Hazard_Pointers is
2
3    ...
4
5    concurrent body HP
6    is
7      entry Assign_And_Register_HP(
8        Th_ID: Thread_ID;
9        P_ID: Pointer_ID;
10       Source_Concurrent_Function: access concurrent function return
             Pointer;
11       Target_Pointer: out Pointer)
12           until Source_Pointer = Target_Pointer is
13     begin
14       Target_Pointer := Source_Concurrent_Function.all;
15       HPs(Th_ID, P_ID) := Target_Pointer; -- RMW
16     end Assign_And_Register_HP;
17
18     -- implementation of entry Assign_And_Register_HP similar to above
19     ...
20
21   end HP;
22
23 end Hazard_Pointers;
```

Listing 3: Hazard pointers.

In the hazard pointer example (body of Listing 3) there are two RMW operations, one of which is shown in line 15. Our proposed semantics for RMW operations guarantees that if the RMW operation succeeds, the statement located immediately after the RMW operation (in program order) will be executed next. If the RMW operation fails, the enclosing entry will be re-executed from the beginning.

Listing 4 is an instantiation of hazard pointers: a lock-free stack.

```
1  package HP_NB_Stack is
2
3    subtype Data is Integer;
4
5    type List;
6    type List_P is access List;
7    type List is
8      record
9        D: Data;
10       Next: List_P;
11     end record;
12
13   subtype Thread_ID is positive range 1..2;
14
15   Empty: exception;
16
17   concurrent Lock_Free_Stack
18   is
19     entry Push(Th_ID: Thread_ID; D: Data);
20     entry Pop(Th_ID: Thread_ID; D: out Data);
21   private
22     Head: List_P with Read_Modify_Write;
23
24     function Get_Head return List_P;
25
26   end Lock_Free_Stack;
27
28 end HP_NB_Stack;
```

```
1  with Hazard_Pointers;
2
3  package body LFSHP is
4
5    procedure Free_List(P: List_P) is
6    begin
7      ... -- do something meaningful
8    end Free_List;
9
10   package My_HPs is new Hazard_Pointers(
11     Thread_ID => Thread_ID,
12     No_Of_HPs_per_Thread => 1,
13     Item => List,
14     Pointer => List_P,
15     Free => Free_List);
16
17   concurrent body Lock_Free_Stack is
18
19     function Get_Head return List_P is
20     begin
21       return Head;
22     end Get_Head;
23
24     entry Push (Th_ID: Thread_ID; D: Data)
25         until Head = Head'OLD is
26       New_Node: List_P := new List;
27     begin
28       New_Node.all := (D => D, Next => Head);
29       Head := New_Node; --RMW
30     end Push;
31
32     entry Pop(Th_ID: Thread_ID; D: out Data)
33         until Head = Head'OLD is
34       Old_Head: List_P;
35     begin
36       My_HPs.HP.Assign_And_Register_HP_C(
37         Th_ID => Th_ID,
38         P_ID => 1,
39         Source_Concurrent_Function => Get_Head'ACCESS,
40         Target_Pointer => Old_Head);
41       if Old_Head /= null then
42         Head := Old_Head.Next; --RMW
43         D := Old_head.D;
44         My_HPs.HP.Retire(Th_ID => Th_ID, P_ID => 1, P => Old_Head);
45       else
46         raise Empty;
47       end if;
48     end Pop;
49
50   end Lock_Free_Stack;
51
52 end lfshp;
```

Listing 4: Implementation of lock-free stack using hazard pointers.

Entry `Push` inserts a new node at the head of the list by allocating a new node in the declarative part. In line 28 `New_Node` is assigned the data and a pointer to the current head of the list. The RMW operation in line 29 ensures the head has not been changed by another thread. If it has been changed, entry `Push` is re-executed from start. Only line 28 is re-executed—not the allocation statement in the declarative part—because line 29 is not data-dependent on the allocation statement.

Entry `Pop` registers `Old_Head` as a hazard pointer and at the same time

assigns it the value of the head of the list via function `Get_Head` (lines 36 to 40). If the list is not empty (`Old_Head /= null`), the RMW operation at line 42 ensures that the head of the list has not been changed by another thread. If it has been changed, entry `Pop` is re-executed from the beginning; if not, the data from `Old_Head` is assigned to the out parameter of `Pop` and `Old_Head` is retired (lines 43 to 44).

To summarize, by employing implicit spin loops with COs, the algorithmic aspects of the code are much easier to comprehend. In reviewing the code, one may first concentrate on the algorithmic aspects. Later on, the synchronization aspects and the implementation of the weak memory model may be studied.

Further examples of concurrent data structures implemented with COs are provided in [26].

4.4 Low-level atomics API

If the programmer needs synchronization on a lower abstraction-level than concurrent objects provide, we provide a pre-defined API. Listing 5 shows package `Memory_Model`. Generic function `Read_Modify_Write` allows use of the RMW operation of the underlying computer hardware.

But when using this API, the programmer must implement sync loops explicitly. To let the runtime change the state to "in_sync_loop" (cf. Section 4.2), we introduce a new aspect `Sync_Loop`. If this aspect is not used correctly, the information given to the runtime may be incomplete or false, which could give rise to concurrency defects such as deadlocks, livelocks, and other problems.

```
1  package Memory_Model is
2
3    type Memory_Order_Type is (
4      Sequentially_Consistent,
5      Relaxed,
6      Acquire,
7      Release);
8
9    subtype Memory_Order_Write_Success_Type is Memory_Order_Type;
10
11   subtype Memory_Order_Write_Failure_Type is Memory_Order_Type
12     range Sequentially_Consistent .. Acquire;
13
14   generic
15      type Some_Synchronized_Type is private;
16      with function Old_Value return Some_Synchronized_Type;
17      with function Update return Some_Synchronized_Type;
18      Read_Modify_Write_Variable: in out Some_Synchronized_Type
19        with Read_Modify_Write;
20      Memory_Order_Write_Success: Memory_Order_Write_Success_Type :=
21        Sequentially_Consistent;
22      Memory_Order_Write_Failure: Memory_Order_Write_Failure_Type :=
23        Sequentially_Consistent;
24   function Read_Modify_Write return Boolean;
25
26 end Memory_Model;
```

Listing 5: Package Memory_Model.

Listing 6 shows the implementation of procedure `Pop` of a non-blocking stack based on our proposed API. Before the instantiation two functions have to be

declared, one for defining the old value and one for the new value of the RMW variable (lines 5 and 9). Function `RMW_Head_Pop` is instantiated in line 13. The actual RMW operation is in line 23. The implementation of `Push` follows along the same lines and has been omitted for space considerations.

```
1  concurrent body NB_Stack is
2  ...
3    procedure Pop(D: out Data) is
4      Old_Head: List_P;
5      function Old_Head_Pop return List_P is
6      begin
7        return Old_Head;
8      end Old_Head_Pop;
9      function Update_Head_Pop return List_P is
10     begin
11       return Old_Head.Next;
12     end Update_Head_Pop;
13     function RMW_Head_Pop return Boolean is
14       new Memory_Model.Read_Modify_Write(
15         Some_Atomic_Type => List_P,
16         Old_Value => Old_Head_Pop,
17         Update => Update_Head_Pop,
18         Read_Modify_Write_Variable => Head);
19   begin
20     loop with Sync_Loop
21       Old_Head := Head;
22       if Old_Head /= null then
23         if RMW_Head_Pop then
24           D := Old_Head.D;
25           exit;
26         end if;
27       else
28         raise Empty;
29       end if;
30     end loop;
31   end Pop;
32 ...
33 end NB_Stack;
```

Listing 6: Lock-free stack implementation (snippet) based on the proposed low-level atomics API.

5 Lock and queue synchronization benchmarks

To determine the scalability and performance of blocking versus non-blocking synchronization, we implemented state-of-the-art locks and concurrent queue data structures to augment our experimental evaluation in Section 6. We gained five insights from this benchmark suite:

1. It provides quantitative evidence that scalability and performance of the surveyed non-blocking synchronization constructs surpass blocking synchronization constructs. (This is on top of the progress guarantees that non-blocking synchronization constructs provide.)

2. To quantify the performance improvement obtainable by using a relaxed memory consistency model, we provide two versions for several of our

locks: one employing SC, and one using acquire-release memory consistency (AR). We are the first to provide an AR-version of the Taubenfeld lock [77].

3. The locks in our benchmark-suite differ in their fairness guarantees; we distinguish between (1) no fairness, (2) FIFO-fairness, and (3) fairness enabled by the Taubenfeld-lock as described in Section 5.2. From this distinction, the relative costs in performance of these fairness guarantees can be gauged.

4. We compare the relative costs in performance of Ada's Hoare-style PO synchronization with state-of-the-art locks, to quantify the performance penalty incurred by the more abstract monitor-style synchronization.

5. We differentiate between single-producer-single-consumer (SPSC) and multi-producer-multi-consumer (MPMC) queues, to obtain quantitative data on the performance improvement related to this specialization.

Our experimental evaluation in Section 6 indicates that the micro-architectural features of a hardware-platform impact the performance with synchronization constructs in subtle ways. We observed that even different models of the same hardware architecture can yield varying results; so no experimental evaluation can be exhaustive. For reproducibility and extension to new platforms, we have therefore open-sourced all of our benchmarks as a GitHub repository [8].

To complement the following brief description of our benchmarks, we refer to the provided source-code under the "`benchmark`" directory in our GitHub repository [8].

5.1 Blocking and non-blocking queue data structures

Non-blocking MPMC queue in Ada: We implemented the non-blocking concurrent queue of Michael and Scott [62] in Ada. To prevent ABA problems [34] in the Ada-implementation, Ada's standard access type had to be modified. We provide the following two approaches, which both used atomic intrinsics from GCC for our implementation.)

1. **Tagged pointers:** For the tagged pointer type [11], we leveraged the fact that the x86_64 architecture uses only the low-order 48 bit of an address. We thus utilized the otherwise unused high-order 16 bit as a counter to track the number of accessing threads. Because the tagged pointer is not a standard type in Ada, additional functions to extract/set a pointer and a counter in a tagged pointer have to be provided. A function to convert this pointer to a standard access type is also necessary. Because our 64 bit tagged-pointers require an 8-byte atomic RMW (CAS) operation, we refer to this queue-implementation as "**Ada CAS-8 MPMC**".

2. **Record-based approach:** Our second approach uses a record type which consists of an 8-byte pointer (access-type) and an 8-byte counter. Thereby

it is possible to avoid the performance overhead such as extracting the address-information from a tagged pointer. However, expensive 16-byte atomic operations are required to perform RMW operations on this record type. We refer to this queue-implementation as "**Ada CAS-16 MPMC**".

Non-blocking MPMC queue in C++ (Boost): The Boost libraries [2] provide industrial-strength, portable C++ code for many algorithms and data structures. We chose Boost's lock-free MPMC queue [3] for our evaluation. As with our Ada implementation described above, this MPMC-queue is based on Michael and Scott's work [62].

Blocking MPMC queue in Ada (POs): A blocking implementation of an MPMC queue in Ada has been chosen from [76, A.18.27] and [76, A.18.28]. It uses a PO for synchronizing access to the queue.

Blocking MPMC queue in Ada (TAS): Borrowing ideas from [68], we adapted the Ada approach with POs to design a blocking queue where the enqueue and dequeue operations employ blocking synchronization via atomic test-and-set (TAS) operations; so the enqueue/dequeue operations are mutually exclusive. TAS operations were implemented with intrinsics provided by GCC.

Non-blocking SPSC queue (Boost): Boost's wait-free SPSC queue data structure is exclusively shared between two threads: one producer and one consumer. Contention from multiple producers and multiple consumers is eliminated by design. C++11 acquire-release semantics are used for implementing push and pop methods.

Non-blocking SPSC queue (cache-aware): B-Queue [82] is a recent research result in the area of wait-free SPSC queues. B-Queue reduces the shared state between producer and consumer, i.e., the queue is the only shared variable left between the two. It is used for both data communication and synchronization. B-Queue takes advantage of batching —it reduces the number of shared-memory accesses and cache-coherence overhead by setting the size of a batch to multiples of the underlying architecture's cache-line size.

5.2 Locks

We employed atomic RMW operations in C++11 to implement four types of locks: TAS-locks, test-and-test-and-set (TATAS) locks [69], array locks [19] and CLH queue locks [57]. Array locks and CLH queue locks provide FIFO-fairness guarantees. We took the Taubenfeld fair synchronization algorithm for non-blocking data structures and combined it with a TAS-lock to obtain a fair lock (see description below). We compared these lock implementations to POSIX mutexes, C++11 mutexes and to an Ada lock-implementation employing a PO. We implemented Peterson's algorithm for 2-thread mutual exclusion [64] and the Filter algorithm for N-thread mutual exclusion [43] as examples of locks without atomic RMW operations. For all locks that we implemented (i.e., all except the POSIX and C++11 mutexes and the Ada PO), we provide two versions: one using sequential consistency (SC), and one using acquire-release consistency (AR).

TAS locks SC & AR: The simplest spin-lock can be implemented with a TAS instruction, which first appeared in the IBM S/360 [36]. Our implementation uses an atomic variable on which the TAS operation is invoked. Control keeps spinning while the returned value is `true`. Once `false` is returned, the caller achieves exclusive permission to enter the critical section. The critical section is completed by resetting the `locked` variable to `false`.

A known performance drawback of this approach is the excessive invalidation of the cache-line that holds variable `locked`. In particular, an invocation of the TAS instruction on one core will invalidate this (shared) cache-line in all other cores. Threads spinning on those other cores will cause sub-sequent re-reads of the cache-line from the core on which TAS was executed.

TATAS locks SC & AR: A TATAS lock is a variant of the TAS lock designed to mitigate the TAS lock's cache-line invalidation problem [69]. The revised function `lock()` implements a read-only spin prepending the costly TAS operation. The read-only spin operation keeps the cache-line holding the `locked` variable in shared state across all competing cores, rather than requiring exclusive access.

Array locks SC & AR: With the TAS and TATAS locks, all threads contend for the same memory location. The Array lock provides FIFO-fairness, and each thread holds an individual flag instead [19]. Calling `lock()` occupies a single flag-slot from an array of flags and spins on a local (cache-) copy of this variable. Function `unlock()` releases the current slot and sets the next slot, on which another thread may spin. In C++, the size of type `bool` is usually smaller than the cache-line size, so *false sharing* [30] may occur. In our implementation, to avoid false sharing we created a custom flag type using padding and aligned memory layout.

CLH queue locks SC & AR: The CLH queue lock is another FIFO-lock; it uses a linked list instead of an array [57]. The CLH queue lock can be implemented with dynamic memory allocation, so the number of participating threads is unlimited. The CLH queue's `lock()` function creates a new list node, pushes it to the tail of the list, and waits until the previous node releases the lock. The `unlock()` function deletes the previous node, which is already released, and clears the `flag` of the current node so that the next thread can acquire the lock. `unlock()` is not allowed to delete the current node because the next thread will check this node.

Ada POLock: This is a straightforward implementation of a lock via Ada's POs. In more detail, a protected procedure is used to ensure mutual exclusion. Because no protected entry is used, FIFO-fairness is not guaranteed [76].

C++ mutex, POSIX mutex: A C++ mutex employing a POSIX mutex underneath, and a POSIX mutex used from within C++.

C++ Peterson SC & AR: We implemented the classical Peterson algorithm [64] for synchronizing two threads [43].

C++ Filter SC & AR: The Filter algorithm is a generalization of Peterson's algorithm for synchronizing $n \geq 2$ threads [43].

C++ Taubenfeld SC & AR: The Taubenfeld algorithm is a fair algorithm for synchronizing n processes [77]. A thread is not allowed to enter the critical

section twice while another thread is waiting to enter the critical section.

6 Experimental results

Table 2: Evaluation platform specifications.

		x86_64	**x86_64_tm**	**ARM_v8**
CPU	Model	Xeon E5-2697 v3	Xeon E5-2699 v4	AWS Graviton
	Microarchitecture	Haswell	Broadwell	Cortex-A72
	cores (sockets/clusters)	28 (2)	44 (2)	16 (4)
	hyperthreads/core	2	2	—
	Clock frequency	2.6–3.6 GHz	2.2–3.6 GHz	2.3 GHz
	RAM size	256 GB	512 GB	32 GB
OS & Tools	Linux distribution	CentOS 7.5.1804	CentOS 7.3	Ubuntu 18.04
	Kernel	3.10.0-229.4.2	4.9.4-1	4.15.0-1043
	GCC/GNAT	7.3.1	6.3.0	8.3.0
	GCC/GNAT opt. level	-O3/-gnatp	-O3/-gnatp	-O3/-gnatp
	LIKWID	4.2.0	4.2.0	ARMv8 branch
	PAPI	5.4.1	5.4.1	5.6.0

We employed three hardware platforms in our experimental evaluation. The key characteristics of these platforms are stated in Table 2. The Broadwell microarchitecture of the x86_64_tm platform contains the Intel transactional synchronization extensions (TSX), which are required with our approach to lock elision. The x86_64 and x86_64_tm platforms provide two hyperthreads per core. To avoid perturbations of measurements from hyperthreading on these platforms, we assigned a dedicated core to each Ada task or C++ thread. The 16 cores of the ARM_v8 platform are part of a single system-on-chip (SoC), consisting of four clusters of four cores each. The ARM_v8 platform constitutes an A1 instance hosted on Amazon EC2 [1]; the other two platforms are non-hypervised servers maintained by the authors of this study.

With our scalability experiments, cores of the first CPU (the first core-cluster on the ARM v8 SoC) were populated consecutively. CPU 2 only received tasks when all cores of the first CPU were occupied. Thus we were able to assess the influence of a multi-socket architecture on scalability. We used the likwid-pin tool [79] for assigning threads to cores. To measure execution time, we instrumented each benchmark program using the PAPI performance counter API [32]. For the versions and optimization levels of the compilers used we refer again to Table 2.

For the lock elision benchmarks, we adjusted the problem size to result in execution times between 2 and 40 seconds. For the remaining performance measurements, we report the median out of 100 runs with 10 million unit-workloads.

All benchmarks ran in a tight loop, which represents the worst-case scenario in terms of synchronization overhead. We chose this setting to control the experiment and avoid any measurement perturbations from application code.

6.1 Lock elision

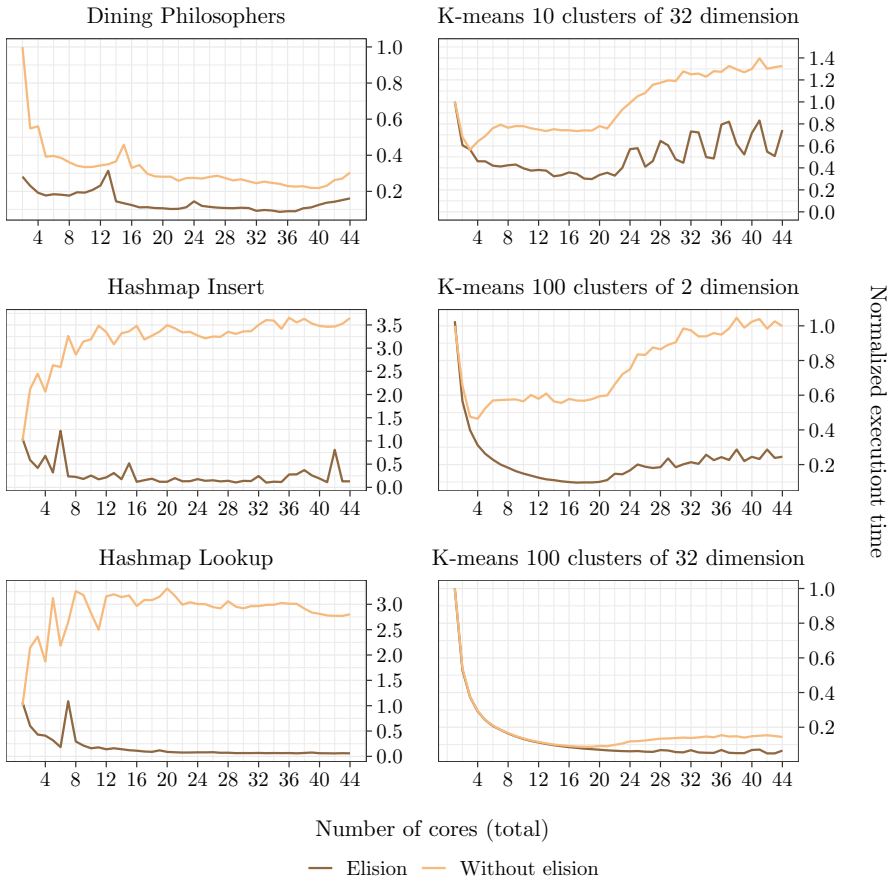


Figure 3: Normalized execution time of lock elision benchmarks.

Figure 3 illustrates a performance comparison between a legacy Ada PO and a PO adapted for lock elision for three different benchmarks: Dijkstra's Dining Philosophers, a concurrent hash-map, and K-means clustering. To highlight the performance improvement from lock elision, each execution time is normalized by the baseline execution time without elision.

For Dining Philosophers, we used a PO as a fork and an Ada task type as a philosopher. Each fork has a state variable to indicate whether it is acquired by some philosopher. The only way to change the fork's state is via protected procedures Acquire and Release, so mutual exclusion is guaranteed. The goal of a philosopher task is to acquire two neighboring forks to have a meal. If any of the two forks is already taken, a task will retry calling the Acquire procedure until it succeeds. A task will release its two forks through the Release procedure,

after having acquired both forks (meals are infinitely fast). We used a locking hierarchy over the forks to avoid deadlock.

To test the scalability of our lock elision, we used an increasing number of tasks (i.e., philosophers). Each philosopher task ran until it had consumed 1 million meals. The top-left diagram of Figure 3 compares the normalized execution time with and without elision. Two normalization factors are: workload per task and execution time of two philosophers without lock elision. Lock elision always showed better performance from 2 to 44 tasks. The transaction abort rate was observed to be 5%–30%.

Our second benchmark, a concurrent hash-map, contains a large table for key-value storage. We implemented it by a PO that contains a protected function `Lookup` and a protected procedure `Insert`. So the PO constitutes a single coarse-grained lock that synchronizes access to the hash-map. A hash-map contains a large amount of shared data, which minimizes the possibility of data conflicts: the benefit from lock elision is maximized.

We tested our concurrent hash-map by randomly generated values. Figure 3 shows the normalized execution time of 50 million insert operations on an empty hash-map, and 50 million lookup operations on a pre-filled hash-map. The number of operations is constant regardless of the number of participating tasks. We observe that a concurrent hash-map without elision does not scale in the number of tasks. This is due to serialization when accessing the PO. On the other hand, the result with elision shows scalability and gets faster when the number of tasks increases. The success rate of transactions was more than 70% for insert and 90% for lookup, proving that data conflicts rarely occur.

K-means clustering, commonly used for data mining, is an algorithm to group vectors in N dimensions into K clusters and calculate the center of the clusters. We selected K-means to show how lock elision is applied to a real-world benchmark. We adopted the STAMP [10, 63] version of the K-means algorithm and implemented it in Ada. The computation of cluster centers can be divided and computed by multiple tasks in an iterative manner. However, the results of concurrent computations must be aggregated and updated to cluster centers at the end of each iteration. The computation is responsible for most of the execution time in the algorithm, thus data conflicts during synchronization should be rare. We used a PO to guarantee mutual exclusion to a cluster center. A cluster center is updated through protected procedures. Other work, such as the computation of cluster centers, is performed in parallel. As a result, the amount of time spent inside the critical section is smaller compared to previous benchmarks.

The second column in Figure 3 depicts the execution time of K-means clustering in different dimensions and clusters. A higher dimension implies more parallelizable work. Larger clusters imply a smaller probability of data conflicts during synchronization. E.g., the K-means algorithm with 10 clusters scales up until 18 tasks, and degrades afterwards due to frequent data conflicts. With 100 clusters of the same dimension, it scales up to 44 tasks. However, the effect from lock elision becomes subtle because the amount of parallelizable work becomes larger. The configuration that benefits most from lock elision

is 100 clusters of two dimensions. It contains large clusters that induce fewer conflicts, and provides a not-too-large amount of parallel work that does not hide the benefit of lock elision. With lock elision we achieved speed-ups of up to 5x.

6.2 Concurrent objects

To mimic Ada compiler-support for *concurrent objects*, which is not available yet, we have implemented the non-blocking COstack from Listing 4 of Section 4.3 by manually translating Ada source-code to C++. (The C++ sources have been made available in our accompanying GitHub repository [8].) This translation constitutes a lowering from the CO abstraction-level to that of C++, which is comparable to our proposed low-level atomics API from Section 4.4. Conceptually, this translation is similar to the lowering of high-level language code to an intermediate representation such as an abstract syntax tree in the front end of a compiler [17]. A major part of the translation is concerned with expanding implicit sync-loops, and we expect this translation to be fully automatable by an Ada+CO ⇒Ada+atomics-API source-to-source translator.

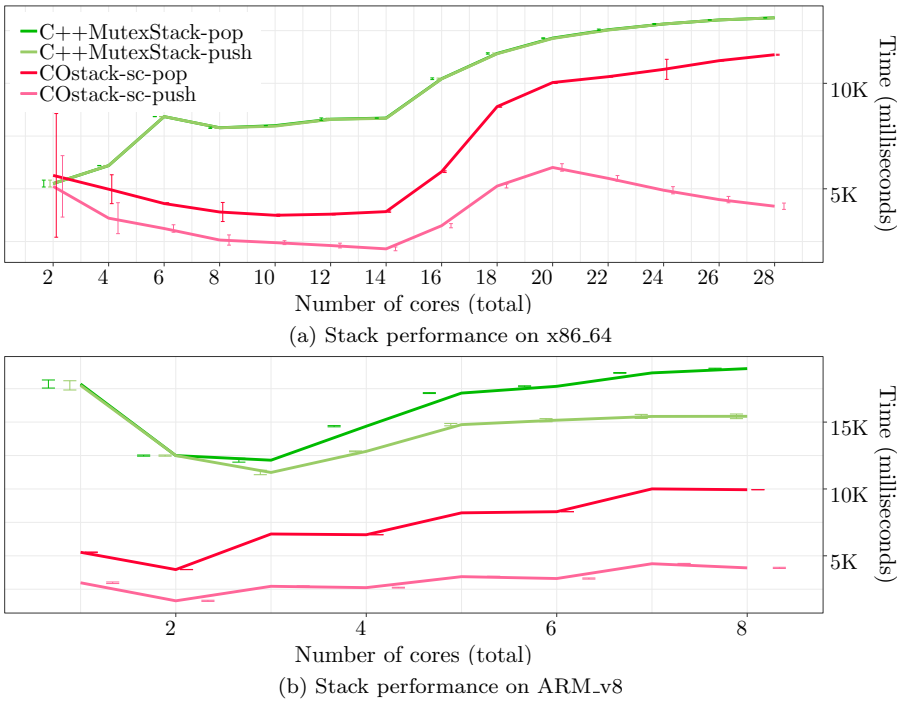
6.2.1 Concurrent object scalability and performance

We evaluated scalability and performance of our non-blocking COstack implementation and compared it with a blocking stack implementation. Unlike use of COstack, hazard pointers are unnecessary in the pop function of the blocking stack; because the ABA problem does not happen inside the blocking stack's critical section, allocated memory can be freed safely.

For the experiment, the participating threads were partitioned into two groups, one group performing a total of 10 million push operations, and the other group conducting the corresponding 10 million pop operations. Work was partitioned statically among the threads in each group.

In terms of the relaxed memory model described in Section 2.2.2 and the algorithm given in [61], the COstack's pop operation needs two write accesses and one read access to the HPlist in program order. The first write is to register the current head node on the HPlist to prohibit other popping threads from reclaiming it illegally. After finishing the load of the head by the RMW operation in line 42 of Listing 4, it is safe to retire the current head. So threads write a `nullptr` on the HPlist for reclamation, which is the second write needed in the pop operation. The load occurs in operation `Retire`.

Figures 4a and 4b show that non-blocking synchronization surpasses blocking synchronization of scalability and performance on both platforms. Because pop needs more memory fences than push to guarantee SC, the non-blocking pop operation shows worse performance than the push operation on both platforms. The performance gap between pop and push increases with the number of threads.



(a) Stack performance on x86_64

(b) Stack performance on ARM_v8

Figure 4: Scalability and performance of non-blocking and blocking stack implementations.

6.2.2 Cyclomatic complexity of COs vs. low-level atomics

Comparing the cyclomatic complexity [59] of the C++11 version and the Ada version with our proposed language extensions, there is a notable difference. The C++ code with low-level atomics contains almost twice as many loops as the Ada code with COs (Ada: 3 loops, C++: 5 loops). This is primarily due to the implicit sync loops in Ada. Each loop increases the cyclomatic complexity by one. International safety standards like ISO 26262 [49] and IEC 62304 [45] mandate coding guidelines that enforce low code complexity.

Non-blocking synchronization is conceptually difficult when exposed to the programmer. Using implicit sync loops with COs simplifies the control flow of non-blocking synchronization constructs, which we expect to reduce the cognitive load and alleviate the challenges programmers face with non-blocking synchronization.

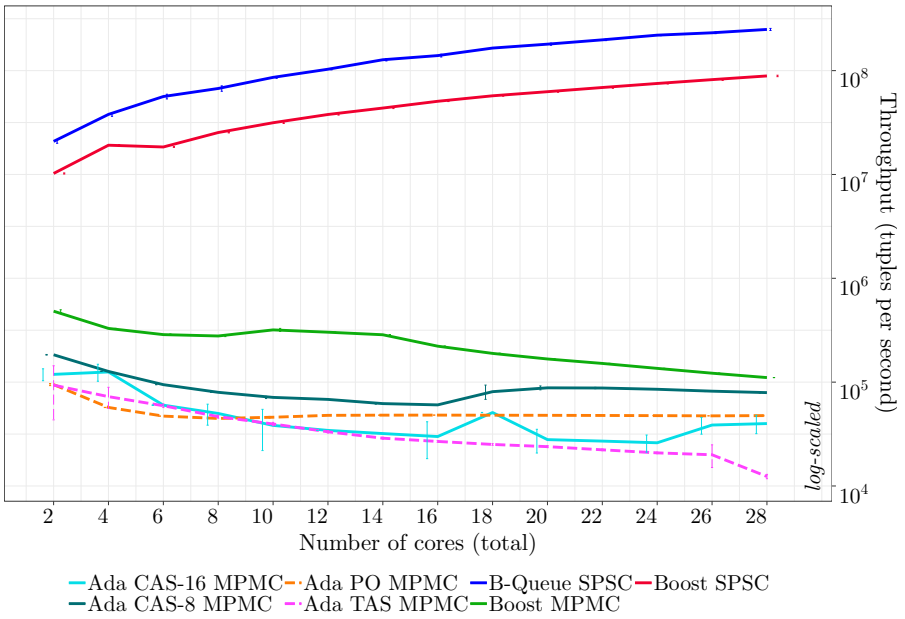


Figure 5: Queue throughput on x86_64.

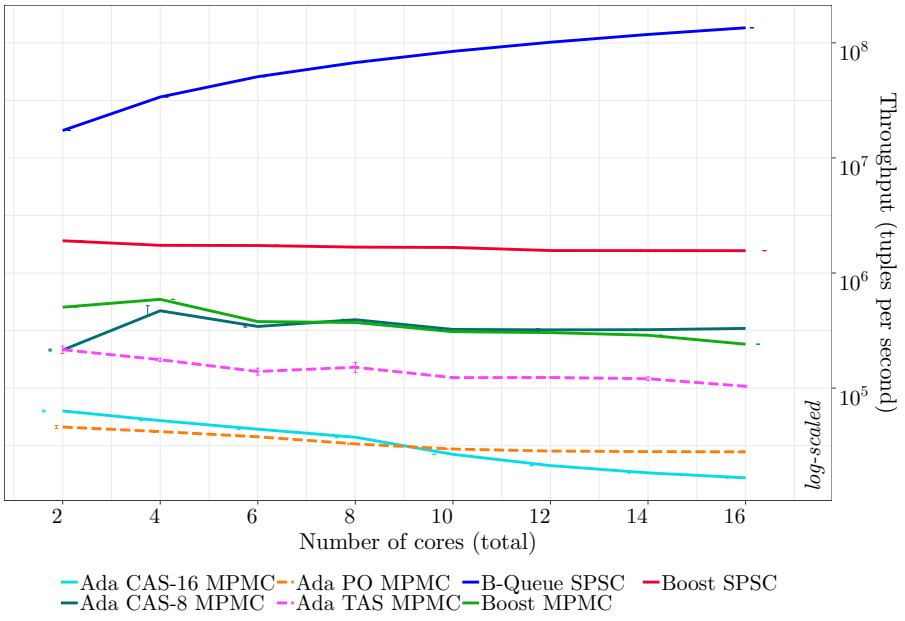


Figure 6: Queue throughput on ARM_v8.

6.3 Synchronization benchmarks for queues and locks

6.3.1 Performance of queue data structures

We obtained the *throughput* of our queue implementations as the number of tuples per second that pass through a particular queue. For each queue we scaled the number of threads from one producer and one consumer up to 14 producers and 14 consumers. Figures 5 and 6 depict the throughput of our queue implementations on the x86_64 and ARM_v8 platforms, respectively. The ordinate in both figures is log-scaled, which is necessitated by the performance gap between SPSC and MPMC queues. The vertical lines on each line of the data-series correspond to the standard deviation of the measured throughput values as the number of cores is scaled.

Comparing the two Ada non-blocking queues (Ada CAS-8/16 MPMC), 8-byte CAS shows better performance than 16-byte ("double-width") CAS. This implies that a 16-byte CAS instruction is computationally more expensive than its 8-byte counterpart, with the difference being more pronounced on ARM_v8. The non-blocking Ada CAS-8 MPMC queue shows higher throughput than the blocking Ada PO MPMC and Ada TAS MPMC queues. Except for 16-byte CAS (which stretches the limits of the 64-bit architectures used with these experiments), the non-blocking queues show consistently higher throughput than their blocking counterparts.

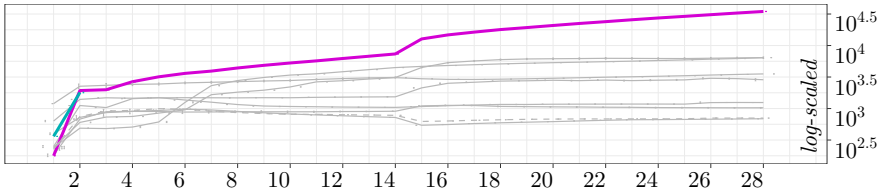
Although the Boost MPMC and Ada CAS-8 MPMC queues constitute identical concurrent data structures, the former is more efficient than the latter on x86_64. But on ARM_v8, they yield comparable throughput except for the tails, where Boost dominates for lower numbers of threads, and Ada dominates for higher numbers of threads.

The SPSC queues achieve substantially higher throughput than the MPMC queues. The reason for this vast difference is the number of synchronizing tasks per queue. With an SPSC queue, a single producer contends with a single consumer for access to the shared queue state. E.g., the experiment with 28 tasks comprises 14 SPSC queues, with one producer/consumer pair per queue. In contrast, with an MPMC queue all synchronizing tasks contend for one single shared queue. The cache coherence overhead increases with the number of synchronizing threads. But the MPMC queues achieve lower throughput even for two tasks, which implies that synchronizing multiple tasks on each end of a queue incurs higher synchronization overhead compared to the SPSC case.

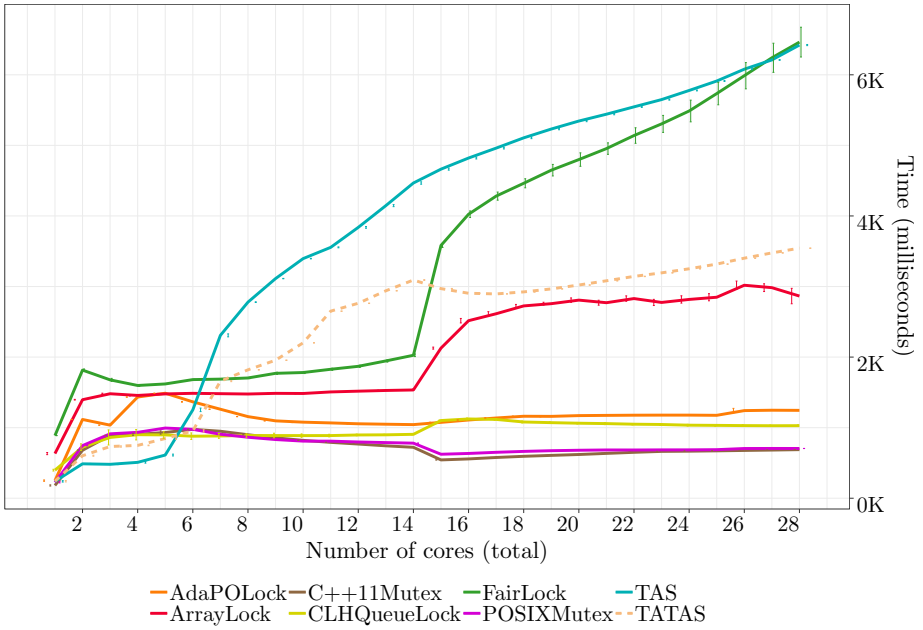
6.3.2 Performance of locks

Execution times of locks have been obtained on the x86_64 platform (Figure 7) and on the ARM_v8 platform (Figure 8). Because the Peterson and Filter locks do not apply atomic RMW operations, they are an order of magnitude slower. Their performance in relation to the remaining locks has been factored out in Figure 7a and Figure 8a.

For the locks that employ atomic RMW operations, performance is diverse and there are no hard-and-fast rules on which lock to use under which conditions.



(a) Filter lock (scaled to 28 cores) and Peterson lock (scaled to 2 cores).



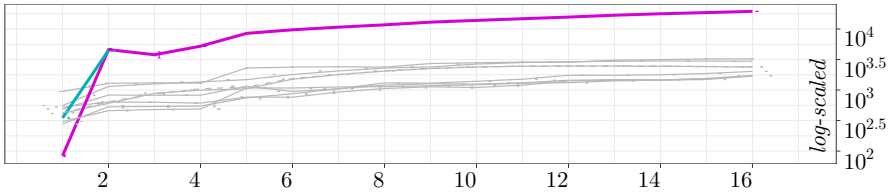
(b) Performance of locks using atomic RMW operations.

Figure 7: Scalability of lock primitives on x86_64.

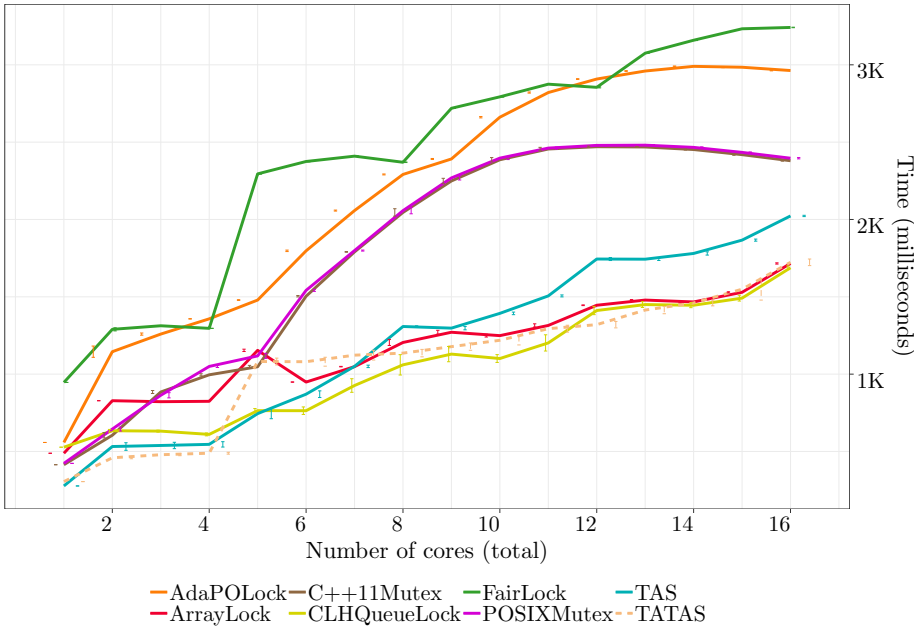
But we identify the following determinants for lock performance: (1) microarchitecture, (2) single versus multiple sockets/clusters, (3) number of competing threads, and (4) implementation characteristics of a lock (in particular those which affect cache coherence overhead and the frequency and type of atomic operations). Hypervised versus non-hypervised workloads may impact performance; but this consideration is not controlled in our experiments (the ARM_v8 platform is hypervised, but the OS of the x86_64 platform runs on the bare metal). The following list summarizes our main observations.

(Observation 1) The C++11 `std::mutex` class employs a POSIX mutex, which uses the Linux kernel's `futex` [4] system call under the hood. Due to this similarity, the two locks show almost identical performance on both platforms.

(Observation 2) The AdaPOLock is a lock implemented with the help of an Ada



(a) Filter lock (scaled to 16 cores) and Peterson lock (scaled to 2 cores).



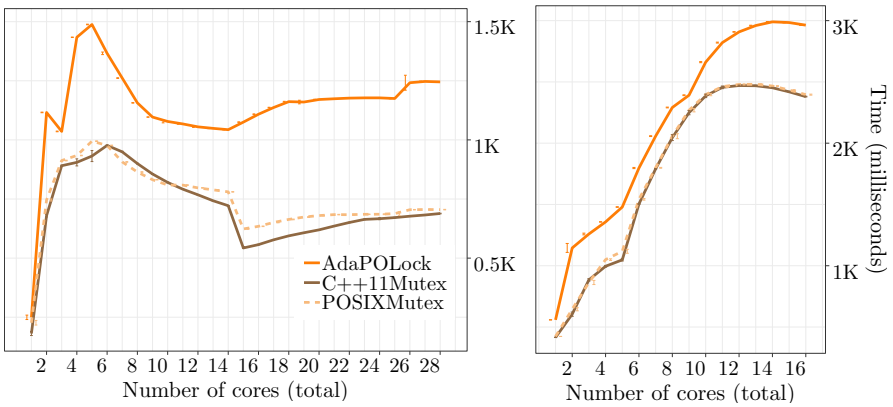
(b) Performance of locks using atomic RMW operations.

Figure 8: Scalability of lock primitives on ARM_v8.

PO. For synchronization it uses a single protected procedure as the critical section. Because an Ada PO employs a POSIX mutex under the hood, performance follows the trend of the C++11Mutex and the POSIXMutex locks. Nevertheless, the AdaPOLock incurs overhead which we attribute to precedence (fairness) guarantees related to the eggshell model [66].

(Observation 3) The x86_64 platform is a two-socket system with 14 cores per CPU. Locks that are substantially negatively impacted on this platform when scaled above 14 cores include the CLHQueueLock, the ArrayLock and the FairLock. All three locks use atomic loads to spin on a flag variable, whereas the TATAS and POSIX mutex-based locks use an atomic RMW operation. Performance of the latter locks slightly improves above 14 cores. (This behavior is repeatable on the x86_64 platform.)

The ARM_v8 SoC consists of four clusters of four cores. Performance



(a) Performance of PO lock and mutexes on x86_64.

(b) Performance on ARM_v8.

Figure 9: Performance comparison of POSIX mutex, C++ mutex and PO lock.

of the FairLock substantially decreases with each additional cluster. For the CLHQueueLock, the ArrayLock and the TASLock, performance is adversely affected when a cluster gets fully populated.

(Observation 4) The entry section of the FairLock needs to obtain the state of each competing thread, which incurs the highest cache coherence overhead among all surveyed locks and results in a performance penalty. Scaling to multiple sockets/clusters has a profound negative impact.

(Observation 5) Fairness guarantees impact performance. The CLHQueueLock and the ArrayLock provide FIFO fairness, whereas the FairLock guarantees that no thread can enter a critical section twice while another thread is waiting. On x86_64, both ArrayLock and FairLock show clear performance penalty compared with the unfair C++ and POSIX mutexes. The CLHQueueLock shows this performance penalty only when scaled to larger numbers of threads.

On the ARM_v8, the C++ and POSIX mutexes exhibit weak performance, which makes the CLHQueueLock and the ArrayLock viable alternatives at the additional benefit of providing FIFO fairness.

On both platforms the FairLock suffers the highest performance penalty, so its weaker fairness-guarantee is more expensive to provide than FIFO-fairness.

PO-lock versus mutex performance:

As outlined in (Observation 2) above, when Ada's monitor-style PO is used for the conceptually simpler mutual exclusion synchronization problem, it induces a noticeable performance overhead compared to a lock. Figure 9 illustrates the result of this comparison. Mutual-exclusion synchronization implemented

with Ada POs ("AdaPOLock") is considerably slower on both the x86_64 and ARM_v8 platforms compared to C++ and POSIX mutexes. This indicates that the overhead of the Ada PO synchronization mechanism does not diminish with synchronization patterns that only require mutual exclusion. Making the GNARL runtime parameterizable to the synchronization problem at hand may help to alleviate this performance issue.

Performance gains under relaxed memory consistency:

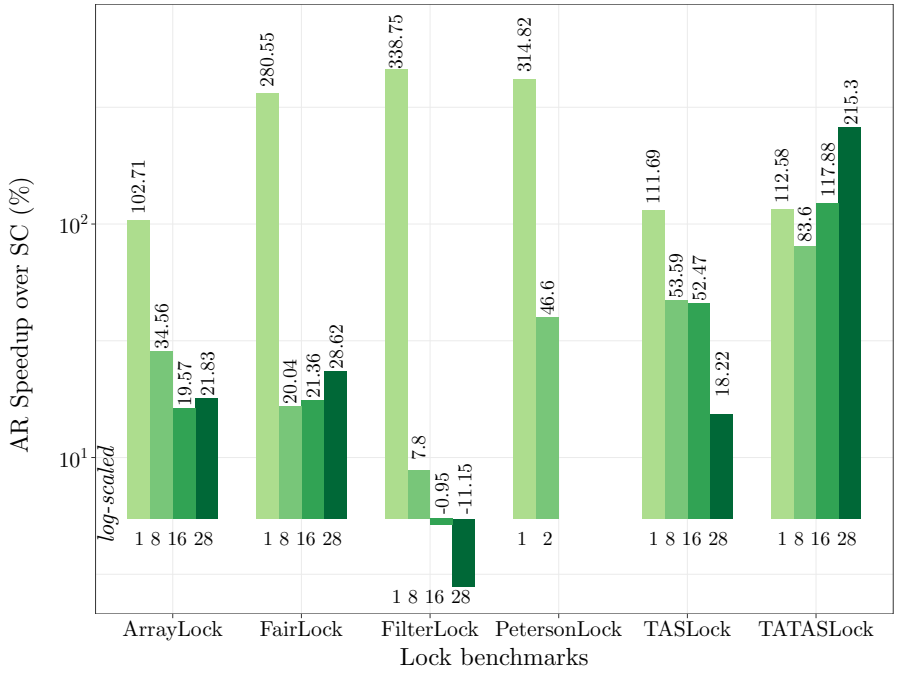


Figure 10: Comparisons of blocking primitives on x86_64.

We have chosen several locks and relaxed them from sequential consistency (SC) to acquire-release consistency (AR) using the C++11 memory consistency model. The research question of interest is whether relaxing the memory consistency is a viable approach to improve performance. We found the speedup gains of an AR relaxation over its SC counterpart for x86_64 and ARM_v8 in Figure 10 and Figure 11. (PetersonLock was designed for two threads only.)

(Observation 6) *Maximum speedup:* On the x86_64 platform, AR relaxation can speed up execution up to 338.75 %. With the exception of the Filter lock, performance improves by at least 18%. On the ARM_v8 platform, maximum speedup of 89.71 % is obtained with four threads of the Filter lock.

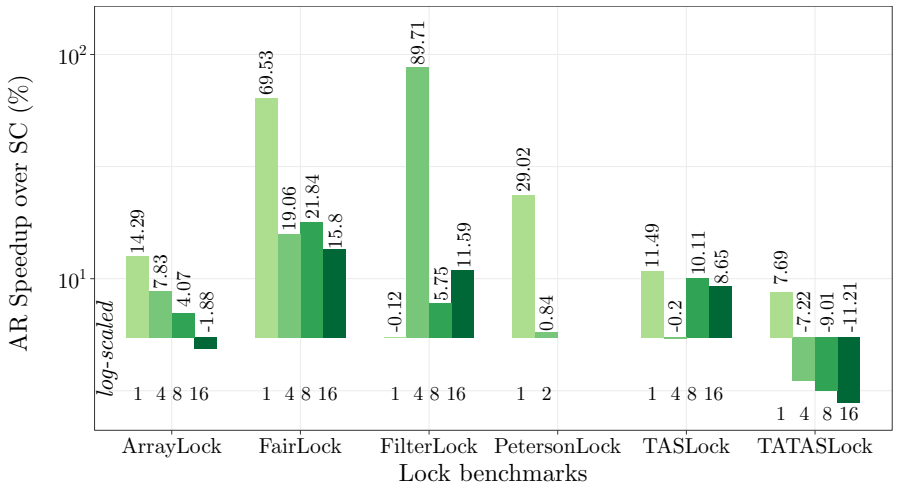


Figure 11: Comparisons of blocking primitives on ARM_v8.

(Observation 7) Although the ARM CPU of the ARM_v8 platform provides a weaker hardware memory model than the x86_64 platform, it benefits less from relaxations.

(Observation 8) *Single-thread performance gains:* A dominant trend across all benchmarks is that the highest speedup from relaxation is gained for a single thread. Speedup decreases in the number of threads, with the exception of the TATAS lock on x86_64.

The speedup for a single thread is highly relevant, because this case constitutes an uncontended lock; so uncontended locks may benefit significantly from relaxation to AR consistency.

(Observation 9) *Micro-architectural analysis:* We performed a microarchitectural analysis using Intel's *toplev* method [53], to gain insights on the large speedups for single threads on the x86. We found that the single-thread workload of the SC version of these synchronization algorithms is nearly 100% backend bound, which means that the CPU cannot use any execution ports due to the pipeline stalls related to the memory fences (the CPU spends most of the time waiting for the synchronization memory accesses to complete). Contrary, the AR-version of these synchronization algorithms is only about 65% backend bound and shows a higher utilization of execution ports. Once threads are added, the workloads become memory bound, due to the cache coherence overhead. The cache coherence overhead dominates the gains from the memory relaxation, which results in smaller speedups for higher numbers of threads.

7 Related work

7.1 Lock elision

In [54], lock elision is applied with POSIX mutexes in the glibc library. Intel TSX is the underlying hardware transactional memory. So-called "adaptive elision" detects transactional aborts and disables lock elision for a certain time for unsuccessful transactions. Any C program using the glibc library can use this elision mechanism.

Yoo et al. [85] aggregated several benchmarks and applied Intel TSX in a high-performance computing environment. The benchmark suite is implemented in C/C++, which does not contain monitor constructs. Their evaluation is restricted to a fourth-generation Intel Core processor with 4 cores. In contrast, our benchmarks improve monitors (Ada POs) by adopting lock elision. Our experiments are conducted on a 2 CPU (44 cores) Xeon E5-2699 v4 platform and we show scalability up to 44 cores in almost all cases.

Preliminary work on Ada's POs with transactions is described in [31]. The term "transaction" as used in [31] is restricted to a single atomic RMW operation, which disallows use of loop statements and accesses to multiple memory locations inside a transaction. Protected entries are not supported. These restrictions are due to the use of hardware atomic primitives at a time when hardware transactional memory was not available. None of these restrictions pertain to hardware transactional memory. But a subtle distinction is worth noting: Intel TSX does not guarantee that a transaction will eventually succeed (thus a fall-back path is needed, as outlined in Section 3.1). Therefore, lock elision based on Intel TSX cannot be non-blocking in the general case. In contrast, the x86 atomic RMW operation `lock cmpxch` [48] has no such restriction. So there is a fundamental trade-off wrt. general-case progress guarantees versus the complexity of transactions between the approach from [31] and our approach.

Unlike lock-based synchronization, lock-free data structures use atomic primitives to synchronize access to shared data. However, building lock-free data structures using fine-grained parallelism is challenging for programmers. In contrast, lock elision allows a programmer to use a coarse-grained lock which exhibits optimistic concurrent execution. Nevertheless, Simple Components [9] and NBAda [7] provide packages of lock-free data structures for Ada. NBAda, but is written in Ada 95 and not available for Ada 2012.

7.2 Concurrent objects

It has been acknowledged [14] that Ada 83 was perhaps the first widely used high-level programming language to give strong support for shared-memory programming. The approach taken with Ada 83 and later language revisions was to require legal programs to be free from synchronization errors, which is the approach taken with SC-for-DRF. In contrast, for reasons of safety and security of its sandboxed execution environment, Java attempts to bound the semantics of programs with synchronization errors. But the original memory model

for Java had to be revised [58]. Later it was even found to be unsound with standard compiler optimizations [80].

Because Ada supports tasks as first-class citizens, it is well-positioned to provide a strict memory model in conjunction with support for non-blocking synchronization. Ada's tasks prohibit inconsistencies resulting from thread functionality provided through libraries [27]. "SC-for-DRF" and two relaxations were formalized for C++11 [28]. It was later adopted for C11, OpenCL 2.0, and for X10 [86]. The C++11 ISO standard (cf. [50, 84]) has already made available a strict memory consistency model. We think that the C++11 effort was not entirely successful for safety and user-friendliness. We are, however, convinced that these challenges can be met in an upcoming Ada standard.

AUTOSAR (AUTomotive Open System ARchitecture) is a worldwide development partnership of vehicle manufacturers, suppliers, service providers, and companies from the automotive electronics, semiconductor, and software sectors [21]. Many papers have shown that the AUTOSAR community is aware of the problems in conjunction with blocking synchronization. AUTOSAR tries to avoid these problems (deadlocks, livelocks, ...) by several means such as no nested locks and bounded blocking for mutexes. There is a trend to use the advantages of non-blocking synchronization [29, 22, 23]. Even blocking analysis for several types of spin locks has been tackled [83]. If Ada brings safe language features for non-blocking synchronization, the automotive systems domain could benefit from using Ada for system implementation.

7.3 Blocking synchronization constructs

Hardlock [75] is a hardware locking unit that guarantees fairness among all competing threads and implements lock and unlock operations that take no more than two clock cycles. It is designed to run on-chip in the main processing unit; the hardware is implemented and tested with 8 cores on a Cyclone 4 FPGA.

Kaiser et al. compare the monitor implementations of Java, C#, and Ada [52]. Ada's PO is designed to comply with an *eggshell* semantics, which guarantees precedence for the queued tasks inside the eggshell over the tasks contending for the lock on the eggshell [66]. Unlike Ada, waiting threads in Java and C# contend for monitors in an unordered manner.

7.4 Profiling of synchronization overhead

Sinclair et al. investigate relaxed memory consistency models for heterogeneous architectures [72]. Their experimental evaluation of relaxed atomics comprises a set of microkernels and applications. Evaluation is conducted on a simulator of an integrated CPU–GPU system. The HeterSync [73] benchmark suite combines the benchmarks for relaxed atomics from [72] with those of synchronization primitives such as mutexes, semaphores, and barriers. Scalability of the relaxed memory consistency benchmarks from [72] is investigated in a simulation of a GPU of a tightly coupled CPU–GPU system. By contrast, we

report measurements on performance and scalability of relaxed atomics for lock implementations on two contemporary multicore/multi-socket CPU systems.

A comprehensive study on the performance overhead of synchronization is presented in [33]. Multiple synchronization levels are investigated, from the cache coherence protocol, atomic RMW operations, locks and message passing, up to application-level concurrent software. Despite its breadth and depth, this study is not directly comparable to our performance results, because their libslock benchmark's C-implementation employs GCC intrinsics, memory barriers and inline assembly rather than the programming primitives of the C++11 memory model.

An extensive performance evaluation of 27 lock algorithms in the context of 35 applications is presented in [13]. By contrast, our study extends to blocking and non-blocking data structures and relaxed atomics. We evaluate the performance of all constructs in isolation rather than in the context of an application.

Gramoli [40] compared the performance of five synchronization techniques with 31 concurrent data structures on the Intel, Ultra-SPARK-T2, and AMD platforms. CAS-based synchronization tends to provide the best performance compared with locks, transactions, and synchronization based on copy-on-write or read-copy-update. The study does not consider relaxed atomics.

7.5 Progress guarantees

A substantial body of work investigates whether algorithms with weaker progress guarantees are practically wait-free, given certain assumptions about the execution environment. Alistarh et al. [18] show that under a stochastic scheduler, the class of so-called single compare-and-swap universal algorithms is wait-free with probability 1. The performance of the same class of lock-free algorithms is analyzed in [20].

Wait-freedom can be obtained for lock-free algorithms at a slight performance loss, if an algorithm runs a lock-free fast path and reverts to a wait-free slow path if the fast path failed [55, 78].

8 Conclusions and future work

We have proposed two techniques to support non-blocking synchronization in Ada. Lock elision of POs replaces coarse-grained locks by optimistic concurrent execution. This transformation is transparent to the programmer; that is, programmers use the blocking synchronization mechanism of Ada POs, and the language runtime system elides the lock to conduct parallel execution of critical sections. We have achieved speed-ups of up to a factor of 5x with a selection of benchmarks by employing the Intel TSX hardware transactional synchronization extensions.

Concurrent objects are a novel programming primitive to encapsulate the complexity of non-blocking synchronization in a language-level construct. The

entities of a CO execute in parallel, due to a fine-grained, optimistic synchronization mechanism framed by the semantics of concurrent entry execution.

We have evaluated the performance improvements achievable with relaxed memory consistency models on the x86 and ARM v8 architectures. Our experiments show that relaxing synchronization constructs can yield significant performance gains, but that care must be taken to balance scalability and performance. A study on the effects of relaxed ordering constraints on the RISC-V architecture is ongoing.

Because the Ada programming language targets safety-critical applications, it remains an open question whether the complexity of *relaxed* memory consistency should be exposed to the programmer, and if so, at what level of the language. Given the complexity of relaxed memory consistency models, a detailed investigation on the trade-offs between programmability and obtainable performance is needed.

Future container libraries for Ada should include lock- and wait-free data structures. We have produced evidence that, for example, different implementations for lock-free MPMC and SPSC queues should be provided. We plan to build a prototype implementation of the source-to-source translator for concurrent objects as outlined in Section 6.2.

9 Acknowledgments

This research was supported by the Austrian Science Fund (FWF) project I 1035N23, and by the Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF), funded by the Ministry of Science, ICT & Future Planning under grant NRF2015M3C4A-7065522.

References

[1] Amazon EC2 A1 instances. `https://aws.amazon.com/ec2/instance-types/a1/`. [Online; accessed: 2019-07-06].

[2] Boost library. `https://www.boost.org/`. [Online; accessed: 2019-01-14].

[3] Boost library lock free queue. `https://github.com/boostorg/lockfree/blob/develop/include/boost/lockfree/queue.hpp`. [Online; accessed: 2019-01-14].

[4] futex(2) — Linux manual page. `http://man7.org/linux/man-pages/man2/futex.2.html`. [Online; accessed: 2019-02-02].

[5] Intel Developer Zone: Pause Intrinsic. `https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-pause-intrinsic`. [Online; accessed: 2019-07-16].

[6] Lock elision anti-patterns. `https://software.intel.com/en-us/articles/tsx-anti-patterns-in-lock-elision-code`. [Online; accessed: 2019-07-16].

[7] NBAda: Non-blocking data structures for Ada Web site. `http://www.gidenstam.org/ada/Non-Blocking/`. [Online; accessed: 2019-07-16].

[8] Shared-memory synchronization benchmark suite for multicore architectures GitHub page. `http://github.com/shm-sync/shm_sync`. [Online; accessed: 2020-01-22].

[9] Simple Components Web site. `http://www.dmitry-kazakov.de/ada/components.htm`. [Online; accessed: 2019-07-16].

[10] STAMP GitHub page. `https://github.com/kozyraki/stamp`. [Online; accessed: 2019-07-16].

[11] Tagged pointer Wikipedia entry. `https://en.wikipedia.org/wiki/Tagged_pointer`. [Online; accessed: 2019-01-28].

[12] The world's simplest lock-free hash table, Preshing on programming blog. `http://preshing.com/20130605/the-worlds-simplest-lock-free-hash-table/`. [Online; accessed: 2019-07-16].

[13] Multicore locks: The case is not closed yet. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 649–662, Denver, CO, June 2016. USENIX Association.

[14] S. V. Adve and H.-J. Boehm. Memory models: A case for rethinking parallel languages and hardware. *Commun. ACM*, 53(8):90–101, Aug. 2010.

[15] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, Dec. 1996.

[16] S. V. Adve and M. D. Hill. Weak ordering—a new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, pages 2–14, New York, NY, USA, 1990. ACM.

[17] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[18] D. Alistarh, K. Censor-Hillel, and N. Shavit. Are lock-free concurrent algorithms practically wait-free? *J. ACM*, 63(4), Sept. 2016.

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

[20] A. Atalar, P. Renaud-Goud, and P. Tsigas. Analyzing the performance of lock-free data structures: A conflict-based model. In Y. Moses, editor, *Distributed Computing*, pages 341–355, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[21] AUTOSAR. `http://www.autosar.org`. [Online; accessed 2019-07-15].

[22] AUTOSAR. Guide to Multi-Core Systems, V1.1.0, R4.1, Rev3, 2014.

[23] AUTOSAR. Overview of Functional Safety Measures in AUTOSAR, Release 4.3.0, 2016.

[24] J. Barnes. *Ada 2012 Rationale: The Language – the Standard Libraries.* Springer LNCS, 2013.

[25] J. Blieberger and B. Burgstaller. Safe Non-blocking Synchronization in Ada 202x. In A. Casimiro and P. M. Ferreira, editors, *Ada-Europe'2018 International Conference on Reliable Software Technologies*, pages 53–69, LNCS 10873, 2018. Springer-Verlag. Vortrag: Reliable Software Technologies - Ada-Europe, Lissabon; 2018-06-18 – 2018-06-22.

[26] J. Blieberger and B. Burgstaller. Safe Non-blocking Synchronization in Ada 202x. *CoRR*, abs/1803.10067, 2018.

[27] H.-J. Boehm. Threads cannot be implemented as a library. *SIGPLAN Not.*, 40(6):261–268, June 2005.

[28] H.-J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 68–78, New York, NY, USA, 2008. ACM.

[29] N. Böhm, D. Lohmann, W. Schröder-Preikschat, and F. Erlangen-Nuremberg. A comparison of pragmatic multi-core adaptations of the AUTOSAR system. In *7th annual workshop on Operating Systems Platforms for Embedded Real-Time applications*, pages 16–22, 2011.

[30] W. J. Bolosky and M. L. Scott. False sharing and its effect on shared memory performance. In *USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems - Volume 4*, Sedms'93, pages 3–3, Berkeley, CA, USA, 1993. USENIX Association.

[31] G. Bosch. Lock-free protected types for real-time Ada. *Ada Lett.*, 33(2):66–74, Nov. 2013.

[32] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *Int. J. High Perform. Comput. Appl.*, 14(3):189–204, Aug. 2000.

[33] T. David, R. Guerraoui, and V. Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 33–48, New York, NY, USA, 2013. Association for Computing Machinery.

[34] D. Dechev, P. Pirkelbauer, and B. Stroustrup. Understanding and effectively preventing the ABA problem in descriptor-based lock-free designs. In *13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, ISORC 2010, Carmona, Sevilla, Spain, 5-6 May 2010*, pages 185–192, 2010.

[35] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 157–168, New York, NY, USA, 2009. ACM.

[36] A. D. Falkoff, K. E. Iverson, and E. Sussenguth. A formal description of system/360. *IBM Systems Journal*, 3(2):198–261, 1964.

[37] C. Fu, D. Wen, X. Wang, and X. Yang. Hardware transactional memory: A high performance parallel programming model. *Journal of Systems Architecture*, 56(8):384 – 391, 2010. Special Issue on HW/SW Co-Design: Tools and Applications.

[38] A. Gamatié, X. An, Y. Zhang, A. Kang, and G. Sassatelli. Empirical model-based performance prediction for application mapping on multicore architectures. *Journal of Systems Architecture*, 98:1 – 16, 2019.

[39] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. *SIGARCH Comput. Archit. News*, 18(2SI):15–26, May 1990.

[40] V. Gramoli. More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms. *SIGPLAN Not.*, 50(8):1–10, Jan. 2015.

[41] M. Halpern, Y. Zhu, and V. Janapa Reddi. Mobile CPU's rise to power: Quantifying the impact of generational mobile CPU design trends on performance, energy, and user satisfaction. In *High Performance Computer Architecture (HPCA), 2016 IEEE 22nd International Symposium on*, 2016.

[42] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[43] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2012.

[44] C. A. R. Hoare. Monitors: An operating system structuring concept. *Commun. ACM*, 17(10):549–557, Oct. 1974.

[45] IEC 62304 – Medical device software – software life cycle processes, 2006.

[46] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manuals*, volume 3. May 2019.

[47] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, volume 1. May 2019.

[48] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, volume 2A. October 2019.

[49] ISO 26262 – Road vehicles – Functional safety, 2011.

[50] ISO/IEC. *Working Draft N4296, Standard for Programming Language C++*. ISO, Geneva, Switzerland, 2014.

[51] S. Jeong, S. Yang, and B. Burgstaller. Lock elision for protected objects using Intel transactional synchronization extensions. In *Reliable Software Technologies – Ada-Europe 2017 – 22nd Ada-Europe International Conference on Reliable Software Technologies, Vienna, Austria, June 12-16, 2017, Proceedings*, pages 121–136, 2017.

[52] C. Kaiser, J.-F. Pradat-Peyre, S. Évangelista, and P. Rousseau. Comparing Java, C# and Ada monitors queuing policies: A case study and its Ada refinement. *Ada Lett.*, XXVI(2):23–37, Aug. 2006.

[53] A. Kleen. toplev GitHub page. `https://github.com/andikleen/pmu-tools/wiki/toplev-manual`. [Online; accessed 2019-07-16].

[54] A. Kleen. Scaling existing lock-based applications with lock elision. *Queue*, 12(1):20:20–20:27, Jan. 2014.

[55] A. Kogan and E. Petrank. A methodology for creating fast wait-free data structures. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, page 141–150, New York, NY, USA, 2012. Association for Computing Machinery.

[56] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, Sept. 1979.

[57] P. Magnusson, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. In *Proceedings of 8th International Parallel Processing Symposium*, IPPS '94, pages 165–171. IEEE, 1994.

[58] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 378–391, New York, NY, USA, 2005. ACM.

[59] T. J. McCabe. A Complexity Measure. *IEEE Trans. on Software Engineering*, SE-2(4):308–320, 1976.

[60] M. McCool, J. Reinders, and A. Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann Publishers Inc., 1st edition, 2012.

[61] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, June 2004.

[62] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. 15th ACM Symp. on Principles of Distributed Computing*, pages 267–275, 1996.

[63] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *4th International Symposium on Workload Characterization (IISWC 2008), Seattle, Washington, USA, September 14-16, 2008*, pages 35–46, 2008.

[64] G. L. Peterson. Myths About the Mutual Exclusion Problem. *Information Processing Letters*, 12(3):115–116, 1981.

[65] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 34, pages 294–305. IEEE Computer Society, 2001.

[66] J. Real, A. Burns, J. Miranda, E. Schonberg, and A. Crespo. Dynamic Ceiling Priorities: A Proposal for Ada0Y. In *Reliable Software Technologies – Ada-Europe 2004 – 9th Ada-Europe International Conference on Reliable Software Technologies, Palma de Mallorca, Spain, June 14–18, 2004, Proceedings*, pages 261–272, 2004.

[67] D. A. Reed and J. Dongarra. Exascale computing and big data. *Commun. ACM*, 58(7):56–68, June 2015.

[68] P. Rogers. Gem #93: High Performance Multi-core Programming – Part 1. `https://www.adacore.com/gems/gem-93-high-performance-multi-core-programming-part-1`. [Online; accessed: 2019-01-14].

[69] L. Rudolph and Z. Segall. Dynamic decentralized cache schemes for MIMD parallel processors. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, ISCA '84, pages 340–347, New York, NY, USA, 1984. ACM.

[70] M. L. Scott. *Shared-Memory Synchronization*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2013.

[71] M. L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition, 2016.

[72] M. D. Sinclair, J. Alsop, and S. V. Adve. Chasing away RAts: Semantics and evaluation for relaxed atomics on heterogeneous systems. In *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*, pages 161–174, 2017.

[73] M. D. Sinclair, J. Alsop, and S. V. Adve. HeteroSync: A benchmark suite for fine-grained synchronization on tightly coupled GPUs. In *2017 IEEE International Symposium on Workload Characterization (IISWC)*, pages 239–249, Los Alamitos, CA, USA, oct 2017. IEEE Computer Society.

[74] D. J. Sorin, M. D. Hill, and D. A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Number 16 in Synthesis Lectures on Computer Architecture. Morgan & Claypool, 2011.

[75] T. B. Strøm, J. Sparsø, and M. Schoeberl. Hardlock: Real-time multicore locking. *Journal of Systems Architecture*, 2019.

[76] S. T. Taft, R. A. Duff, R. Brukardt, E. Plödereder, P. Leroy, and E. Schonberg. *Ada 2012 Reference Manual. Language and Standard Libraries - International Standard ISO/IEC 8652/2012 (E)*, volume 8339 of *Lecture Notes in Computer Science*. Springer, 2013.

[77] D. Taubenfeld. Fair synchronization. *Journal of Parallel and Distributed Computing*, 97C:1–10, 2016.

[78] S. Timnat and E. Petrank. A practical wait-free simulation for lock-free data structures. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, page 357–368, New York, NY, USA, 2014. Association for Computing Machinery.

[79] J. Treibig, G. Hager, and G. Wellein. LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*, San Diego CA, 2010.

[80] J. Ševčík and D. Aspinall. On validity of program transformations in the Java memory model. In *Proceedings of the 22nd European Conference on Object-Oriented Programming*, ECOOP '08, pages 27–51, Berlin, Heidelberg, 2008. Springer-Verlag.

[81] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael. Evaluation of Blue Gene/Q hardware support for transactional memories. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 127–136, New York, NY, USA, 2012. ACM.

[82] J. Wang, K. Zhang, X. Tang, and B. Hua. B-Queue: Efficient and practical queuing for fast core-to-core communication. *International Journal of Parallel Programming*, 41(1):137–159, 2013.

[83] A. Wieder and B. B. Brandenburg. On Spin Locks in AUTOSAR: Blocking Analysis of FIFO, Unordered, and Priority-Ordered Spin Locks. In *Proceedings of the IEEE 34th Real-Time Systems Symposium, RTSS 2013*, pages 45–56, Vancouver, BC, Canada, Dec 2013.

[84] A. Williams. *C++ Concurrency in Action*. Manning Publ. Co., Shelter Island, NY, 2012.

[85] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance evaluation of Intel transactional synchronization extensions for high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 19:1–19:11, New York, NY, USA, 2013. ACM.

[86] A. Zwinkau. A memory model for X10. In *Proceedings of the 6th ACM SIGPLAN Workshop on X10*, X10 2016, pages 7–12, New York, NY, USA, 2016. ACM.