This is a preprint of the article "Lock Elision for Protected Objects Using Intel Transactional Synchronization Extensions" by S. Jeong, S. Yang, and B. Burgstaller. This article was published in the proceedings of the 22nd International Conference on Reliable Software Technologies, Ada-Europe 2017, pp. 121-136.

The published version, from Springer International Publishing, is available at https://doi.org/10.1007/978-3-319-60588-3_8.

# Lock Elision for Protected Objects Using Intel Transactional Synchronization Extensions

Seongho Jeong, Shinhyung Yang, and Bernd Burgstaller*

Department of Computer Science, Yonsei University, Seoul, Korea
{seongho.jeong, shinhyung.yang, bburg}@yonsei.ac.kr

**Abstract.** Lock elision is a technique to replace coarse-grained locks by optimistic concurrent execution. In this paper, we introduce lock elision for protected objects (POs) in Ada. We employ Intel Transactional Synchronization Extensions (TSX) as the underlying hardware transactional memory facility. With TSX, a processor can detect dynamically whether tasks need to serialize through critical sections protected by locks. We adapt the GNU Ada run-time library (GNARL) to elide locks transparently from protected functions and procedures. We critically evaluate opportunities and difficulties of lock elision with protected entries. We demonstrate that lock elision can achieve significant performance improvements for a selection of three synthetic and one real-world benchmark. We show the scalability of our approach for up to 44 cores of a two-CPU, 44-core Intel E5-2699 v4 system.

## 1 Introduction

Since the advent of multicore processors, software developers have been relying on multi-threaded software to achieve performance improvements. Multi-threaded software requires synchronization to protect data shared among multiple threads. Locks allow to transform code into a critical section, which is a block of code that can only be executed by one thread at a time. This property of critical sections is called mutual exclusion. 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.

A coarse-grained lock protects a large amount of shared data and thus is prone to become a highly-contended scalability bottle-neck. Fine-grained locking protects shared data at a finer granularity, which allows a higher degree of parallel access because empirically not all threads require access to the same data-item. To achieve fine-grained locking, a programmer must partition a shared data-structure into parts and introduce a mutual exclusion lock for each part. E.g., instead of protecting an entire linked list with a single, coarse-grained lock, individual list nodes can be protected by a lock. This thought-process is error-prone and complex: obtaining locks without a global order among locks (a

* Corresponding author

locking hierarchy) will result in dead-locks, and the higher degree of parallelism associated with fine-grained locking makes race-conditions harder to avoid.

Lock-free programming relies on hardware primitives to provide concurrent operations on data-structures [20,15]. Although such algorithms provide high scalability, they are mostly complex.

Lock elision [22] is a technique to reduce serialization with lock-based code. The key insight with lock elision is that many dynamic data sharing patterns among threads do not conflict and thus do not require the acquisition of a lock. E.g., a concurrent hash-map [11] contains multiple key-value pairs. Two threads updating different keys will not conflict and hence do not require serialization (locking). Serialization is only required among threads updating the same key's value.

With lock elision, a thread will *speculatively* execute a critical section (called 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 thus 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 read from within the transactional region, and 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. 1), and discarded otherwise.

**Definition 1. 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 (adopted from [17]).*

Lock elision requires hardware support to be efficient. Recent CPU architectures from Intel, IBM and Sun/Oracle provide hardware transactional memory (TM) extensions [17,27,13], which allow a processor to dynamically detect data conflicts. (Transactional memory was originally proposed in 1993 by Herlihy and Moss [14].)

In this paper, we focus on lock elision for protected objects (POs) in Ada. As the underlying hardware mechanism we employ Intel TSX [17]. To the best of our knowledge, lock elision for Intel TSX until now has only been attempted with mutual exclusion locks in C and C++ [28,18]. In contrast, Ada's POs [24] implement the monitor concept [16]. 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.

Introducing lock elision with the Ada PO implementation is a promising concept, because it will make coarse-grained concurrent data-structures susceptible to fine-grained locking, at the cost of no or only minor changes of the application source-code. Our paper makes the following contributions:

1. We adapt the GNU Ada run-time library (GNARL) to elide locks transparently from protected functions and procedures.
2. We investigate opportunities and difficulties for lock elision with protected entries. We outline two possible elision schemes for protected entries.
3. We experimentally evaluate our approach for a selection of three synthetic benchmarks and one real-world benchmark. We show the scalability of our approach for up to 44 cores of a two-CPU, 44-core Intel E5-2699 v4 system.
4. We provide programming- and language-design directions to leverage the parallelism obtainable from lock elision with POs in Ada.

The remainder of this paper is structured as follows. In Sec. 2, we discuss lock elision for Ada POs. Sec. 3 contains our experimental results. We discuss the related work in Sec. 4 and draw our conclusions in Sec. 5. For an accessible introduction to Intel TSX, we refer the reader to [18,23].

## 2 Lock Elision with GNARL

To access the Intel TSX instruction set extensions from Ada code, we created a package with a procedural interface to each TSX instruction. The specification of this package is depicted in Fig. 1.

A transaction is started via instruction `xbegin`. Upon execution of `xbegin`, the processor returns the value `XBEGIN_STARTED` in the EAX-register and "memorizes" the next instruction's address. When the processor detects a data conflict, it will abort the transaction and transfer control to this "memorized" address. The processor commits a transaction when it reaches instruction `xend`. The memory updates of a transaction become visible to other processors (and cores) when the transaction commits. A commit happens atomically. Instruction `xtest` allows software to test whether the processor is currently inside a transaction. Instruction `xabort` allows software to explicitly abort the current transaction.

Whenever a transaction aborts, control is transferred to the address "memorized" by `xbegin`. In case of an abort, the processor sets a specific bit in the EAX register to signal the type of conflict which caused the abort. Unless the EAX value is 0xffffffff, which denotes `XBEGIN_STARTED`, software must make a decision whether to retry or fall back to the locking code. Note that there is no guarantee from the processor that a transaction will eventually succeed. Thus, the fall-back path with the conventional lock-based code is required.

We employ inline assembly to emit bytes corresponding to Intel TSX instructions. Fig. 2 depicts our implementation for the `xbegin` instruction. The remaining TSX instructions are implemented in a similar manner.

```
package TSX_inst is
pragma Preelaborate;
  type uint32 is mod 2**32;
  -- Status codes returned by the CPU in the x86's EAX register:
  XBEGIN_STARTED   : constant uint32 := 16#ffffffff#;
  XABORT_EXPLICIT  : constant uint32 := 2**0;
  XABORT_RETRY     : constant uint32 := 2**1;
  XABORT_CONFLICT  : constant uint32 := 2**2;
  XABORT_CAPACITY  : constant uint32 := 2**3;
  XABORT_DEBUG     : constant uint32 := 2**4;
  XABORT_NESTED    : constant uint32 := 2**5;
  function XABORT_CODE (state : uint32) return uint32;

  function xbegin return uint32; -- start transaction
  procedure xend;                -- end transaction
  function xtest return uint32;  -- test for execution inside transaction
  procedure xabort;              -- abort transaction
  procedure pause;               -- processor hint (not part of Intel TSX)
private
  pragma Inline (XABORT_CODE);
  pragma Inline_Always (xbegin);
  pragma Inline_Always (xend);
  pragma Inline_Always (xtest);
  pragma Inline_Always (xabort);
  pragma Inline_Always (pause);
end TSX_inst;
```

Fig. 1: Specification of package TSX_inst to use Intel TSX with Ada

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

Fig. 2: Ada implementation for TSX instruction `xbegin`

### 2.1 Lock Elision for Protected Functions and Procedures

The Ada 2012 RM [24, Chapter 9.5.1(5)] states that execution of a protected procedure requires exclusive read-write access to the PO. Speculative execution of critical sections with elided locks does not fulfill this requirement. Rather, serialization is achieved by re-executing a critical section in case of a transactional abort [18,23].

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 it is released by another task. (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 a data conflict, it commits the updates in the write-set to memory (where they become visible to other tasks). During this entire process, the lock appears to be free to all tasks.

Because transactions tend to abort frequently, we must keep the existing lock-based code as a fall-back solution (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 is released (alike the conventional GNARL implementation).

```
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 procedure Try_Elision
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
```

Fig. 3: Elision of a PO lock in GNARL (in pseudo-code). `Try_Elision` is called from procedure `Write_Lock` before entering a critical section. The call returns either inside of 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.

Fig. 3 depicts the PO lock elision scheme that we implemented within 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 require 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. However, the lock library underneath GNARL does not allow to read a lock without changing it. As a work-around, we introduced a shadow-lock variable per PO lock (as outlined in [5]). 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 used 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 applied 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 [2]. This optimization yielded a considerable performance improvement. Second, a task attempts transactional execution several times before falling back to the lock. As depicted in Fig. 3 (lines 10 and 15), a task tries up to `MAX_RETRY` times to execute a critical section as a transaction. Lastly, 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 re-try 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. Note that 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 of a transaction on the same processor core. (A scenario possible with processors that support hyperthreads.) Note that in line 29 of Fig. 3 a task which encountered a data conflict will wait to allow competing tasks to finish their transactions before re-trying. Procedure `Exponential_Backoff` contains a busy-waiting loop with the iteration count specified by the procedure's argument. The waiting time increases with every unsuccessful re-try ("exponential backoff"). Thereby situations are avoided where tasks keep mutually aborting each other's transactions without overall progress.

Lock elision may not always improve performance. For example, critical sections which routinely lead to capacity overflow will always fall back to the lock.

Different values for the maximum number of attempted lock-elisions and the calibration of the busy-waiting loop (lines 10 and 11 in Fig. 3) will differ across benchmarks and HW platforms. Putting the before-mentioned tuning parameters and the decision whether to elide a PO lock under programmer control (e.g., via a pragma) seems advisable. Alternatively, GNARL itself may be extended with dynamic profiling capabilities to take decisions at run-time. At the present stage, we have hand-tuned these values, as described in Sec. 3.

### 2.2 Lock Elision for Protected Entries

The Ada 2012 RM [24, 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"). The reason for this requirement is likely to avoid starvation, according to the following definition.

**Definition 2. Freedom from Starvation:** *every task that attempts to acquire the PO eventually succeeds (adopted from [15]).*

Obviously, RM Clause 9.5.3(16) restricts the degree of parallelism obtainable with lock elision: consider Task A queued at an open entry and Task B calling the same entry (or another entry, or a protected procedure or function of the PO): Task A is required to proceed first, which requires Task B to serialize irrespective of inter-task data conflicts.

It should be noted that for many parallel workloads, freedom from starvation is not a concern (latency and/or throughput is). E.g., both with the "stencil" and the more general "map" programming pattern from [19], the amount of work is usually constant and known *a priory*. 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. Even if work is not bounded, e.g., with streaming applications, then individual work items will be bounded, and tasks will synchronize at a barrier before moving from one work-item to the next.

A not entirely unrelated issue has been discussed by the Ada Conformity Assessment Authority (ACAA) in June 2016 to allow parallel processing of Ada standard containers [1].

To elide locks from POs with entries, we have envisioned the following two schemes.

**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 any order in parallel, 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, which is added to the read-set of every PO transaction. The task which is about to enqueue will write to the `is_queued`-flag, which will abort all ongoing transactions. Procedure `Try_Elision` from Fig. 3 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 queues are empty.

The core part of the Ada language does not specify the order in which entry queues shall 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 required, 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.

One note is due to the controlling variables that occur in barrier expressions. If these variables are frequently written inside of protected operations, the success-rate of transactions will be negatively impacted. A possible remedy can be to encourage programmers to re-queue entry calls from a barrier with a non-trivial condition to a `true` barrier after the update of control variables has been completed. If the PO selectively performs elision for entries with `true` barriers, only the barrier evaluation and controlling variable update is serialized, and the remaining work of the protected operation can proceed in transactional mode. One example that will benefit from this approach is a grow-able hash-map which allows insert operations up to a certain load-factor and then temporarily suspends to grow the hash-map. The number of elements inserted will be a controlling variable for the barrier of the insert operation, and this variable will be decremented with every insert operation. Without a re-queue to an entry that performs the actual insertion, the transactional success rate can be expected to be very low.

## 3 Experimental Results

To evaluate the effect of lock elision with Ada POs, we selected three synthetic benchmarks: a concurrent linked-list, Dijkstra's Dining Philosophers, and a concurrent hash-map. The purpose of the synthetic benchmarks was to study lock elision in isolation, without perturbing effects that a large application (client) might incur. We investigated one real-world application, K-means clustering, from the Stanford Transactional Applications for Multi-Processing (STAMP) benchmark suite [21,9]. STAMP benchmarks are implemented in C, and we ported the K-means clustering benchmark to Ada.

In our evaluation, we used GCC/GNAT version 6.3.0. Lock elision for protected functions and procedures was implemented in GNAT's run-time system as described in Sec. 2. We evaluated our benchmarks on a 2 CPU Intel Xeon E5-2699 v4 system. Each of the E5-2699 CPUs provide 22 cores with 2 hyper-threads per core. Our evaluation platform runs the CentOS Linux distribution (version 7.3, kernel version 4.9.4-1).

We employed likwid-pin from the LIKWID tool-suite [26,3] to pin Ada tasks onto CPU cores. The rationale for task-pinning is to prevent the Linux kernel scheduler from migrating tasks across CPU cores. Such migrations would otherwise perturb the experiments by (1) increasing the cache coherence overhead and (2) increase the transaction failure-rate. The order of our pinning scheme

was to assign one task to each core of the first CPU, then one task to each core of the second CPU (tasks 1–44 were assigned this way). Note that we did not use the CPU's hyper-threading facility in our experimental evaluation.

All performance measurements were conducted using hardware performance counters. We created a thick Ada binding to the PAPI C-API. PAPI [25,7] is a library which provides a hardware-independent interface to count micro-architectural events occurring in a CPU during program run-time. In particular, we determined CPU cycle counts and the transactional success rate using PAPI. For the latter, Intel TSX-capable CPUs provide two hardware cycle counters: (1) the total number of transactional cycles and (2) the number of aborted transactional cycles.

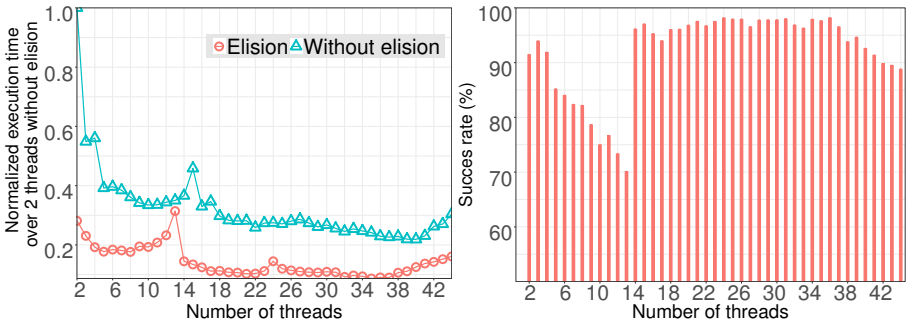
We ran each experiment consecutively for three times. Workloads (i.e., the number of times each task would synchronize at a PO), were adjusted such that a benchmark would execute between 2 and 40 seconds. For each benchmark execution, the execution-time of the longest-running task was obtained. The median of these execution times was reported.

In our lock-elided GNARL implementation, we maintain variable `MAX_RETRY` to set the number of transactional trials before falling back to lock-based synchronization (see Fig. 3). This variable is a tuning-knob; from our experiments, the value of this variable has to be larger than the number of participating tasks to ensure competitive performance. On our 44-core test platform, we set this value to 200, i.e., a task would try 200x to perform a protected function or procedure in transactional mode before falling back to the lock. Constant `BACKOFF` was set to 10. This set-up was used with all benchmarks.

**Linked Lists—A Counter-Example.** We start our evaluation with a data-structure where lock elision is *not* effective. Our linked lists consist of nodes of the following type `Node`, where each node contains a pointer to the next list element. Because nodes are dynamically allocated, each node ends up in a cache-line of its own, although the size of a node is only 16 B on a 64 bit architecture.

```
1 type Node;
2 type pNode is access Node;
3 type Node is record
4   value : Integer;
5   next : pNode;
6 end record;
```

With coarse-grained locking, a single PO would be employed to synchronize access to a linked list of this type. Each list operation (e.g., insert, lookup, ...) will be implemented as a protected operation. If the PO is elided, each list operation constitutes a transaction. However, traversing a linked list of N nodes will accumulate N cache-lines in the read-set of this transaction, which we found to exceed the CPU resources for all but the smallest linked lists. As a consequence, for linked lists beyond this size constraint, transactions will always fail and fall back to the lock-based code. For this particular example, transactions aborted after traversing 100 nodes. This result will vary with the size of the nodes.




(a) Execution Time (b) Transaction Success Rate

Fig. 4: Analysis of the Dining Philosophers algorithm at 1 million synchronization steps ("meals") per philosopher

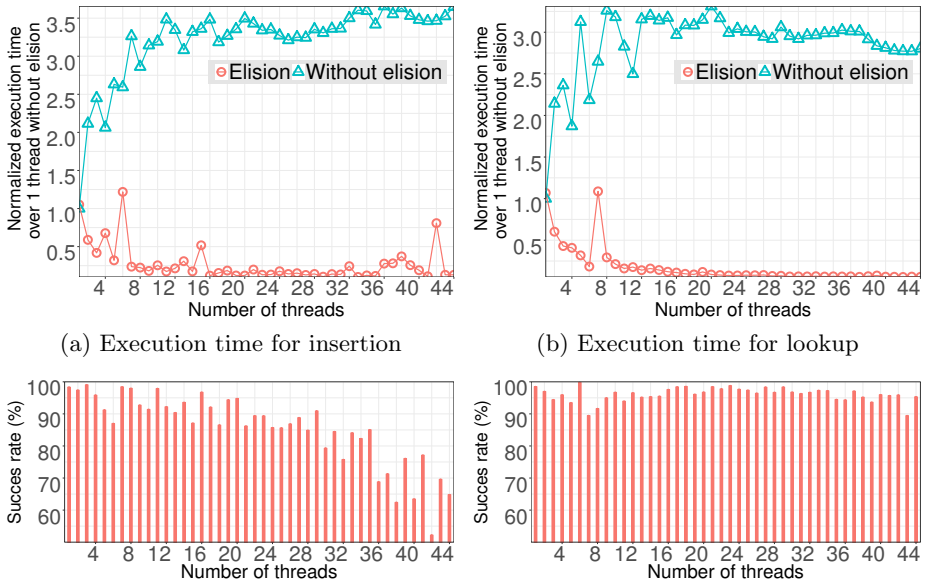
From this example, it becomes clear that lock elision cannot be applied in the general case. Rather, a critical assessment is necessary to decide whether lock elision will be effective for a given use-case. Such an assessment can be done by the programmer (if the programming language provides means to explicitly enable/disable elision), the compiler (through static program analysis), and by the run-time system (through dynamic program analysis).

For the same reason, it seems preferable to conduct lock elision as part of the Ada programming language implementation rather than to divert it to a lower layer such as the glibc library.

Note that for the above example lock elision can be put to work by introducing fine-grained locking of individual list nodes (see, e.g., [15, Chapt. 9.5]). We did not pursue this direction, because it would constitute re-factoring of the application source-code rather than lock-elision of course-grained PO locks.

**Dining Philosophers.** In our implementation of Dijkstra's Dining Philosophers, each philosopher is an instance of an Ada task type and each fork is a PO. Fork acquisition is realized as a protected procedure, which guarantees mutual exclusion. Each fork maintains a state variable that indicates whether a fork is free or taken. When a philosophers acquires a fork, this state variable is set to false. That way, neighboring philosophers aiming at the same fork will find out that the fork is already taken, and re-try calling the acquire procedure until the fork becomes available. After fork acquisition, a philosopher releases the forks immediately by calling the forks' Release procedure (philosophers focus on synchronization only). Scalability of our proposed lock elision is tested by increasing the number of philosophers and forks.

Fig. 4a compares the execution time of the Dining Philosophers algorithm with and without lock elision. Execution times are normalized over the execution time for two philosophers without lock elision. The second normalization factor




(a) Execution time for insertion

(b) Execution time for lookup

(c) Transaction success rate for insertion

(d) Transaction success rate for lookup

Fig. 5: Concurrent hash-map operations (50 million operations each)

is the workload per task (each philosopher performs 1 million synchronization steps, irrespective of the number of participating philosophers).

Lock elision shows superior performance from 2 up to 44 tasks. The performance gap is at the largest with the 2-task baseline case. The gap is the smallest for 13 tasks where lock elision suffers from an abnormal increase in execution time.

**Concurrent Hash-Map.** A concurrent hash-map has high potential to harness the benefits of transactional memory, because data accessed by different tasks tends to be located in different memory locations. The speculative, parallel execution mechanism provided by transactional memory has thus a high chance to succeed.

We implemented a concurrent hash-map using coarse-grained locking via a single PO. Our hash-map provides an insert operation and a lookup function. We employed open addressing for collision resolution. Our hash-map's keys are of type 32 bit unsigned integer, and the value type is `Character`. We purposefully used a value type of small size, to prevent the experiment from being perturbed by costly data movement operations which could arise, e.g., with strings. The purpose of the experiment was to determine the performance improvement possible with lock elision.

We processed insertion and lookup with random keys generated by the random number generator of package `Ada.Numerics.Discrete_Random`. Those op-

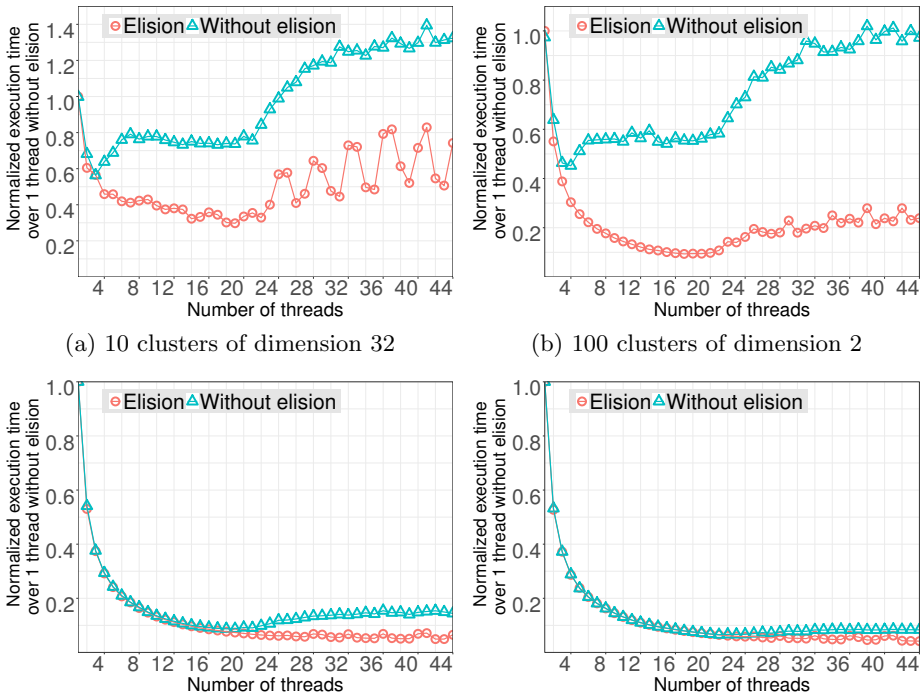
erations were performed in a tight loop. One run of the experiment entailed 50 million insert operations on an empty hash-map, and 50 million lookup operations on a pre-filled hash-map. Our hash-map was of size $2 \times X$. Note that in this experiment, the overall number of operations (50 million) was independent of the number of participating tasks. As more tasks are involved, the number of operations per task decrease.

Fig. 5a and Fig. 5b depict the execution times for the insertion- and lookup-operations. Clearly the global lock (line-graph "Without Elision") does not scale to multiple tasks. Because of lock contention, tasks serialize at the PO. The result with the elided PO (line-graph "Elision") shows superior scalability and performance. It should be noted that tasks ran these operations in a tight loop, which is an unrealistic scenario for real-world applications.

Fig. 5c and Fig. 5d illustrate the success rate of transaction cycles. As more tasks are accessing the PO, the probability of data conflicts during insertion operations increases. Lookup operations do not write shared data in the hash-map hence a data conflict with other tasks is unlikely.

**K-means Clustering.** The K-means algorithm groups objects located in an $N$-dimensional space into $K$ clusters. This algorithm is commonly used for data-mining. The STAMP [21,9] version of the K-means algorithm partitions objects and employs threads such that each thread processes a subset of objects iteratively. With STAMP, a transaction is used to protect the update of the cluster centers, which occurs during each iteration. The algorithm spends most of the time computing cluster centers, and data conflicts resulting from cluster center updates are rare. The algorithm thus benefits from the optimistic concurrency of transactional memory.

To evaluate the effects of lock elision with K-means clustering in Ada, we ported the C-implementation from STAMP to Ada. Instead of using transactional memory, we protected the cluster center updates by a PO. Compared to the previous examples, the percentage of CPU-cycles spent inside of the critical sections is very small (restricted to the cluster center updates), which results in smaller impact from lock elision. Fig. 6 depicts the performance difference for elided and non-elided POs for the K-means clustering benchmark. A constant number of points was clustered for varying numbers of clusters. A larger number of clusters reduces the potential for data conflicts during the tasks' joint update of the cluster centers. Higher-dimensional data points incur more parallelizable work and hence diminish the the non-parallelizable part (i.e., synchronization) relative to the total amount of work. With ten clusters (Fig. 6a), scalability cannot be maintained past 18 tasks, because of the high probability of data conflicts. On the other hand, with 100 clusters of the same dimension (Fig. 6c), performance does not degrade. This is due to the fact that with 100 cluster centers there is a lower probability of conflicts than with 10 cluster centers. For data points of higher dimension (Fig. 6d), the benefit from lock elision diminishes, because the computation-to-communication ratio increases. Fig. 6b clearly shows




(a) 10 clusters of dimension 32

(b) 100 clusters of dimension 2

(c) 100 clusters of dimension 32

(d) 100 clusters of dimension 64

Fig. 6: Execution time of Ada K-means clustering (65536 points)

the benefits of lock elision for a small computation-to-communication ratio. Lock elision can decrease the execution time by more than a factor of 5 in this case.

## 4 Related Work

In [18], a comprehensive introduction to lock elision with Intel TSX is provided. The same author implemented lock elision for the glibc library [10]. There, POSIX mutexes are incorporated with Intel TSX. Lock elision is enabled on TSX-capable systems via the "`--enable-lock-elision=yes`" parameter stated at compile time. While lock elision promises to provide the scalability of non-blocking synchronization, its actual performance will vary regarding the frequency and cost of data conflicts. The author proposes an "adaptive elision" policy which is currently used in glibc and subject for improvement. In the author's description of the policy, transactional aborts and unsuccessful transactional execution will adaptively enable elision skips for a given period of time. The author opens possibilities for improving the heuristics in future versions of the library. It will improve C programs which are built against glibc. However, such volatility can introduce an unwanted threat to Ada run-time systems. The

Ada programming language strictly specifies the behavior of its protected objects due to the language's provision for reliability and safety. Glibc's lock elision support depends on the capabilities of the underlying hardware and the adaptive heuristic elision algorithm which is not directly accessible to Ada programmers who utilize POs. Our Linked Lists counter example supports the necessity of a means to assess the practicality of lock elision for each use-case. A more effective solution is to provide fine-grained lock elision policies for Ada POs tailored in-line with the design principles of the Ada programming language.

In [28], Yoo et al. apply Intel TSX to a set of benchmarks in the high-performance computing domain (HPC). They survey a broad spectrum of workloads, including a parallel, user-level TCP/IP stack. Their benchmarks are all implemented in C/C++ and do not contain monitor constructs like Ada's POs. Their evaluation is restricted to a 4th generation Intel Core processor with 4 cores (2 hyperthreads per core). In contrast, we investigated lock elision with Ada POs. Our evaluation platform is a state-of-the-art 2 CPU 44 cores Xeon E5-2699 v4 system. Our experiments show scalability up to 44 cores in almost all cases.

In previous work on Ada's protected objects in real-time systems [12], the term *transaction* is employed with single atomic primitives (e.g., read-modify-write operations) in conjunction with lock-free programming. In contrast, our approach allows to combine multiple statements of a protected procedure into a transaction, with TSX as the underlying hardware mechanism. The work in [12] imposes several restrictions to achieve *transactional* behavior, such as disallowing the use of multiple memory locations and loop statements. These restrictions are due to the use of hardware atomic primitives when hardware TM was not available.

Lock-free data-structures provide concurrent access to shared data without relying on locks to achieve mutual exclusion [20,15]. They rely on hardware primitives such as a compare-and-swap (CAS) instruction and it is an agreed-upon fact that the design of lock-free algorithms is complex. Compared to lock elision, lock-free programming shifts the burden of achieving fine-grained parallelism to the programmer. Nevertheless, this approach can achieve good scalability. Both Simple Components [8] and Non-Blocking Ada [6] are collections of lock-free data-structures implemented in Ada. As of June 2016, the Ada Conformity Assessment Authority (ACAA) has been concerned with the provision of concurrent access to Ada's container libraries [1].

## 5 Conclusions

We have implemented hardware lock elision for protected functions and procedures in Ada 2012. For entries, we presented two possible schemes for lock elision, with varying degrees of parallelism. We demonstrated that lock elision can achieve significant performance improvements. We showed the scalability of our approach for several benchmarks up to 44 cores of a two-CPU, 44-core Intel E5-2699 v4 system. To the best of our knowledge, this is the first approach to

lock elision for monitor constructs. Our benchmark source code and the GNARL implementation have been made available on GitHub [4].

**Acknowledgements.** Research supported 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 NRF2015M3C4A7065522.

## References

1. ACAA Web site on "Concurrent access to Ada container libraries". http://www.ada-auth.org/cgi-bin/cvsweb.cgi/ai12s/ai12-0196-1.txt?rev=1.4, accessed: 2017-01-20
2. Intel Developer Zone: Pause Intrinsic. https://software.intel.com/en-us/node/524249, accessed: 2017-01-12
3. LIKWID GitHub page. https://github.com/RRZE-HPC/likwid/wiki, accessed: 2017-01-10
4. Lock-elided protected object resources on GitHub. https://github.com/bbur/pobj-tsx.git, created: 2017-03-28
5. Lock elision anti-patterns. https://software.intel.com/en-us/articles/tsx-anti-patterns-in-lock-elision-code, accessed: 2017-03-20
6. NBAda: Non-blocking data structures for Ada Web site. http://www.gidenstam.org/ada/Non-Blocking/, accessed: 2017-01-20
7. Performance application programming interface (PAPI) Web site. http://icl.cs.utk.edu/papi/, accessed: 2017-01-10
8. Simple Components Web site. http://www.dmitry-kazakov.de/ada/components.htm, accessed: 2017-01-20
9. STAMP GitHub page. https://github.com/kozyraki/stamp, accessed: 2017-01-20
10. The GNU C Library is now available. https://lists.gnu.org/archive/html/info-gnu/2013-08/msg00003.html, accessed: 2017-01-22
11. The world's simplest lock-free hash table, Preshing on programming blog. http://preshing.com/20130605/the-worlds-simplest-lock-free-hash-table/, accessed: 2017-01-22
12. Bosch, G.: Lock-free protected types for real-time Ada. Ada Lett. 33(2), 66–74 (Nov 2013)
13. Dice, D., Lev, Y., Moir, M., Nussbaum, D.: 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. pp. 157–168. ASPLOS XIV, ACM, New York, NY, USA (2009)
14. Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. In: Proceedings of the 20th Annual International Symposium on Computer Architecture. pp. 289–300. ISCA '93, ACM, New York, NY, USA (1993)
15. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2008)
16. Hoare, C.A.R.: Monitors: An operating system structuring concept. Commun. ACM 17(10), 549–557 (Oct 1974)

17. Intel Corporation: Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1 (Dec 2016)
18. Kleen, A.: Scaling existing lock-based applications with lock elision. Queue 12(1), 20:20–20:27 (Jan 2014)
19. McCool, M., Reinders, J., Robison, A.: Structured Parallel Programming: Patterns for Efficient Computation. Morgan Kaufmann Publishers Inc., 1st edn. (2012)
20. Michael, M.M.: The balancing act of choosing nonblocking features. Commun. ACM 56(9), 46–53 (Sep 2013)
21. Minh, C.C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford transactional applications for multi-processing. In: 4th International Symposium on Workload Characterization (IISWC 2008), Seattle, Washington, USA, September 14-16, 2008. pp. 35–46 (2008)
22. Rajwar, R., Goodman, J.R.: Speculative lock elision: Enabling highly concurrent multithreaded execution. In: Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture. pp. 294–305. MICRO 34, IEEE Computer Society (2001)
23. Scott, M.L.: Shared-Memory Synchronization. Synthesis Lectures on Computer Architecture, Morgan & Claypool Publishers (2013)
24. Taft, S.T., Duff, R.A., Brukardt, R., Plödereder, E., Leroy, P., Schonberg, E.: Ada 2012 Reference Manual. Language and Standard Libraries - International Standard ISO/IEC 8652/2012 (E), Lecture Notes in Computer Science, vol. 8339. Springer (2013)
25. Terpstra, D., Jagode, H., You, H., Dongarra, J.: Collecting performance data with PAPI-C. In: Müller, M.S., Resch, M.M., Schulz, A., Nagel, W.E. (eds.) Parallel Tools Workshop. pp. 157–173. Springer (2009)
26. Treibig, J., Hager, G., Wellein, G.: 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)
27. Wang, A., Gaudet, M., Wu, P., Amaral, J.N., Ohmacht, M., Barton, C., Silvera, R., Michael, M.: Evaluation of blue gene/q hardware support for transactional memories. In: Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques. pp. 127–136. PACT '12, ACM, New York, NY, USA (2012)
28. Yoo, R.M., Hughes, C.J., Lai, K., Rajwar, R.: 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. pp. 19:1–19:11. SC '13, ACM, New York, NY, USA (2013)