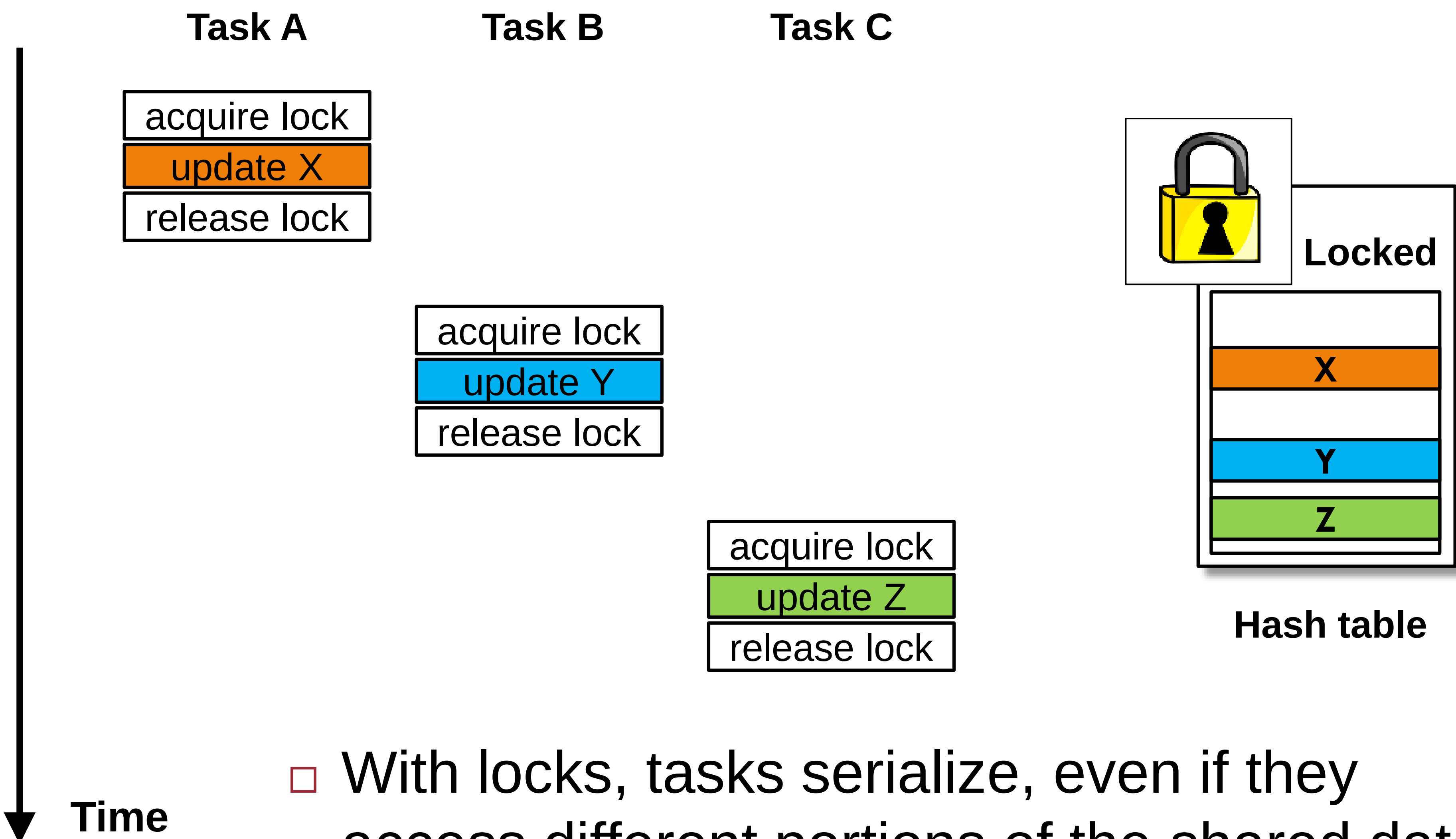
Lock Elision For Protected Objects Using Intel Transactional Synchronization Extensions

Seongho Jeong, Shinhyung Yang and Bernd Burgstaller
Department of Computer Science
Yonsei University, Korea

Motivation

- Locks are commonly used to protect shared data from data races
- A coarse-grained lock protects a large amount of shared data
  - Advantage: easy to implement, hardly any bugs
  - Disadvantage: scalability bottleneck
- Ada protected objects (POs) are a monitor construct for mutual exclusion
  - Same scalability problem if used for coarse-grained locking
- Example: concurrent hash table
  - Protect the entire hash table by a coarse-grained lock
  - → tasks serialize even if accessing different keys ☹

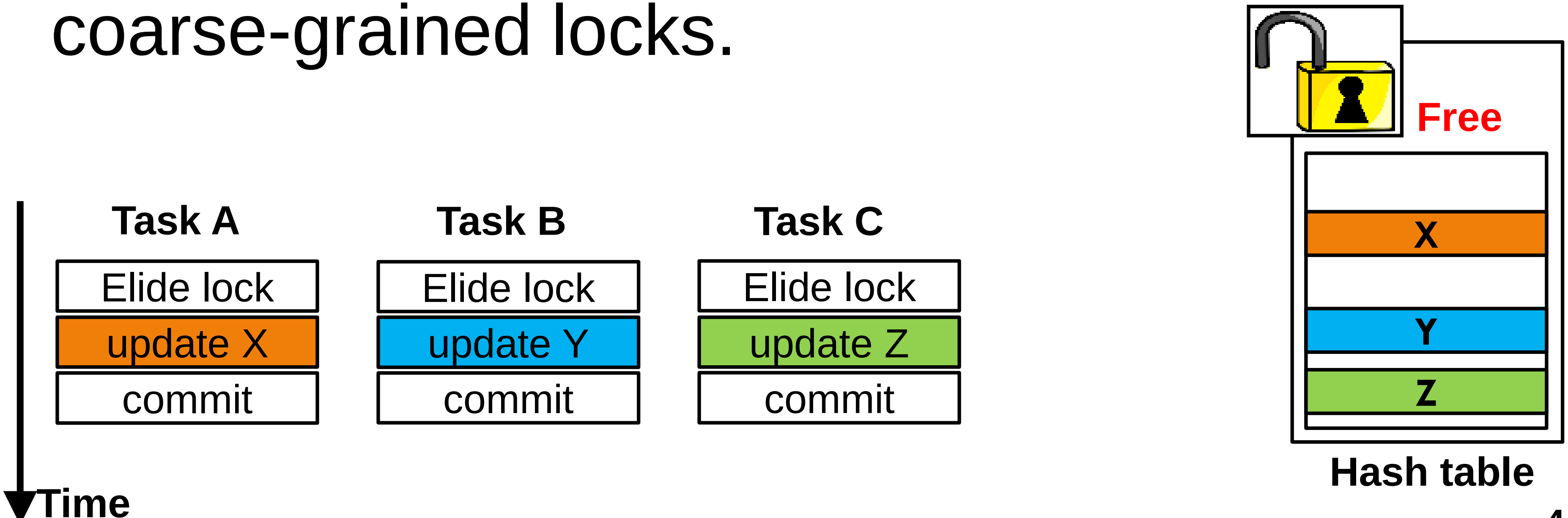
Motivation (cont.)



- With locks, tasks serialize, even if they access different portions of the shared data.

Motivation (cont.)

- **Observation**: not all tasks using a PO access the same part of the shared data
  - Fine-grained locking of individual data-items can be a fix (but an error-prone one).
- **Goal**: provide fine-grained parallelism for coarse-grained locks.



Our contributions

1. Adapt GNU Ada run-time library (GNARL) to elide locks from protected functions and procedures.
2. Investigate opportunities and difficulties with lock elision of protected entries.
3. Evaluate the approach for multiple benchmarks in terms of scalability.
4. Provide programming- and language-design directions for more parallelism obtainable from lock elision with POs in Ada.

Transactions enable parallelism

- Transaction
  - Indivisible process
  - Composed by multiple operations inside transactional region
  - Accesses multiple memory locations atomically
- Speculative execution
  - Tentative and invisible to other tasks
  - Either commits or aborts
  - Keeps read-set and write-set of a transaction
- Possible to run in parallel because changes are tentative
  - Transaction **commits** in the absence of data-conflict.
  - **Start over** in case of transaction abort.

Data conflict in transaction

Assume task T is inside a transaction:

- A **data conflict** occurs **iff** another task
  - reads a location in task T's write-set, **or**
  - writes a location in task T's read-set or write-set.

- Transaction aborts if data conflict is detected.
  - Changes in write-set will be discarded.

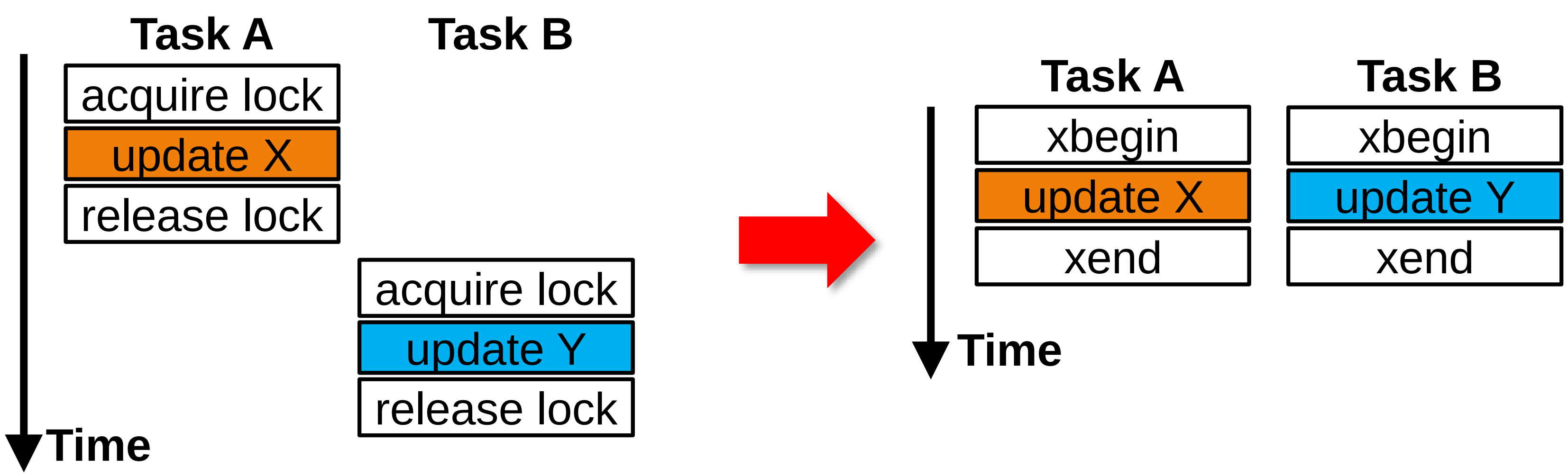
Intel TSX

- *`xbegin`*, *`xend`* to delimit transactional region
- Transaction aborts if
  - a data conflict occurs
    - In units of cache-lines
  - a task exceeds the read/write set capacity limit,
  - an illegal instruction is executed
    - interrupt, system call, ..., **or**
  - an explicit abort by software is called (*`xabort`* instruction).
- Abort state is reported in EAX register.
- Fall-back path is always required.
  - Transaction can fail endlessly.
  - Need to ensure progress by falling back on conventional lock.

```
    xbegin L1
    <Transactional operations>
    xend   ;commit changes
    <Operations after commit>
    ...
L1: <Operations on abort>
```

Lock elision with Intel TSX

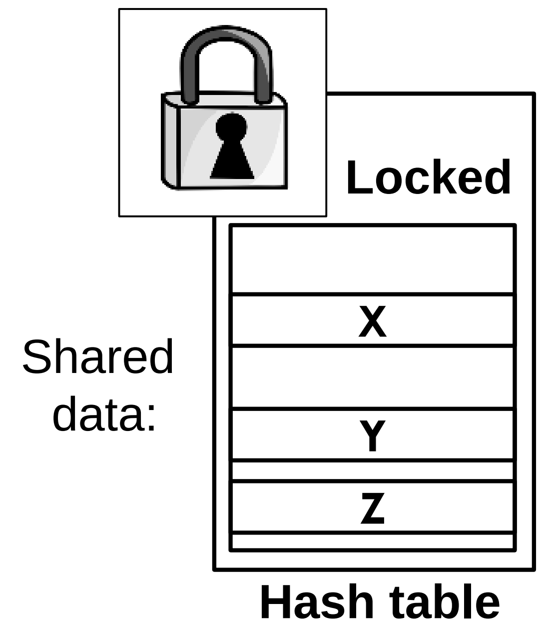
- Run critical section inside transaction.
  - Elide lock acquisition to enter critical section.
  - Commit upon exit of critical section.
  - Multiple tasks in critical section at the same time.
    - As long as tasks access different **cache lines.**



Protected objects in GNAT

- GNARL wraps protected function and procedure calls by a lock/unlock pair to achieve mutual exclusion:

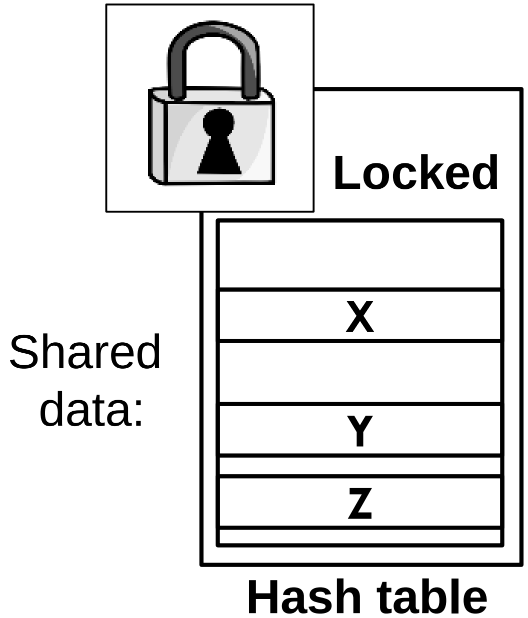
```
protected type HashTable is
  procedure Insert (key, val);
private
  Slots : ...; -- Shared data
end HashTable;
```



Protected objects in GNAT

- GNARL wraps protected function and procedure calls by a lock/unlock pair to achieve mutual exclusion:

```
protected type HashTable is
  procedure Insert (key, val);
private
  Slots : ...; -- Shared data
end HashTable;
```

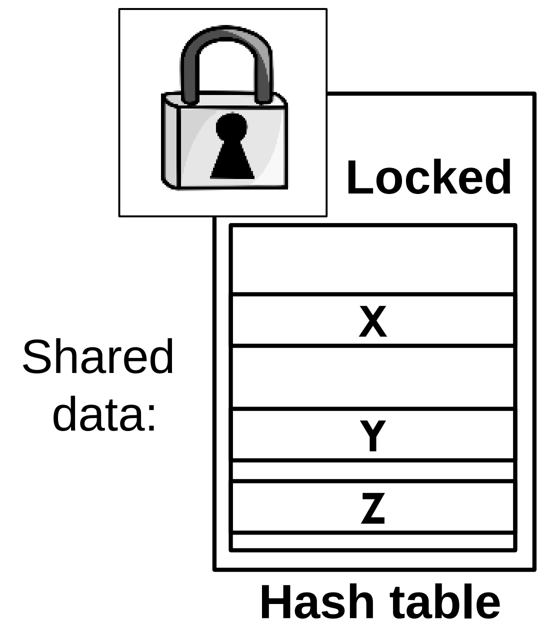


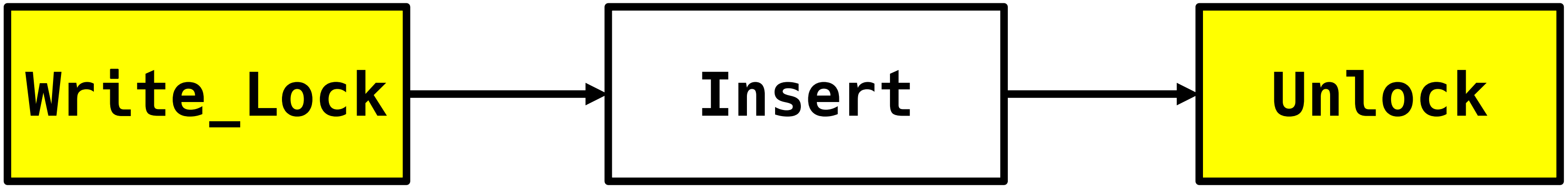
Insert

Protected objects in GNAT

- GNARL wraps protected function and procedure calls by a lock/unlock pair to achieve mutual exclusion:

```
protected type HashTable is
   procedure Insert (key, val);
private
   Slots : ...; -- Shared data
end HashTable;
```





GNARL lock elision

```
procedure Write_Lock
  result := Try_Elision
  if result = fail then
    acquire PO.lock
  end if
  return
end Write_Lock
```

Adapted Write_Lock:

- Attempt lock elision
  - Call Try_Elision
- Else acquire lock
  - Fall-back path
  - to prevent infinite abort

```
Max_Retry : constant Natural...

procedure Try_Elision
  retry := 0
  while retry < Max_Retry loop
    state := xbegin
    if state = STARTED then
      if PO.lock = free then
        return success
      else
        xabort
      end if
    else if state = CAPACITY or
            state /= RETRY then
      return fail
    else
      Backoff Exponentially
      wait until PO.lock = free
      retry := retry + 1
    end if
  end loop
  return fail
end Try_Elision
```

Successful elision

```
procedure Write_Lock
   result := Try_Elision
   if result = fail then
      acquire PO.lock
   end if
   return
end Write_Lock
```

- Call Try_Elision before acquiring lock
  - Start transaction via **`xbegin`**

```
Max_Retry : constant Natural...

procedure Try_Elision
   retry := 0
   while retry < Max_Retry loop
      state := xbegin
      if state = STARTED then
         if PO.lock = free then
            return success
         else
            xabort
         end if
      else if state = CAPACITY or
               state /= RETRY then
         return fail
      else
         Backoff Exponentially
         wait until PO.lock = free
         retry := retry + 1
      end if
   end loop
   return fail
end Try_Elision
```

Successful elision

```
procedure Write_Lock
  result := Try_Elision
  if result = fail then
    acquire PO.lock
  end if
  return
end Write_Lock
```

```
Max_Retry : constant Natural...

procedure Try_Elision
  retry := 0
  while retry < Max_Retry loop
    state := xbegin
    if state = STARTED then
      if PO.lock = free then
        return success
      else
        xabort
      end if
    else if state = CAPACITY or
            state /= RETRY then
      return fail
    else
      Backoff Exponentially
      wait until PO.lock = free
      retry := retry + 1
    end if
  end loop
  return fail
end Try_Elision
```

- Call Try_Elision before acquiring lock
  - Start transaction via **`xbegin`**
- On success:
  - Return and proceed in transactional mode

Transaction aborts

- If a transaction aborts, the CPU transfers control back to statement **xbegin**
  - `State = STARTED` is false at this moment
- Two causes for transaction abort:
  - A. Non-retryable
    - **Capacity overflow**
    - **Illegal instructions**
  - B. Retryable
    - **Data conflict**
    - **Software-induced abort**

Transfer here on transaction abort

```
Max_Retry :

procedure T
   retry := 
   while retry <   Max_Retry loop
      state := xbegin
      if state = STARTED then
         if PO.lock = free then
            return success
         else
            xabort
         end if
   A  else if state = CAPACITY or
               state /= RETRY then
         return fail
   B  else
         Backoff Exponentially
         wait until PO.lock = free
         retry := retry + 1
      end if
   end loop
   return fail
end Try_Elision
```

Non-retryable aborts

```
procedure Write_Lock
  result := Try_Elision
  if result = fail then
    acquire PO.lock
  end if
  return
end Write_Lock
```

- On abort due to capacity:
  1. Asynchronous transfer of control to **`xbegin`**.
  2. CPU reports the cause of abort (**`state`**).
  3. Do not retry in hopeless situations.
  4. Fall back to conventional lock acquisition.

Transfer here on transaction abort

```
Max_Retry :

procedure T
  retry := 
  while retry < Max_Retry loop
    state := xbegin
    if state = STARTED then
      if PO.lock = free then
        return success
      else
        xabort
      end if
    else if state = CAPACITY or
            state /= RETRY then
      return fail
    else
      Backoff Exponentially
      wait until PO.lock = free
      retry := retry + 1
    end if
  end loop
  return fail
end Try_Elision
```

Retryable aborts

```
procedure Write_Lock
  result := Try_Elision
  if result = fail then
    acquire PO.lock
  end if
  return
end Write_Lock
```

Retry may succeed if:

A. A lock is held by a competing task dwelling on the fall-back path
   - Once task detects this, it will abort explicitly (**`xabort`**)

B. A data conflict occurred.

```
Max_Retry : constant Natural ...

procedure Try_Elision
  retry := 0
  while retry < Max_Retry loop
    state := xbegin
    if state = STARTED then
      if PO.lock = free then
        return success
      else
        xabort
      end if
    else if state = CAPACITY or
            state /= RETRY then
      return fail
    else
      Backoff Exponentially
      wait until PO.lock = free
      retry := retry + 1
    end if
  end loop
  return fail
end Try_Elision
```

A

B Data Conflict

How to retry?

```
procedure Write_Lock
  result := Try_Elision
  if result = fail then
    acquire PO.lock
  end if
  return
end Write_Lock
```

- On explicit abort or abort because of data conflict:
  - 1–4 Confirm retry might succeed
  - 5 Back off and wait until lock is free.
  - 6 Proceed to retry.

```
Max_Retry :

procedure T
  retry := 0
  while retry < Max_Retry loop
    state := xbegin
    if state = STARTED then
      if PO.lock = free then
        return success
      else
        xabort
      end if
    else if state = CAPACITY or
            state /= RETRY then
      return fail
    else
      Backoff Exponentially
      wait until PO.lock = free
      retry := retry + 1
    end if
  end loop
  return fail
end Try_Elision
```

Transfer here on transaction abort

Giving up on elision

```
procedure Write_Lock
   result := Try_Elision
   if result = fail then
      acquire PO.lock
   end if
   return
end Write_Lock
```

- Retry with backoff:
  - 1–6 Try Max_Retry times
  - 7 Report failure
  - 8 Fall back to conventional lock acquisition

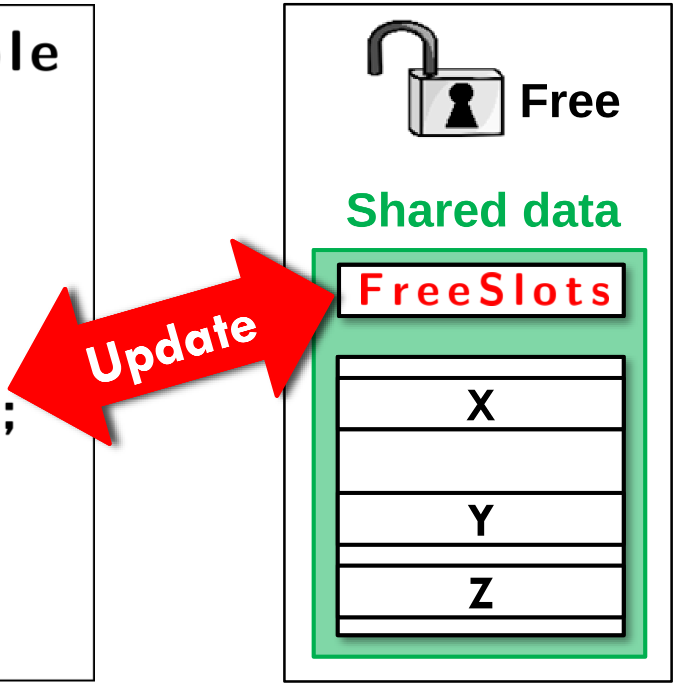
```
Max_Retry : constant Natural...

procedure Try_Elision
   retry := 0
   while retry < Max_Retry loop
      state := xbegin
      if state = STARTED then
         if PO.lock = free then
            return success
         else
            xabort
         end if
      else if state = CAPACITY or
              state /= RETRY then
         return fail
      else
         Backoff Exponentially
         wait until PO.lock = free
         retry := retry + 1
      end if
   end loop
   return fail
end Try_Elision
```

Lock elision for protected entries

- Variables occurring in entry_barrier constitute **shared data**.
  - POs update such **state variables** inside protected operations.
  - **Examples**: growable hash table, queue, semaphore, ...
- State variable updates drastically increase data conflicts.
  - Annihilate performance gain from elision. (Observed ~50% abort rate)

```
protected body GrowableHashTable
is
  entry Insert (key, val)
    when FreeSlots > 0 is
  begin
    FreeSlots := FreeSlots − 1;
    −− Update shared data...
  end Insert;
end GrowableHashTable;
```

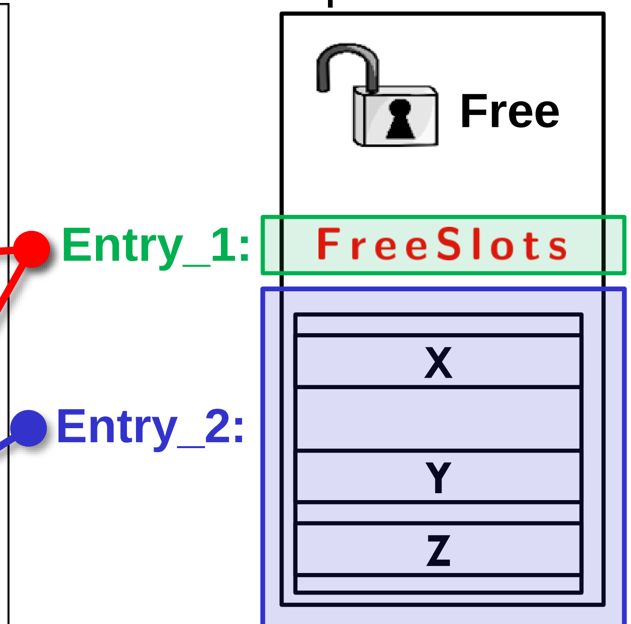


Manual code transformations

- To leverage parallelism, programmer may transform entry into two halves:
  1) Entry_1 (original barrier, not elided): update state variables, requeue on Entry_2.
  2) Entry_2 (**true** barrier, elided): update remaining shared data.
- Limitations:
  a) Not applicable if state variables need to be updated at end of entry-code.
  b) Only profitable if parallelizing Entry_2 is profitable.
  c) Manual code-transformation may introduce concurrency bugs.

❖ Observed up to 5x speedup over non-split entry from this example.

```
protected body GrowableHashTable
is
   entry Insert (key, val)
     when FreeSlots > 0 is
   begin
     FreeSlots := FreeSlots - 1;
     -- Update shared data...
   end Insert;
end GrowableHashTable;
```



Restrictions of the Eggshell model

- Ada 2012 RM Chapter 9.5.3(16):
  - Queued entry calls with an open barrier **must** precede all other protected operations (eggshell model).
  - The RM does not state the reason for Clause 9.5.3(16), but probably to avoid starvation.
  - Clause 9.5.3(16) restricts the parallelism obtainable from lock elision.

We considered two possible implementation scenarios:

1. Permissive lock elision
   - Waive Clause 9.5.3(16) by PO type annotation.
   - Reason: for many parallel workloads, starvation is not an issue, throughput is.
2. Restrictive lock elision
   - Switch the PO's mode from elided to non-elided when an entry call is enqueued at a closed barrier.
   - Switch back to elided mode after all queued entries have been processed.

Experimental evaluation

- Three synthetic and one real-world benchmarks
  1) Linked lists ☹
     - A counter-example due to capacity aborts from traversal.
  2) Dijkstra's Dining Philosophers ☺
  3) Concurrent hash table ☺
  4) K-means clustering ☺
     - From Stanford STAMP suite
  - Employed protected procedures & functions only.
- Evaluation platform:
  - 44 cores (2 CPU Intel Xeon E5-2699 v4 system)
  - Linux kernel version 4.9.4, GCC/GNAT 6.3.0

Measurement set-up

- All synthetic benchmarks ran in a tight loop
  - Example: Dining Philosophers

```
for i in 1 .. MaxIterations loop
   acquireFork_1;
   acquireFork_2;
      null;    -- eat
   releaseFork_1;
   releaseFork_2;
end loop;
```

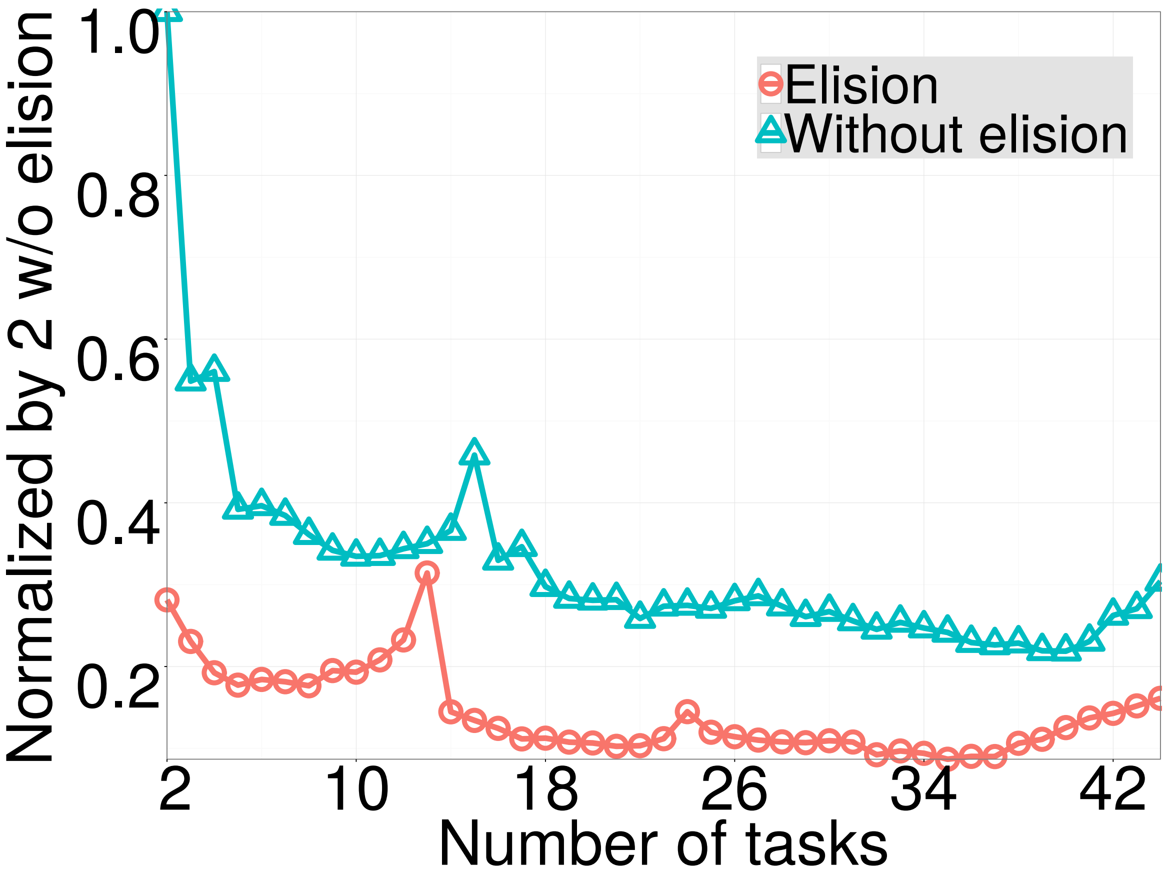
- Computation-to-communication ratio thereby minimized
  - Simulates a highly-contended PO
  - An upper bound for the best-possible performance improvement (Amdahl’s Law)

Performance tuning

- Max_Retry
  - **Empirically**, Max_Retry should be higher than the number of participating tasks
  - Max_Retry = 200 for all our experiments.

- Padding & alignment to prevent **false sharing**
  - Reduces possibility of data conflict
  - Data structure layout may need to be revised.

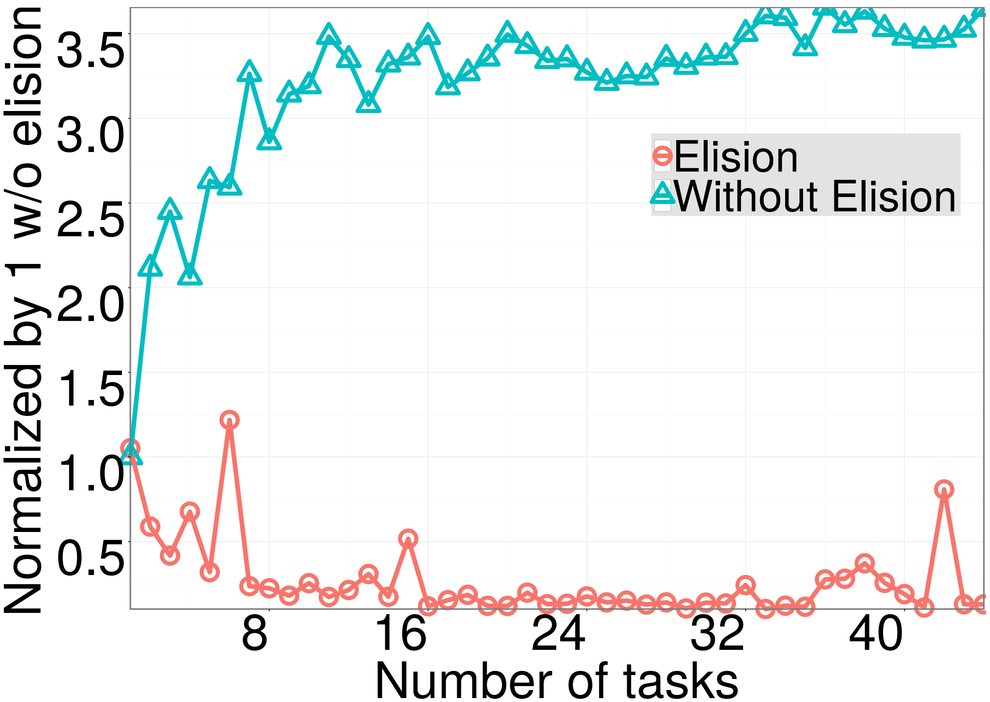
Dining philosophers

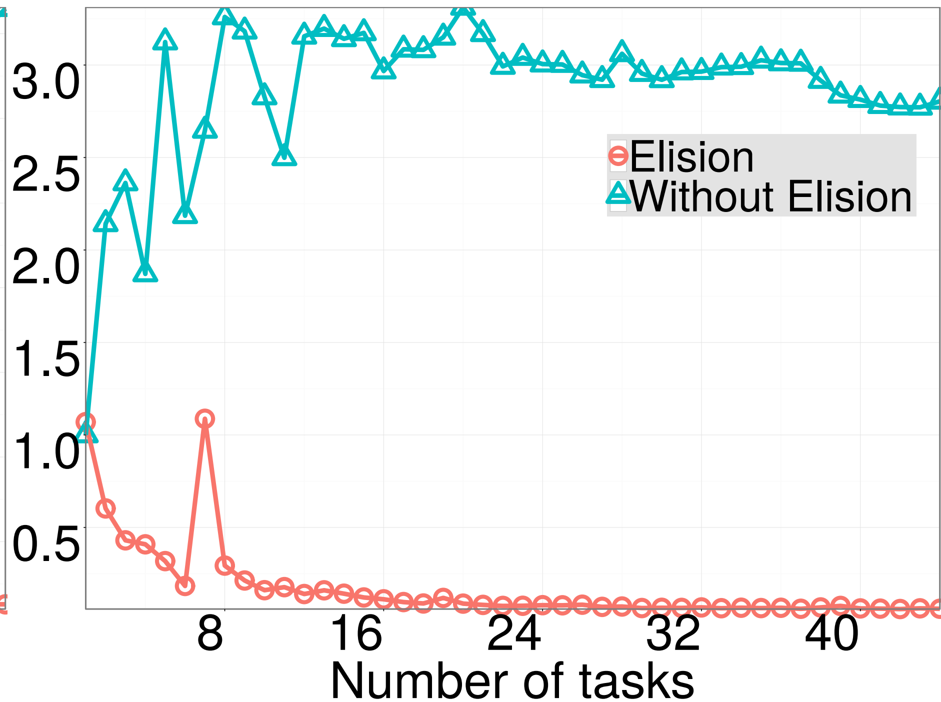


Normalized execution time

- Each fork as a PO
- 1 million meals per philosopher
  - Acquire & release forks repeatedly
- Show benefit from omitting lock acquisition

Concurrent hash table

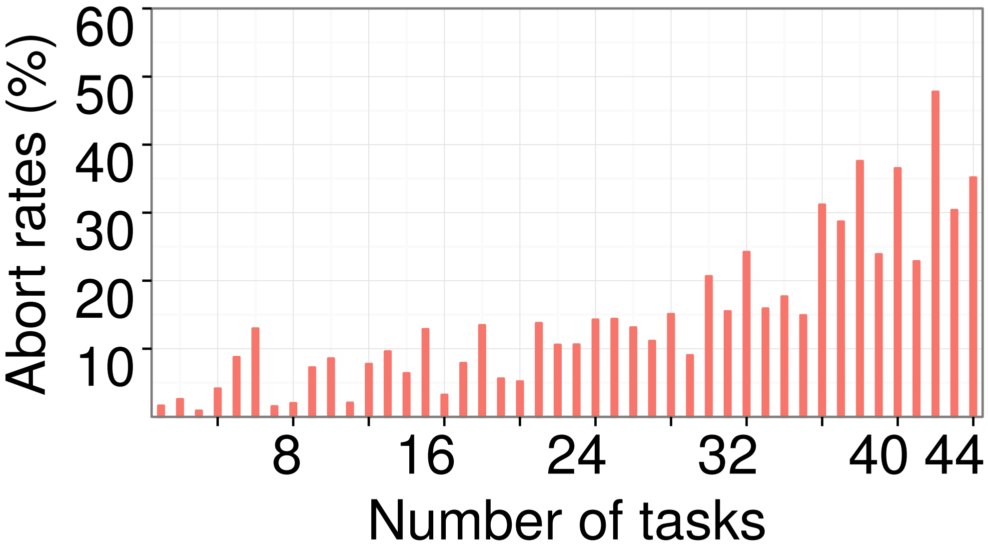




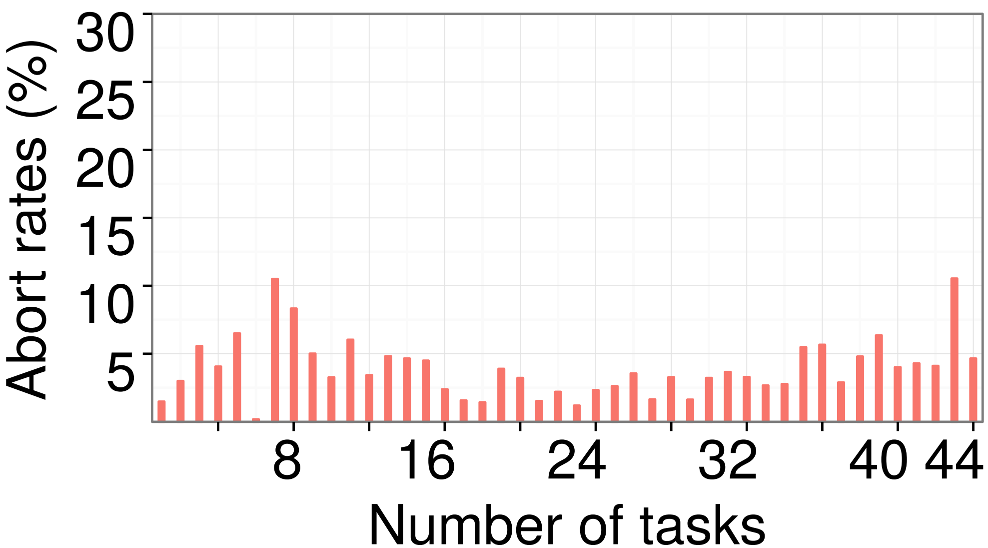
Normalized time for insert **Normalized time for lookup**

- Random key generation for operations
- 50 million operations in a tight loop

Concurrent hash table (cont.)

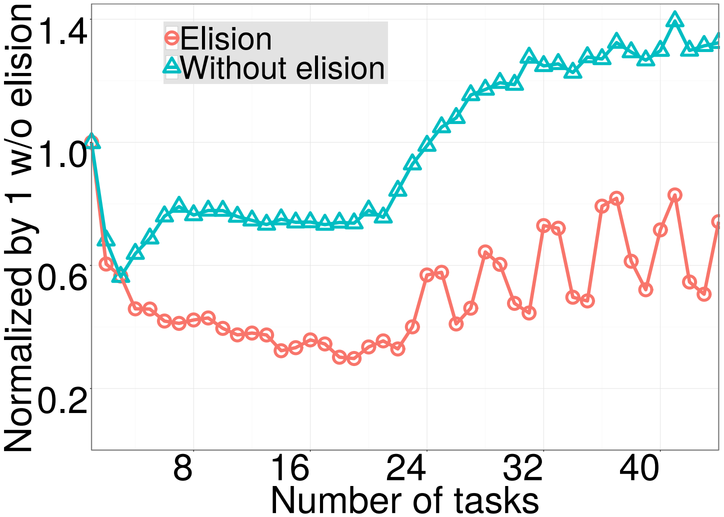


Abort rates (%) for insert

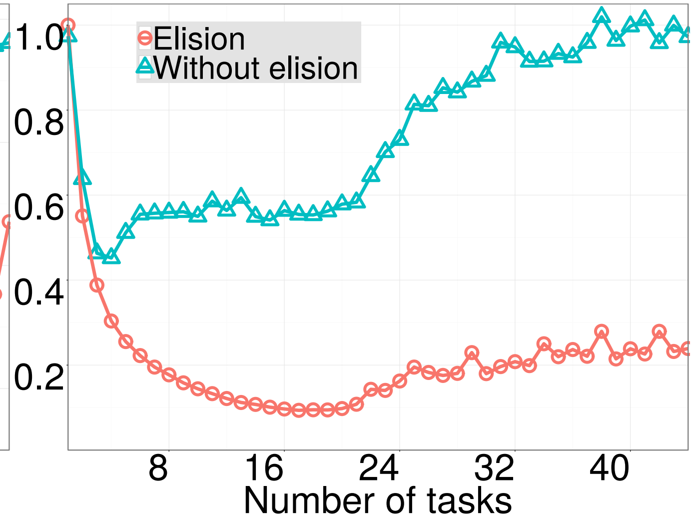


Abort rates (%) for lookup

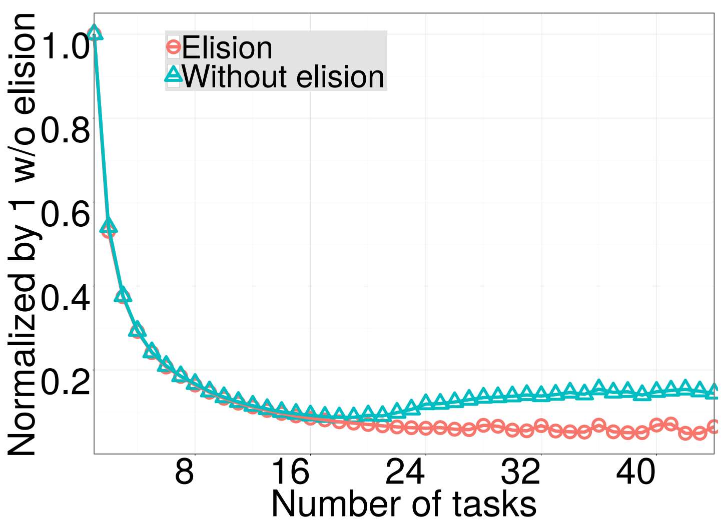
K-means clustering



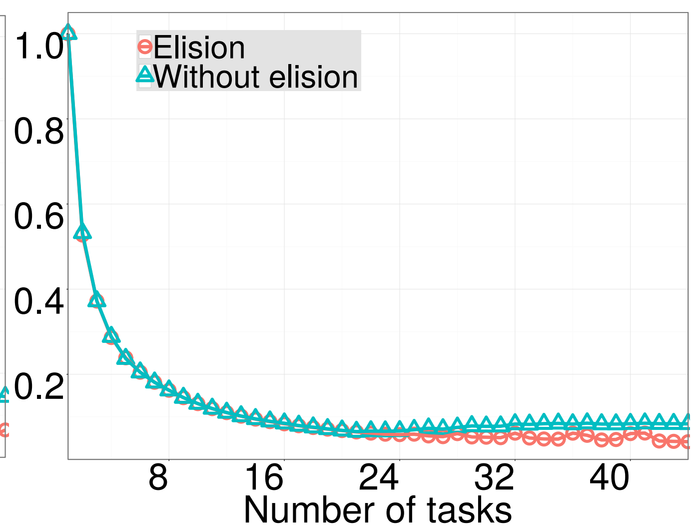
10 clusters, 32 dimensions



100 clusters, 2 dimensions



100 clusters, 32 dimensions



100 clusters, 64 dimensions

- Ported from STAMP benchmark suite
- Cluster centers as a PO
- More clusters
  → fewer data conflicts
- Higher dimension
  → less benefit
  from lock elision

Conclusion

- Implemented lock elision for protected functions and procedures in the Ada 2012 GNARL.
- Presented possible schemes for lock elision with entries.
- Demonstrated that lock elision can improve performance significantly.
  - Not all types of POs benefit from speculative execution.
  - Programmer intervention may be required to selectively enable elision for certain POs.
- Provided tuning methods to optimize performance.
- Experimental results showed the scalability of lock elision for several benchmarks on up to 44 cores.

Q&A

Thank you!