Pragmatic Primitives for Non-blocking Data Structures

PODC 2013

Trevor Brown, University of Toronto
Faith Ellen, University of Toronto
Eric Ruppert, York University

June 27, 2013

Goal: non-blocking data structures

Data structures that can be accessed concurrently by many processes are

- important
- hard to design
- hard to prove correct

We focus on linearizable, non-blocking data structures.

Software transactional memory

Transactional memory

Enclose each data structure operation in an atomic transaction.

Pros:

- simple to design
- simple to prove correct

Cons:

- less efficient than hand-crafted data structures
- coarse-grained transactions limit concurrency

Right solution for “casual” data structure designers.

Direct implementations

Handcrafted non-blocking implementations from hardware primitives.

Pros:

- allows good efficiency
- allows high degree of concurrency

Cons:

- hard to get implementation (provably) right

Right solution for designing libraries of data structures.

Why is it hard to use hardware primitives?

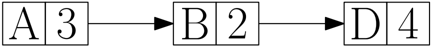
Key difficulty of implementing data structures from hardware primitives:

- Data structure operations access several words atomically
- Hardware primitives operate only on single words

Example: multiset

Multiset can be represented as a sorted, singly linked list with nodes storing keys and multiplicities.

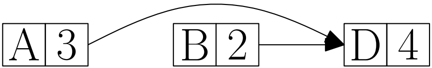
DELETE(B, 2):



Example: multiset

Multiset can be represented as a sorted, singly linked list with nodes storing keys and multiplicities.

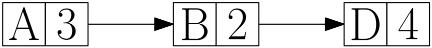
DELETE$(B, 2)$:



Example: multiset

How to add some copies of a key to the multiset.

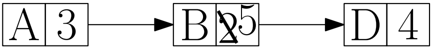
INSERT$(B, 3)$:



Example: multiset

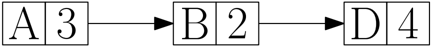
How to add some copies of a key to the multiset.

INSERT$(B, 3)$:



Example: multiset

Problems arise if we concurrently INSERT(B, 3) and DELETE(B, 2).



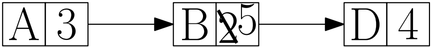
1. Each operation prepares to do its CAS.
2. INSERT occurs
3. DELETE occurs, three copies of B are lost.

DELETE should succeed only if node B is unchanged.

Need multi-word primitives.

Example: multiset

Problems arise if we concurrently INSERT(B, 3) and DELETE(B, 2).



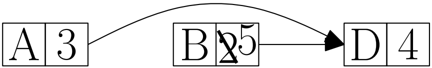
1. Each operation prepares to do its CAS.
2. INSERT occurs
3. DELETE occurs, three copies of B are lost.

DELETE should succeed only if node B is unchanged.

Need multi-word primitives.

Example: multiset

Problems arise if we concurrently INSERT$(B, 3)$ and DELETE$(B, 2)$.



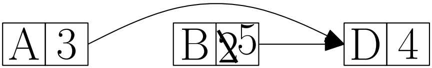
1. Each operation prepares to do its CAS.
2. INSERT occurs
3. DELETE occurs, three copies of B are lost.

DELETE should succeed only if node B is unchanged.

Need multi-word primitives.

Example: multiset

Problems arise if we concurrently INSERT(B, 3) and DELETE(B, 2).



1. Each operation prepares to do its CAS.
2. INSERT occurs
3. DELETE occurs, three copies of B are lost.

DELETE should succeed only if node B is unchanged.

Need multi-word primitives.

Our approach

Build "medium-level" primitives that can access multiple words.

- higher-level than CAS or LL/SC
- lower-level than full transactional memory

Advantages:

- General enough to be used in many data structures
- Specialized enough to create quite efficient implementations
- Modular proof of correctness: large parts can be reused

Data records

Our primitives work on data records.

Each data record has

- some mutable fields (one word each)
- some immutable fields

Use a data record for some natural "unit" of a data structure

- node in a tree
- entry in a table

Our primitives

Our primitives extend load-link (LL) and store-conditional (SC).

LL/SC object

- stores a single word
- LL reads value stored
- SC(v) (store-conditional) writes v only if value has not changed since last LL by process performing SC.

LLX and SCX

LLX(r) returns a snapshot of the mutable fields of r

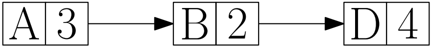
SCX(V, R, field, new) by process p

- writes value *new* into *field*,
  which is a mutable field of a data record in V
- finalizes all data records in $R \subseteq V$
- only if no record in V has changed since p's LLX on it

After a data record is finalized, no further changes allowed.

Example: removing all copies of a key in multiset

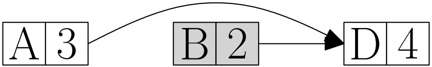
DELETE$(B, 2)$ using LLX and SCX.
Use one data record for each node.



1. LLX(A) $\rightarrow \langle A.count = 3, A.next = B \rangle$
2. LLX(B) $\rightarrow \langle B.count = 2, B.next = D \rangle$
3. SCX($\langle A, B \rangle, \langle B \rangle, A.next, D$)
   - changes $A.next$ to D
   - finalizes B
   - succeeds only if no changes since LLXs on A and B

Example: removing all copies of a key in multiset

DELETE$(B, 2)$ using LLX and SCX.
Use one data record for each node.



1. LLX(A)
   $\rightarrow \langle A.count = 3, A.next = B \rangle$
2. LLX(B)
   $\rightarrow \langle B.count = 2, B.next = D \rangle$
3. SCX$(\langle A, B \rangle, \langle B \rangle, A.next, D)$
   - changes $A.next$ to D
   - finalizes B
   - succeeds only if no changes since LLXs on A and B

Other multi-word primitives

Others have built medium-level multi-word primitives.

- Large LL/SC objects (Anderson Moir 1995, ...)
  $\Rightarrow$ unable to access multiple objects atomically
- Multi-word CAS (Israeli Rappoport 1994, ...)
  $\Rightarrow$ more general, less efficient
- Multi-word RMW (Afek Merritt Taubenfeld Touitou 1997, ...)
  $\Rightarrow$ even more general, less efficient
- k-compare-single-swap (Luchangco Moir Shavit 2009)
  $\Rightarrow$ lacks ability to finalize
  $\Rightarrow$ less efficient for some applications

Other multi-word primitives

Others have built medium-level multi-word primitives.

- Large LL/SC objects (Anderson Moir 1995, ...)
  $\Rightarrow$ unable to access multiple objects atomically
- Multi-word CAS (Israeli Rappopport 1994, ...)
  $\Rightarrow$ more general, less efficient
- Multi-word RMW (Afek Merritt Taubenfeld Touitou 1997, ...)
  $\Rightarrow$ even more general, less efficient
- *k*-compare-single-swap (Luchangco Moir Shavit 2009)
  $\Rightarrow$ lacks ability to finalize
  $\Rightarrow$ less efficient for some applications

Other multi-word primitives

Others have built medium-level multi-word primitives.

- Large LL/SC objects (Anderson Moir 1995, ...)
  $\Rightarrow$ unable to access multiple objects atomically
- Multi-word CAS (Israeli Rappoport 1994, ...)
  $\Rightarrow$ more general, less efficient
- Multi-word RMW (Afek Merritt Taubenfeld Touitou 1997, ...)
  $\Rightarrow$ even more general, less efficient
- *k*-compare-single-swap (Luchangco Moir Shavit 2009)
  $\Rightarrow$ lacks ability to finalize
  $\Rightarrow$ less efficient for some applications

Other multi-word primitives

Others have built medium-level multi-word primitives.

- Large LL/SC objects (Anderson Moir 1995, ...)
  $\Rightarrow$ unable to access multiple objects atomically
- Multi-word CAS (Israeli Rappoport 1994, ...)
  $\Rightarrow$ more general, less efficient
- Multi-word RMW (Afek Merritt Taubenfeld Touitou 1997, ...)
  $\Rightarrow$ even more general, less efficient
- k-compare-single-swap (Luchangco Moir Shavit 2009)
  $\Rightarrow$ lacks ability to finalize
  $\Rightarrow$ less efficient for some applications

More detailed specification: LLX

LLX(r) can return one of the following results.

- a snapshot of mutable fields of r
- FINALIZED (iff r has been finalized by an SCX)
- FAIL (in our implementation this happens only if a concurrent SCX accesses r)

We also allow reads of individual mutable fields.

More detailed specification: SCX

Before calling SCX(V, R, *field*, *new*), process p must get a snapshot from an LLX(r) on each record r in V.

For each r in V, the last LLX(r) by p is linked to the SCX.

If any r in V was changed since the linked LLX(r)
$\Rightarrow$ SCX returns FAIL.

Non-failed SCX sets *field* $\leftarrow$ *new* and finalizes records in R.

Spurious failures of SCX are allowed.

Progress properties of LLX and SCX

Individual LLXs and SCXs are wait-free, but may fail.

- If LLXs and SCXs are performed infinitely often, they succeed infinitely often.
- If SCXs are performed infinitely often, they succeed infinitely often.

Also, if no overlap between V-sets of SCX's, all will succeed.

Progress properties of LLX and SCX

Individual LLXs and SCXs are wait-free, but may fail.

- If LLXs and SCXs are performed infinitely often, they succeed infinitely often.
- If SCXs are performed infinitely often, they succeed infinitely often.

Also, if no overlap between V-sets of SCX's, all will succeed.

Progress properties of LLX and SCX

Individual LLXs and SCXs are wait-free, but may fail.

- If LLXs and SCXs are performed infinitely often, they succeed infinitely often.
- If SCXs are performed infinitely often, they succeed infinitely often.

Also, if no overlap between V-sets of SCX's, all will succeed.

Key idea of implementation

Lock-free Locks

- "Locks" on data records acquired by SCX operations
- If a record you need is locked by another SCX, you can help that SCX and release lock
- Finalized records remain permanently locked

Based on cooperative technique of Turek et al. [1992] and Barnes [1993]

SCX records

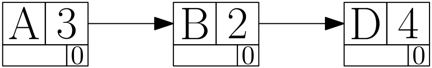
Each SCX creates an SCX record.

An SCX record contains all information needed to help SCX.

V	R	$field$	new	
expected values				(for CAS steps)
$state$			bit	

Add two fields to each data record *r*:

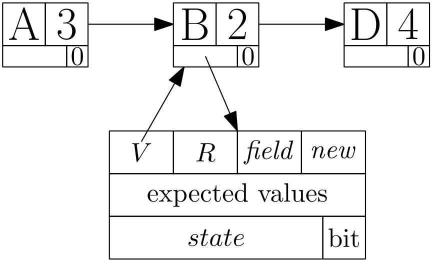
- *info*: pointer to SCX record of last SCX that locked *r*
- *marked*: boolean used to finalize *r*



Expected value for CAS comes from LLX.
$\Rightarrow$ CAS succeeds only if info field unchanged since LLX

Add two fields to each data record *r*:

- *info*: pointer to SCX record of last SCX that locked *r*
- *marked*: boolean used to finalize *r*

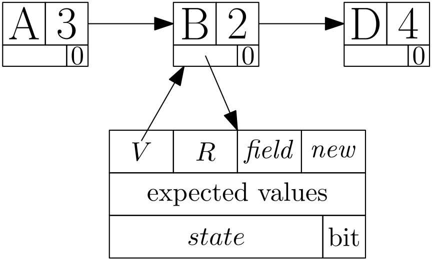


Expected value for CAS comes from LLX.

⇒ CAS succeeds only if info field unchanged since LLX

Add two fields to each data record *r*:

- *info*: pointer to SCX record of last SCX that locked *r*
- *marked*: boolean used to finalize *r*



Expected value for CAS comes from LLX.
⇒ CAS succeeds only if info field unchanged since LLX

Structure of SCX algorithm

SCX(V, R, *field*, *new*)
    create SCX record s
    for each r in $s.V$
        [lock r]
    CAS $s.new$ into $s.field$
    $s.state \leftarrow committed$
end SCX

Structure of SCX algorithm

~~SCX(V, R, *field*, *new*)~~ HELP(s)

~~create SCX record s~~

for each r in $s.V$

[lock r]

CAS $s.new$ into $s.field$

$s.state \leftarrow committed$

end ~~SCX~~ HELP

Help algorithm

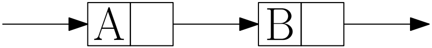
```
HELP(s)
    for each r in s.V
        try to CAS s into r.info
        if r.info ≠ s then
            if s.locksSucceeded then
                return TRUE % Someone else finished the operation
            else
                s.state ← aborted
                return FALSE
    s.locksSucceeded ← TRUE
    r.marked ← TRUE for each data record in R
    CAS s.new into s.field
    s.state ← committed
    return TRUE
end HELP
```

Key things to prove

- Locking correctly protects all mutable fields of a record
- All helpers of an SCX agree on outcome (failed/succeeded)
- No ABA problems on fields accessed by CAS
- Progress properties

Progress: livelock

Problems arise if different SCX operations lock records in different orders.

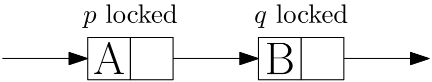


1. p locks A, B
   q locks B, A
2. Real locks: deadlock!
3. Lock-free locks: abort & retry, but repeat forever $\Rightarrow$ livelock!

Need SCXs to “lock” in consistent order
($\Rightarrow$ one will eventually succeed)

Progress: livelock

Problems arise if different SCX operations lock records in different orders.

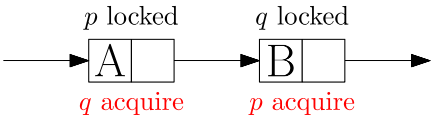


1. p locks A, B
   q locks B, A
2. Real locks: deadlock!
3. Lock-free locks: abort & retry, but repeat forever $\Rightarrow$ livelock!

Need SCXs to "lock" in consistent order
($\Rightarrow$ one will eventually succeed)

Progress: livelock

Problems arise if different SCX operations lock records in different orders.

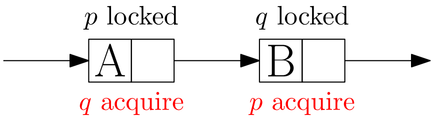


1. *p* locks *A*, *B*
   q locks *B*, *A*
2. Real locks: deadlock!
3. Lock-free locks: abort & retry, but repeat forever ⇒ livelock!

Need SCXs to “lock” in consistent order
(⇒ one will eventually succeed)

Progress: livelock

Problems arise if different SCX operations lock records in different orders.

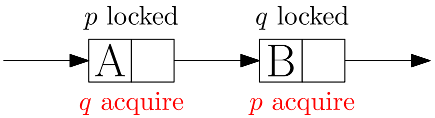


1. *p* locks *A*, *B*
   q locks *B*, *A*
2. Real locks: deadlock!
3. Lock-free locks: abort & retry, but repeat forever ⇒ livelock!

Need SCXs to "lock" in consistent order
(⇒ one will eventually succeed)

Progress: livelock

Problems arise if different SCX operations lock records in different orders.

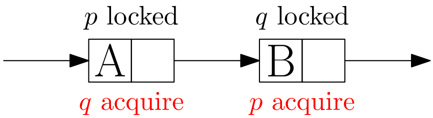


1. *p* locks *A*, *B*
   q locks *B*, *A*
2. Real locks: deadlock!
3. Lock-free locks: abort & retry, but repeat forever ⇒ livelock!

Need SCXs to "lock" in consistent order
(⇒ one will eventually succeed)

Progress: livelock

Problems arise if different SCX operations lock records in different orders.



1. p locks A, B
   q locks B, A
2. Real locks: deadlock!
3. Lock-free locks: abort & retry, but repeat forever $\Rightarrow$ livelock!

Need SCXs to “lock” in consistent order
($\Rightarrow$ one will eventually succeed)

Avoiding livelock intelligently

Constraint

After SCX's stop succeeding, eventually all new SCX's must have consistent order on V-sets.

Easy to satisfy, because you can ignore concurrency.

Complexity

With no contention:
SCX performs

- $k + 1$ CAS steps if it depends on k LLXs
- $f + 2$ writes if it finalizes f data records

LLX only performs reads.

With contention, LLXs and SCXs may have to help and/or retry.

Future work: Amortized complexity bounds with contention.

Summary

Contributions:

- Semantics of LLX and SCX
  (could be implemented, e.g., with HTM)
- Vastly simplifies proofs of correctness for non-blocking data structure implementations

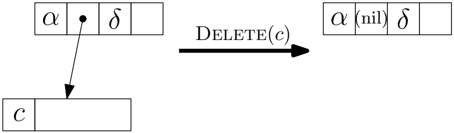
Further work:

- VLX (generalizes validate instruction)
- Non-blocking balanced BSTs
  (and template for building other trees)
- Experimental results

Extra slides

When k-compare-single-swap (kCSS) is inefficient

Example: tree where each node has 32 child pointers (or keys).



Requires a 33-compare-single-swap operation

With no contention:

- kCSS: 2 CASs, 2 writes, 66 non-cached reads
- SCX+LLXs: 2 CASs, 1 write, $\leq$ 13 non-cached reads