State of the art in using operation semantics to boost concurrency

Marek Zawirski
INRIA & UPMC, France
marek.zawirski@lip6.fr

ANR ConcoRDanT & STREAMS, Paris, June 2011

Part I:

Concurrency-control in synchronous* systems (multi-threaded applications, multi-cores)

Transactions and abstraction level

- Transactional system
- Typical requirements:
  strict serializability / opacity / dynamic atomicity
- Old problem: *efficient* concurrency-control
  - Studied in DB-context, transactional ADT
- New motivation & context: Transactional Memory, multi-cores

Intuitive observations:

- **Low-level primitives (like *read/write*) limit concurrency**
- **High-level abstract data types bring more information to use**
  [Weihl 1988, Gray et al. 1996, Ni et al. 2007,
  Koskinen et al. 2010, …]

Coarse-Grained Transactions

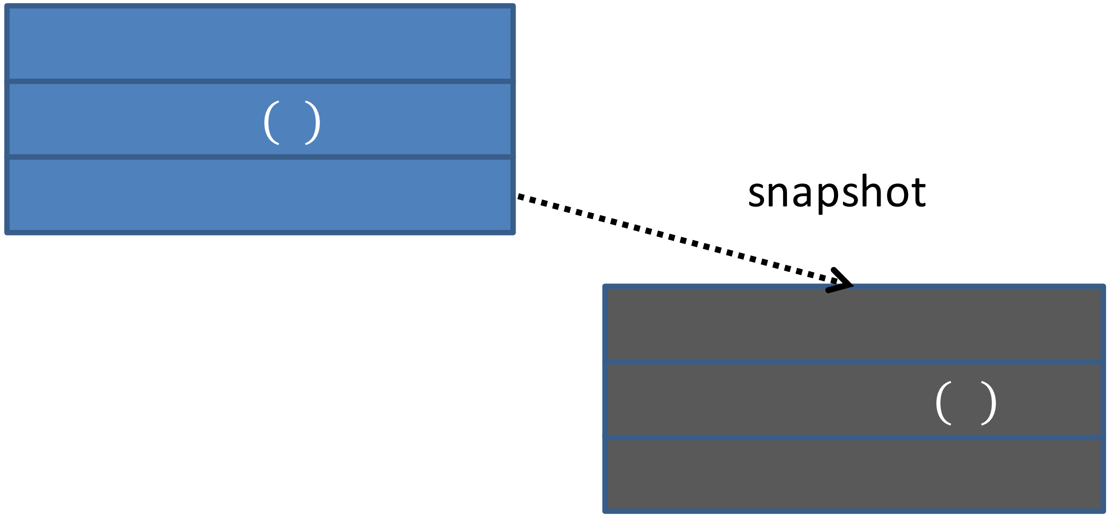
- General model – threads submitting **transactions**
- Thread accesses **thread-local variables** or invokes methods on **shared linearizable objects** (within transaction)
- Simple language defining thread's program:

```
beg    // transaction
  res:=set.contains(”a”)
  if (res) then
    set.add(”b”)
  else
    set.add(”a”)
cmt
```

- **Generic execution semantics** exploiting allowable concurrency Output: *strict serializable* & *opaque* histories
- Defined by nondeterministic automata

[Koskinen et al. 2010]:

CGT: optimistic execution semantics



~Committed trace (way simplified!)

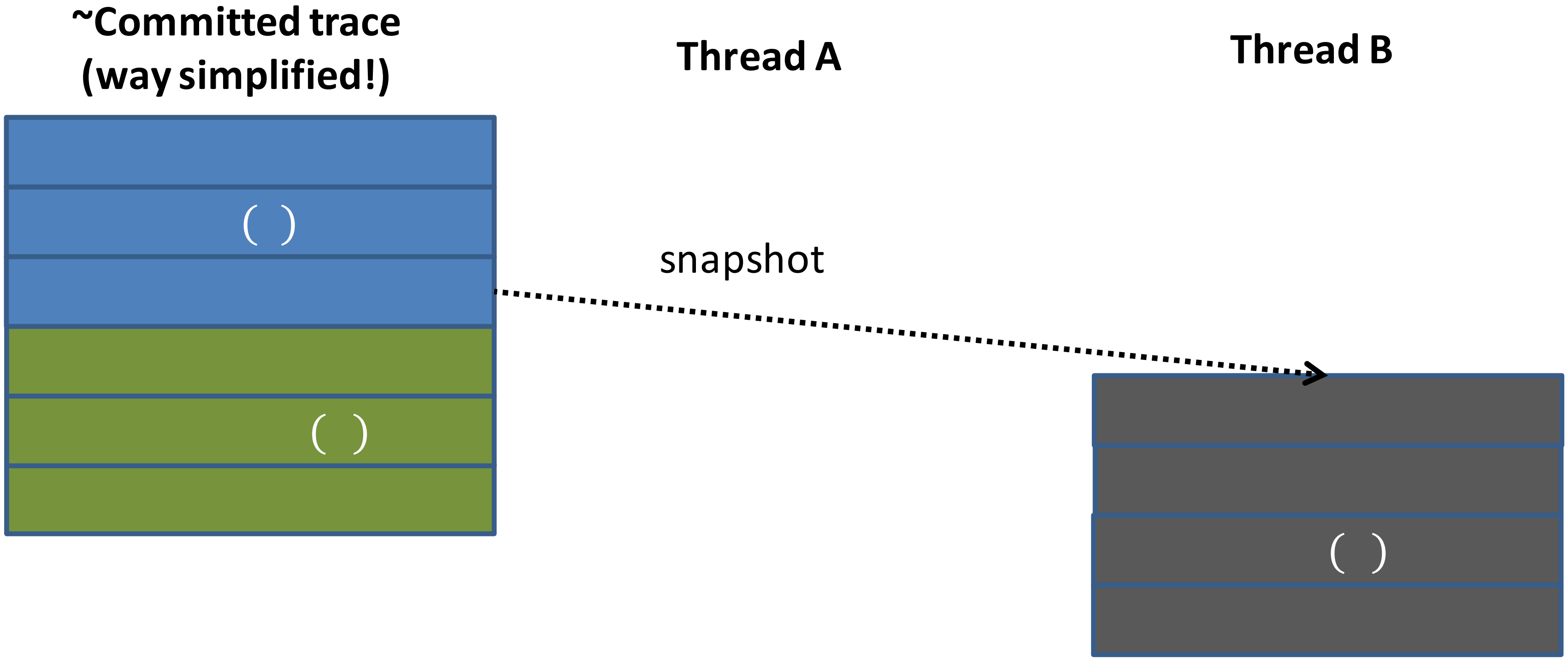
Thread A

snapshot

- Snapshot - commited state
- Works in isolation on snapshot
- Apply changes on the shared state

[Koskinen et al. 2010]:

CGT: optimistic execution semantics



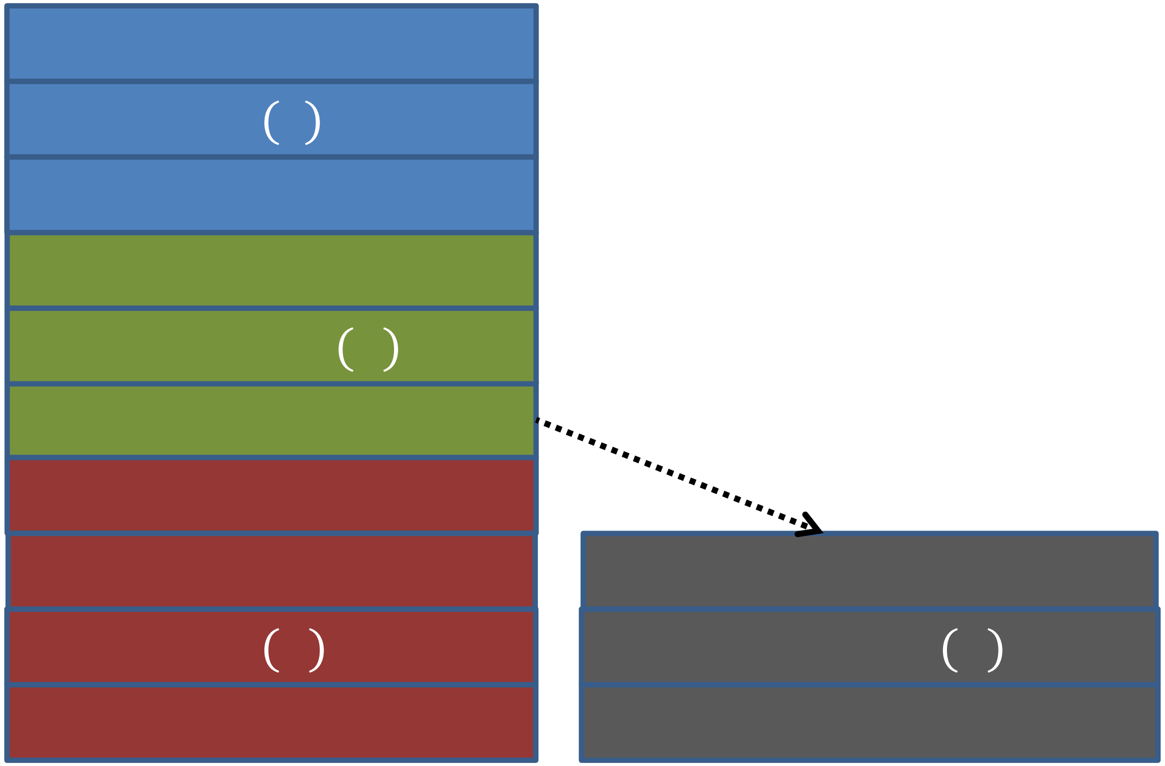
Commitment guard rule:

- All methods executed by must be ***right-movers*** of methods concurrently committed by
- ( ) ( )

(similar relations: left-movers and both-movers)

[Koskinen et al. 2010]:

CGT: optimistic execution semantics



~Committed trace (way simplified!)

Thread A

Thread B

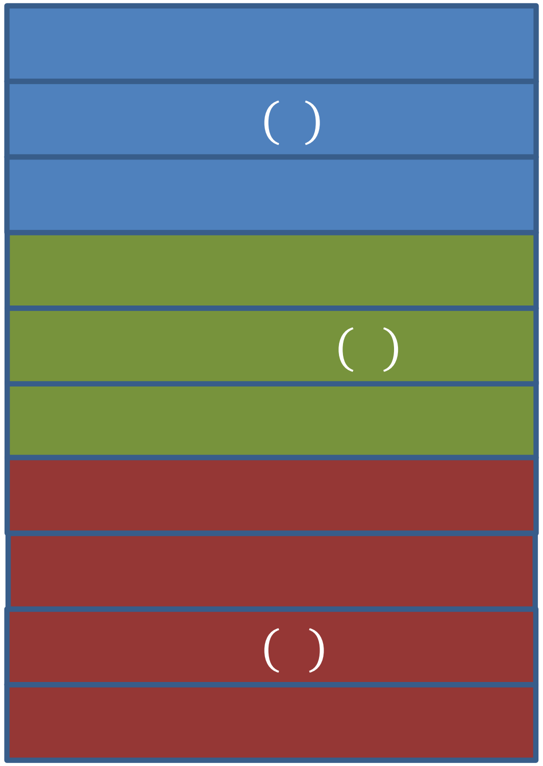
- ( ) ( )

Need to abort!

[Koskinen et al. 2010]:

CGT: optimistic execution semantics

~Committed trace (way simplified!)

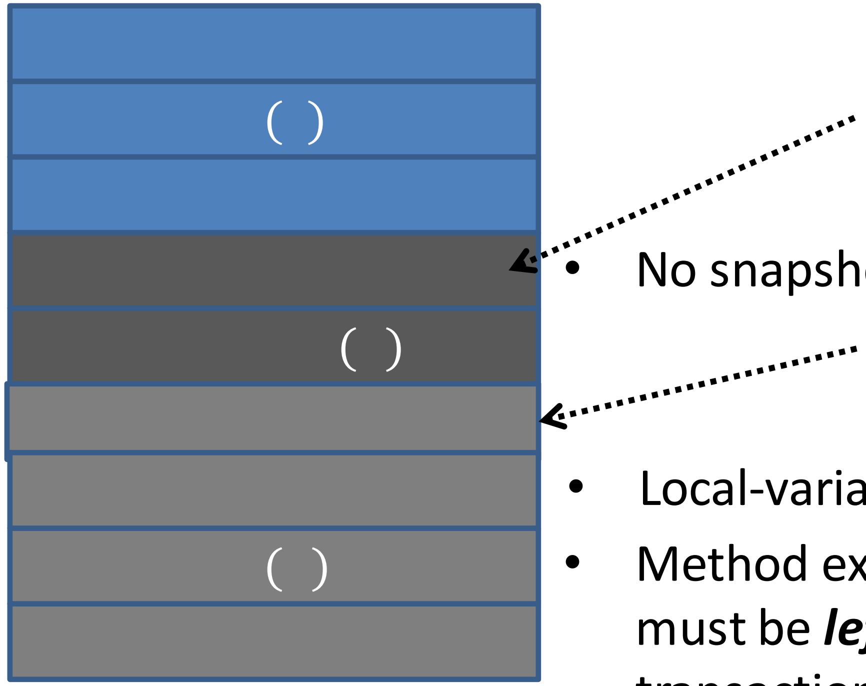


- Optimistic execution semantics subsumes most read/write-set based STM implementations!
- I.e. implementation using shared memory object with methods ( ) and ( )
- Not much space for concurrency at low-level!

( ) ( )
( ) ( )
( ) ( )
( ) ( )

[Koskinen et al. 2010]:

CGT: pessimistic execution semantics



Thread A

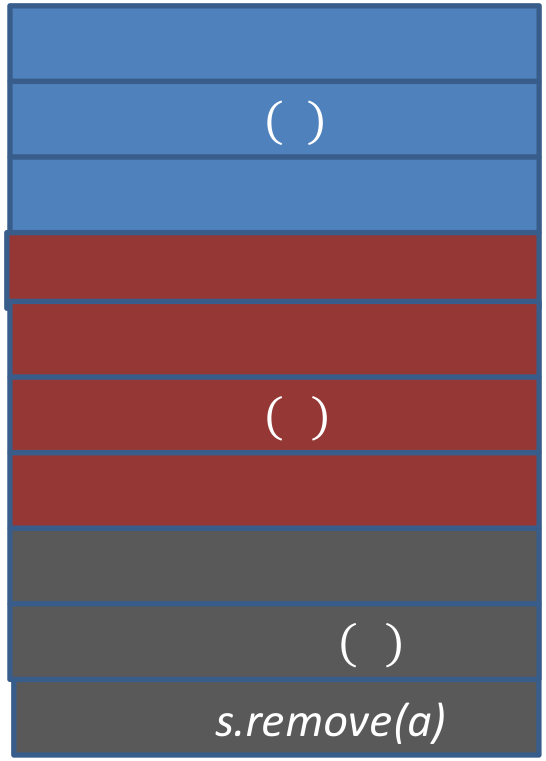
- No snapshot, working with live-organism (shared state)!

Thread B

- Local-variable operation
- Method execution guard:
  must be ***left-mover*** of all methods executed by active transaction : ( ) ( )
- Commit at any time! (no guard)

[Koskinen et al. 2010]:

CGT: pessimistic execution semantics



- Undo, e.g. due to deadlock detection

- Pessimistic semantics captures Transactional Boosting implementation [Koskinen & Herlihy 2008]
- Inherent differences between the two semantics

[Koskinen et al. 2010]:

Turning theory into practice

- How to determine methods moverness/commutativity?
  - Automated analysis of object specification and/or code: some ideas in [Rinard & Diniz 1997, Aleen & Clark 2009]
  - Model-checking [Dennis et al. 2004]
  - Manual analysis

- How to make use of such commutativity specification?
  - Runtime needs to use conflicts (non-movers) detection algorithm to implement pessimistic or optimistic semantics
  - Generic implementations and/or methodology needed!
  - E.g. Transactional Boosting [Koskinen & Herlihy 2008]: turns a linearizable object implementation into a transactional object
  - E.g. Commutativity Lattice [Kulkarni et al. 2011]

Commutativity specifications

Linearizable object Set

$add(a):r_1$

$remove(a):r_1$

$contains(a):r_1$

[Kulkarni et al. 2011]

Commutativity specifications

Specification for linearizable object Set

	$[add(b):r_2]_{\sigma 2}$	$[remove(b):r_2]_{\sigma 2}$	$[contains(b):r_2]_{\sigma 2}$
$[add(a):r_1]_{\sigma 1}$			
$[remove(a):r_1]_{\sigma 1}$			
$[contains(a):r_1]_{\sigma 1}$			

holds true

then m_1 and m_2 can be swapped in any history

[Kulkarni et al. 2011]

Commutativity specifications

Invalid specification for linearizable object Set

	$[add(b):r_2]_{\sigma 2}$	$[remove(b):r_2]_{\sigma 2}$	$[contains(b):r_2]_{\sigma 2}$
$[add(a):r_1]_{\sigma 1}$			
$[remove(a):r_1]_{\sigma 1}$			
$[contains(a):r_1]_{\sigma 1}$			

[Kulkarni et al. 2011]

Commutativity specifications

Specification for linearizable object Set

	$[add(b){:}r_2]_{\sigma 2}$	$[remove(b){:}r_2]_{\sigma 2}$	$[contains(b){:}r_2]_{\sigma 2}$
$[add(a){:}r_1]_{\sigma 1}$			
$[remove(a){:}r_1]_{\sigma 1}$			
$[contains(a){:}r_1]_{\sigma 1}$			

Generalization to order on valid specifications:

[Kulkarni et al. 2011]

Commutativity specifications

Least specification for linearizable object Set

	$[add(b):r_2]_{\sigma 2}$	$[remove(b):r_2]_{\sigma 2}$	$[contains(b):r_2]_{\sigma 2}$
$[add(a):r_1]_{\sigma 1}$			
$[remove(a):r_1]_{\sigma 1}$			
$[contains(a):r_1]_{\sigma 1}$			

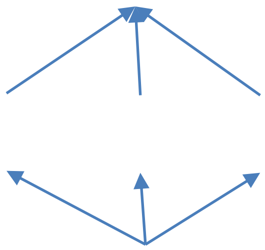
[Kulkarni et al. 2011]

Commutativity specifications

Precise specification for linearizable object Set

	$[add(b):r_2]_{\sigma 2}$	$[remove(b):r_2]_{\sigma 2}$	$[contains(b):r_2]_{\sigma 2}$
$[add(a):r_1]_{\sigma 1}$			
$[remove(a):r_1]_{\sigma 1}$			
$[contains(a):r_1]_{\sigma 1}$			

Partially ordered set of valid specifications ( ) with and constitutes a lattice!



[Kulkarni et al. 2011]

Specif. classes & implementations

Goal: *sound* and *complete* online commutativity checker

Specification class	Allowed logic for condition	Example condition	Checker implementation
SIMPLE	**L1:** *or* conjunctions on arguments or retur values	(set)	Abstract locking [Ni et al. 2007], generalized
ONLINE-CHECKABLE	**L2**: Function on component (args,), but not both on and	(kd-tree) ( )	Method logging
OTHERS	**L3: L2 +** functions on both and	( )]	Method logging + undo

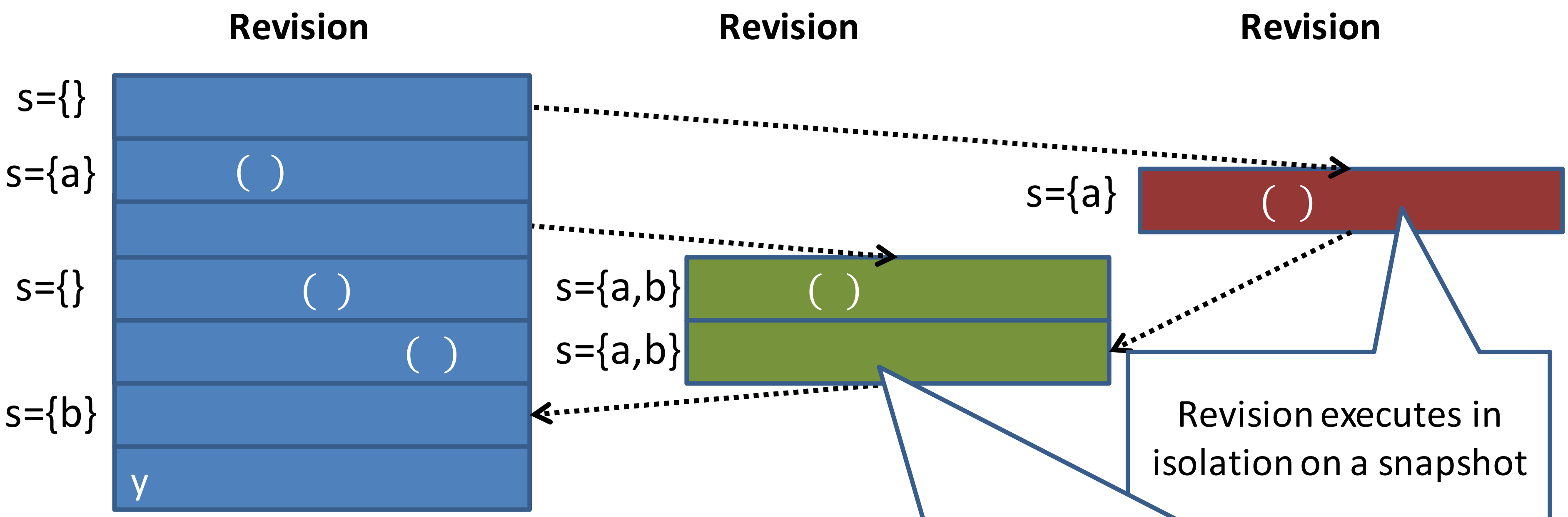
Findings (theoretical & experimental):

- Overhead of implementation does not pay off in all cases!
- Lattice should be exploited for a particular application and object

[Kulkarni et al. 2011]

Programming in Concurrent Revisions

Limited fork & join model, inspired by Unix processes and revision ctrl



Revision executes in isolation on a snapshot

Join is blocking, never fails! It triggers per-object three-way-merge:

( )

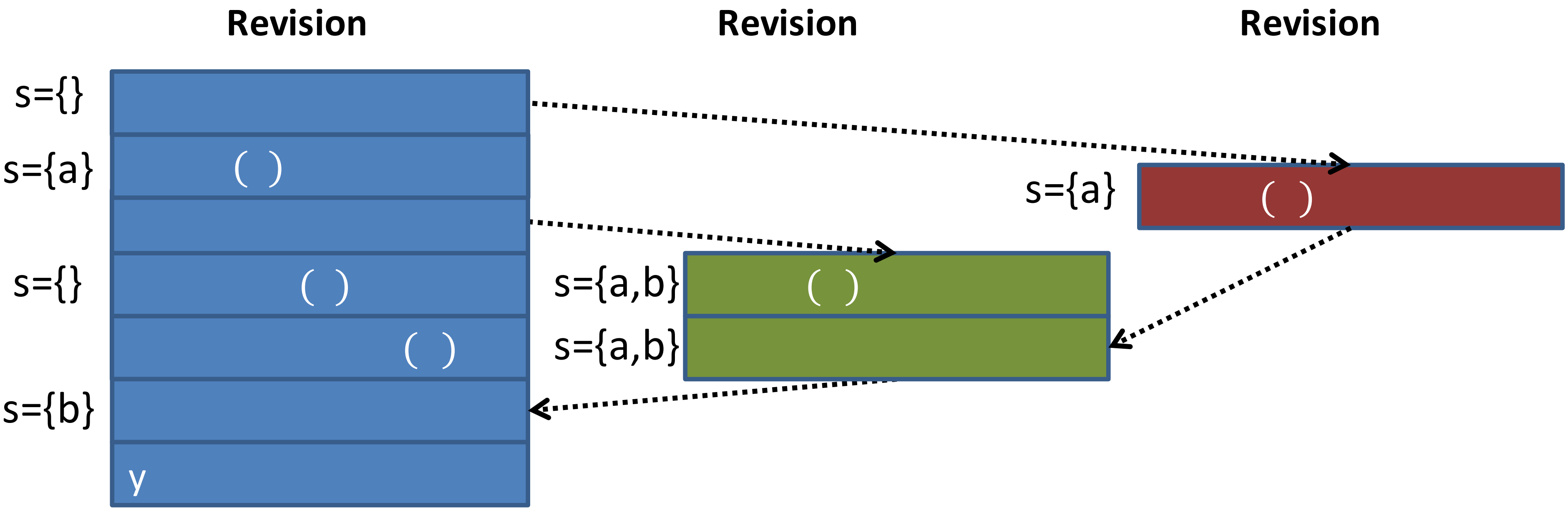
Merge is an arbitrary function:

))

[Burckhardt & Leijen 2011]

Programming in Concurrent Revisions

Limited fork & join model, inspired by Unix processes and revision ctrl



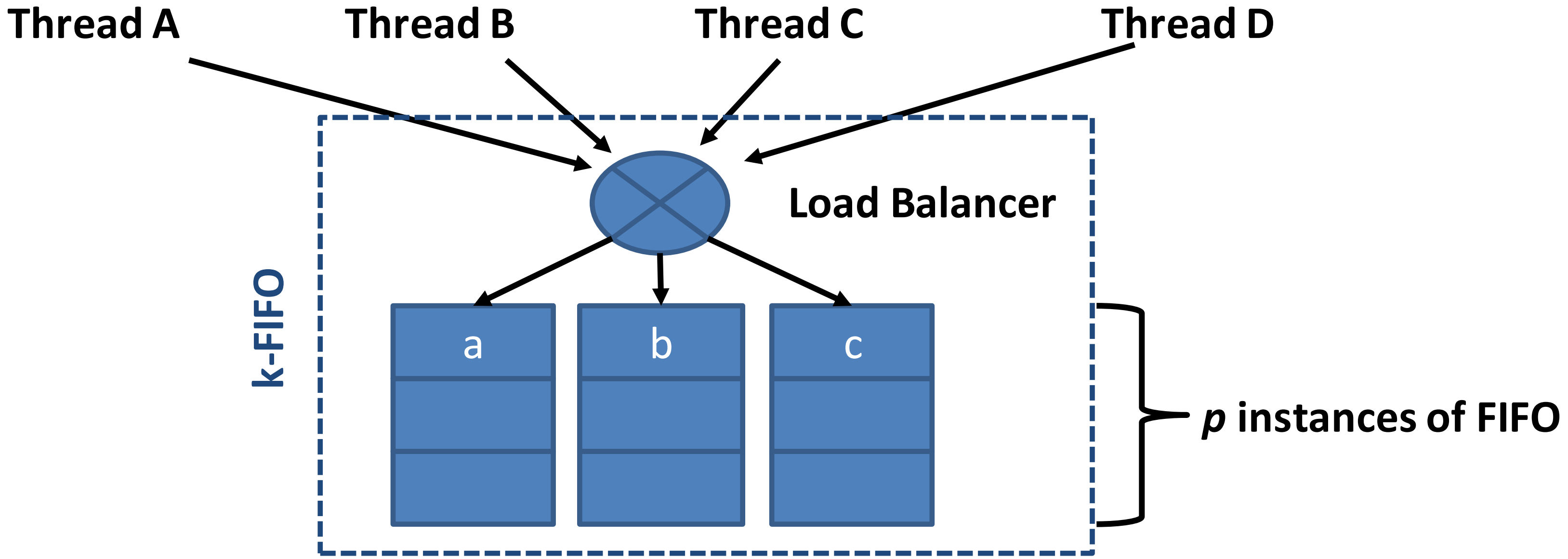
Interesting properties:

- Joined revision is never “aborted”
- Merge is custom, may union the results, give priority to a particular revision etc.
- “Abelian” objects have *sequential merges* (e.g. counter with add() operation)
- Computation is deterministic

[Burckhardt & Leijen 2011]

Implementing linearizable object

- Hot topic in the age of multi/many-cores:
  - Lock-free implementations; some general techniques [Herlihy&Shavit]: exchangers (op. inverses), elimination arrays/trees, combiners…
  - For hot-spot objects, linearizable implementation may be too costly!
- k-linearizable implementations: k-FIFO [Payer et al. 2011]



- Outcome: better performance, but up to *k*-reorderings: *k=f(p, LB quality)*
- Duality of specification: altered sequential specification or linearizability

Part II:

Replication in asynchronous systems

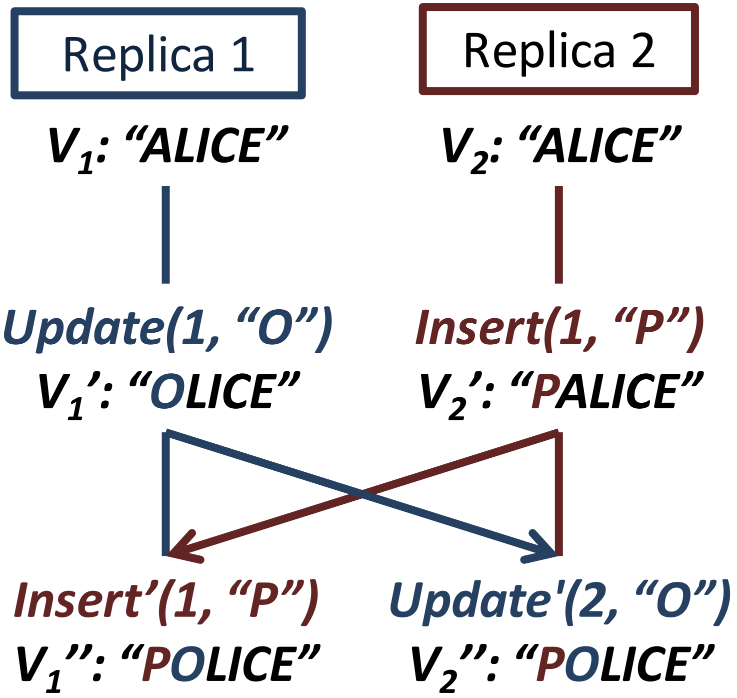
Systems using operation constraints

- High-level operations with constraints
- System tries to ensure operations constraints
- Might end-up in the conflict, application is made aware of a problem and assisted in resolution
- Conflict-resolution typically requires coordination & rollback
- Working systems:
  - Telex [Pierpaolo’s presentation, paper 2009] and older systems
  - Similar ideas in video authoring [Novikov et al. 2003] and CoAct system [Klingemann & Tesch 1996]

(Hidden) Commutativity at the extreme

- **Operational Transformation:** Operation performed locally without blocking & propagated
- *Transformation Function* used against concurrent operations:

  $$TF(o_1, o_2) = o_1'$$
- *Integration algorithm* and *TF* properties ensure consistency
- Conditions for "consistent" *TF* [Ressel et al. 1996]
  TP1 - for centralized integration alg.
  TP2 – for decenetralized system
- TP2 issues & TTF [Oster et al.]



Replica 1

Replica 2

V_1: "ALICE"

V_2: "ALICE"

Update(1, "O")

Insert(1, "P")

V_1': "OLICE"

V_2': "PALICE"

Insert'(1, "P")

Update'(2, "O")

V_1'': "POLICE"

V_2'': "POLICE"

Here*: TF* modifies an index

$$V_1'' = V_2''$$

Commutative Replicated Data Types

- Explicit commutativity at the extreme: all operations commute
- **1st era of CRDTs**:
  - LWW [Thomas 1979]
  - Dictionary & Log [Wuu & Bernstein 1984] – apparent commutativity
- **2nd era of CRDTs:**
  - WOOT: operations made non-trivially commutative [Oster et al. 2006]
  - RADT: LWW + causality [Roh et al. 2011, 2006]
  - Treedoc: the concept + core-nebula [Preguiça & Shapiro et al. 2007-10]
  - Logoot: undo, hierarchical extension [Weiss et al. 2009, 2010]
- The generic framework + various types portfolio :
  - Conflict-free Replicated Data Types [Shapiro et al. 2011]
  - Convergent Data Types [Baquero & Moura 1999] - equivalent to CRDTs

Alternative ways?

- Keep storage simple, leave the burden on the application
  - Dynamo [DeCandia et al. 2007], Riak, Cassandra

- Reduce RDBMS guarantees and/or features to scale better:
  - PIQL [Armbrust et al. 2010], MegaStore [Baker et al. 2011]

- Use *monotonic logic* programming model to encourage creating pieces of program can run concurrently:
  - Bloom [Alvaro et al. 2011]

Consistency and self-stabilization?

- Enforcing invariants back after they get broken:
  - r-operators [Ducourthial et al. 2001-2005]