

Implementing software transactions

Tim Harris

Microsoft Research Cambridge

Setting

- A managed language, like C# or Java
- Simple “atomic” blocks:
 - Strong atomicity
 - No explicit “abort”
 - Condition synchronization through “retry”
 - No access to native code

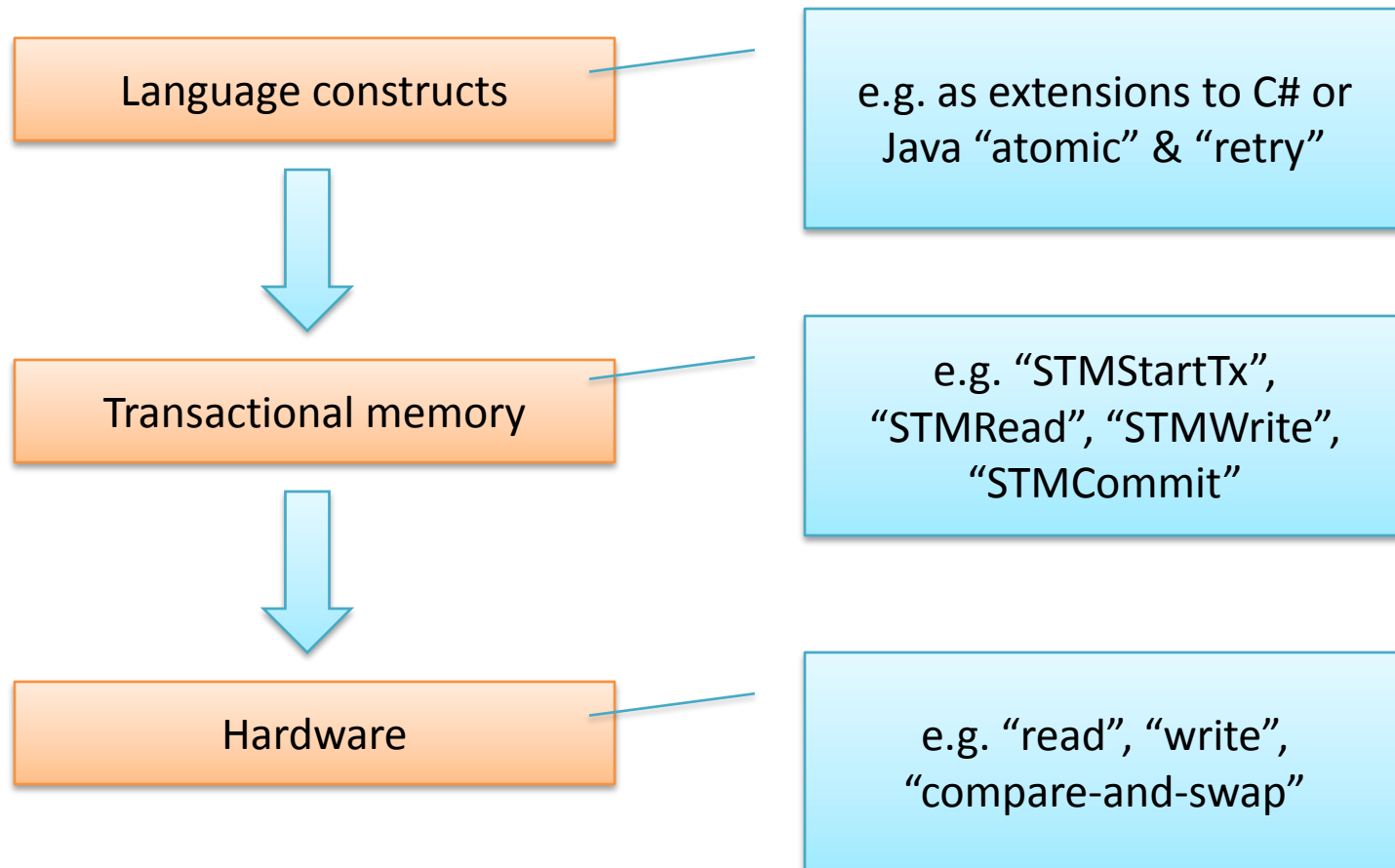
Acknowledgments

- I'm reporting on a lot of other people's research work here
 - Each lecture finishes with future reading suggestions pointing to the original sources
 - Let me know if I've misunderstood or missed something
- My own work's been in collaboration with many others at Cambridge, MSR, and the Microsoft Parallel Computing Platform group
- The “Multiprocessor Architecture” slides are by Maurice Herlihy & Nir Shavit to accompany their excellent book “The art of multiprocessor programming”

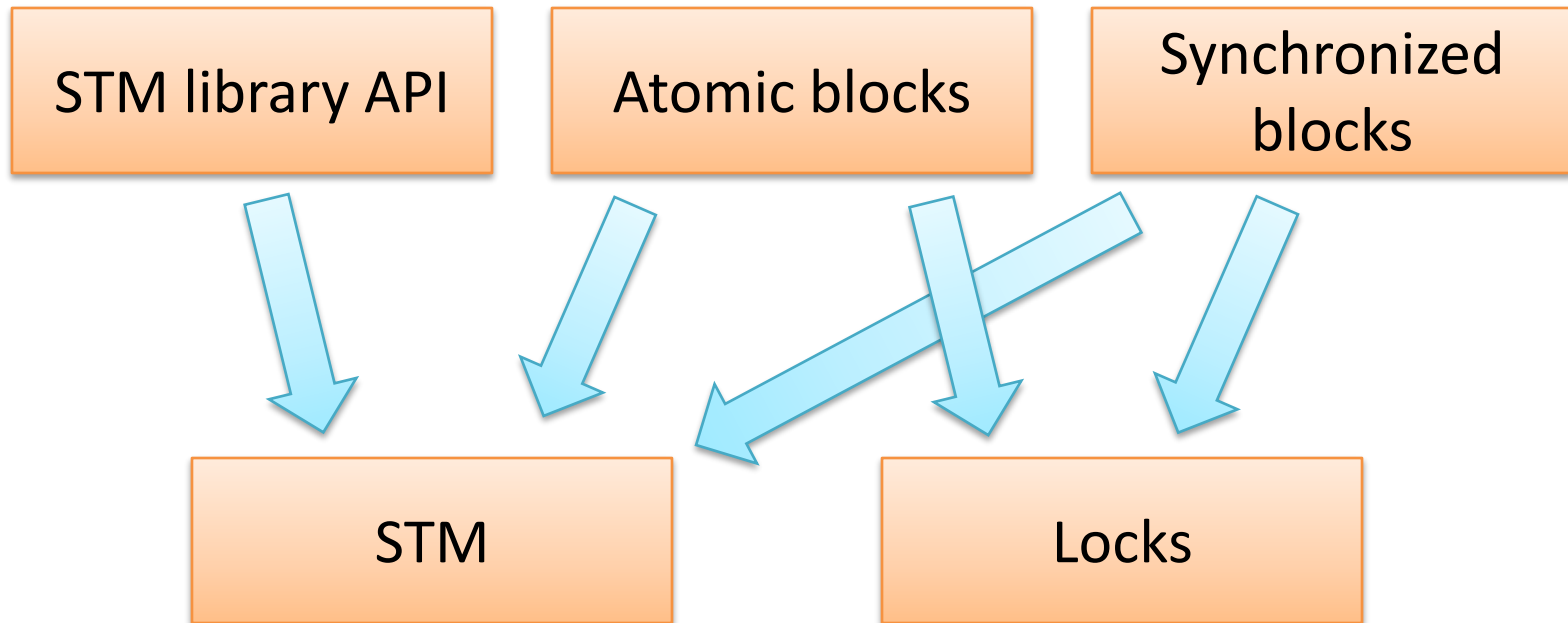
Lecture 1

Introduction, layering,
STM design space

Layering; “atomic” \neq “STM”



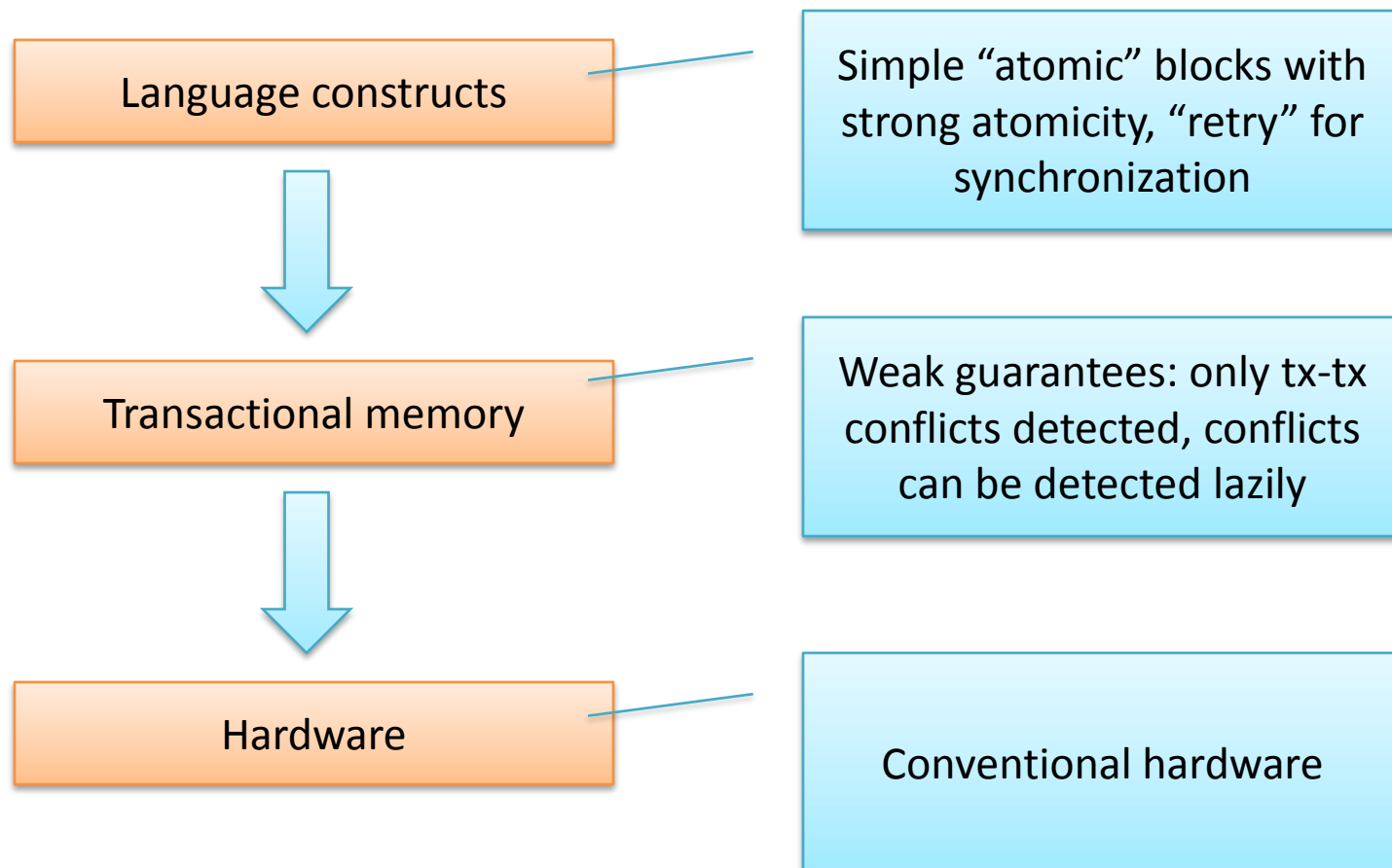
Alternatives to keep in mind



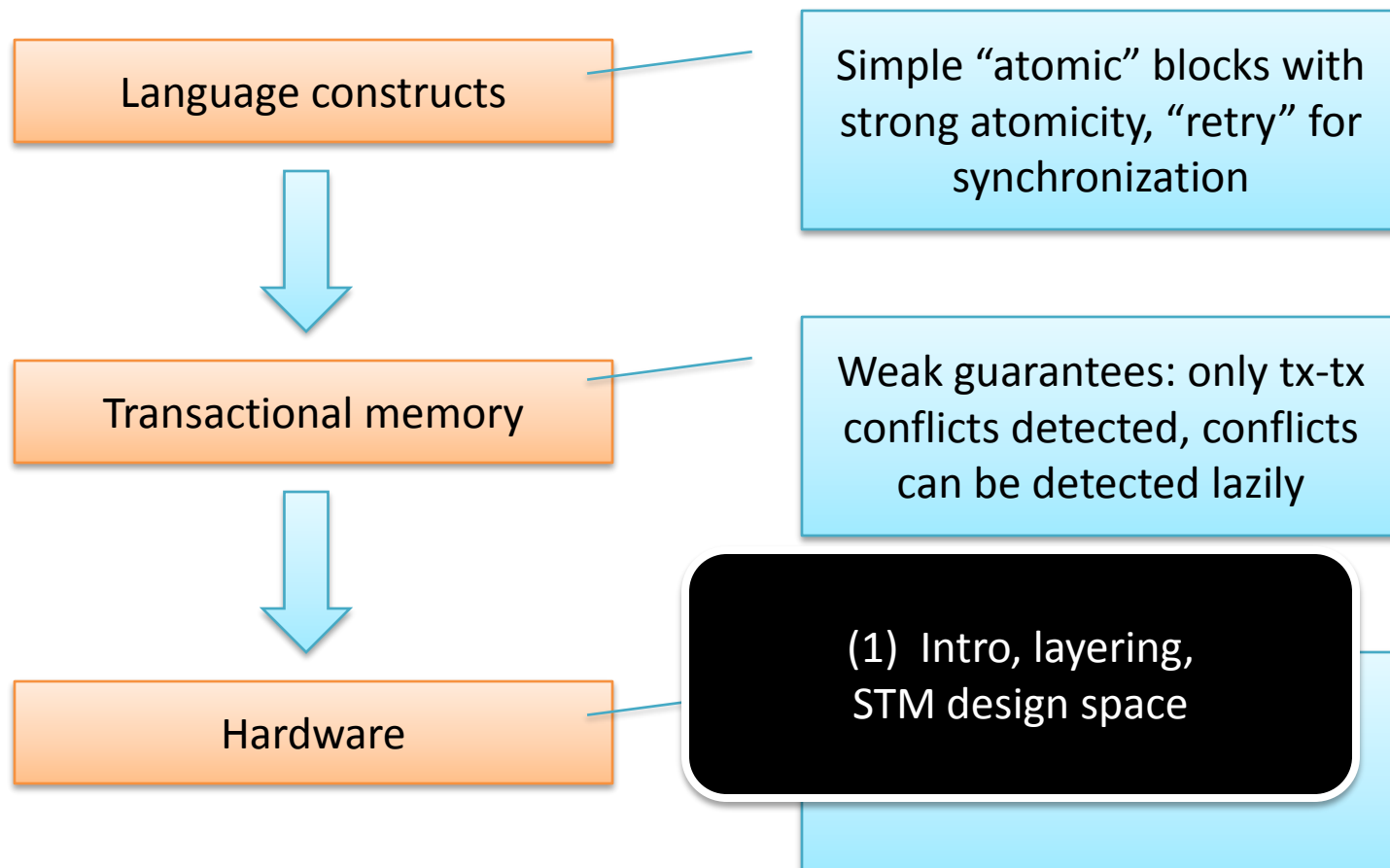
Atomic-over-STM: where do things go?

- We can shift responsibility between
 - Possible extensions to the hardware or OS
 - The STM implementation
 - The way that the STM implementation is used in implementing language features
 - The guarantees (or lack of them) given by the language to the programmer
- Design choices in prototypes often reflect different ways of dividing this responsibility

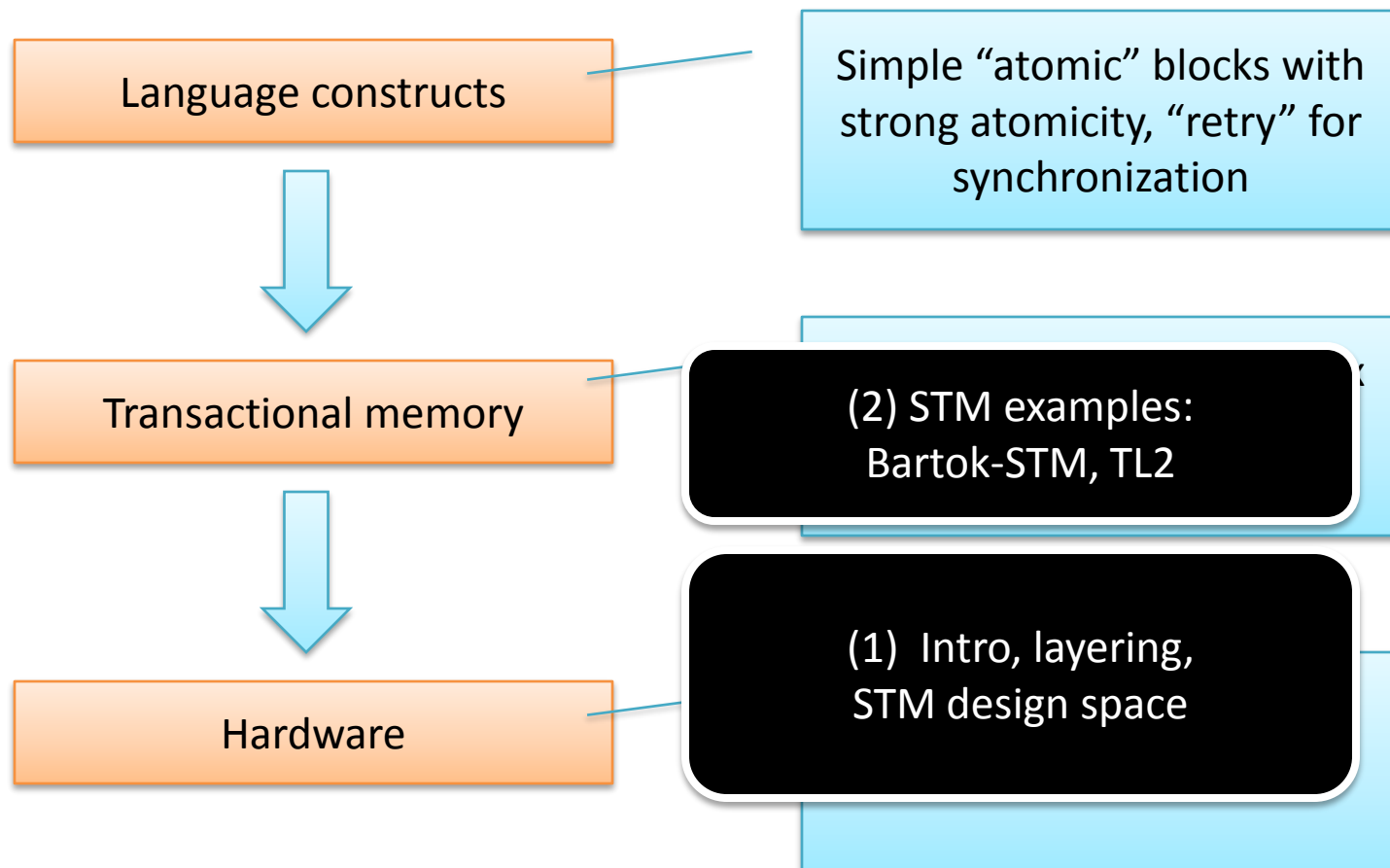
Initial assumptions in these lectures



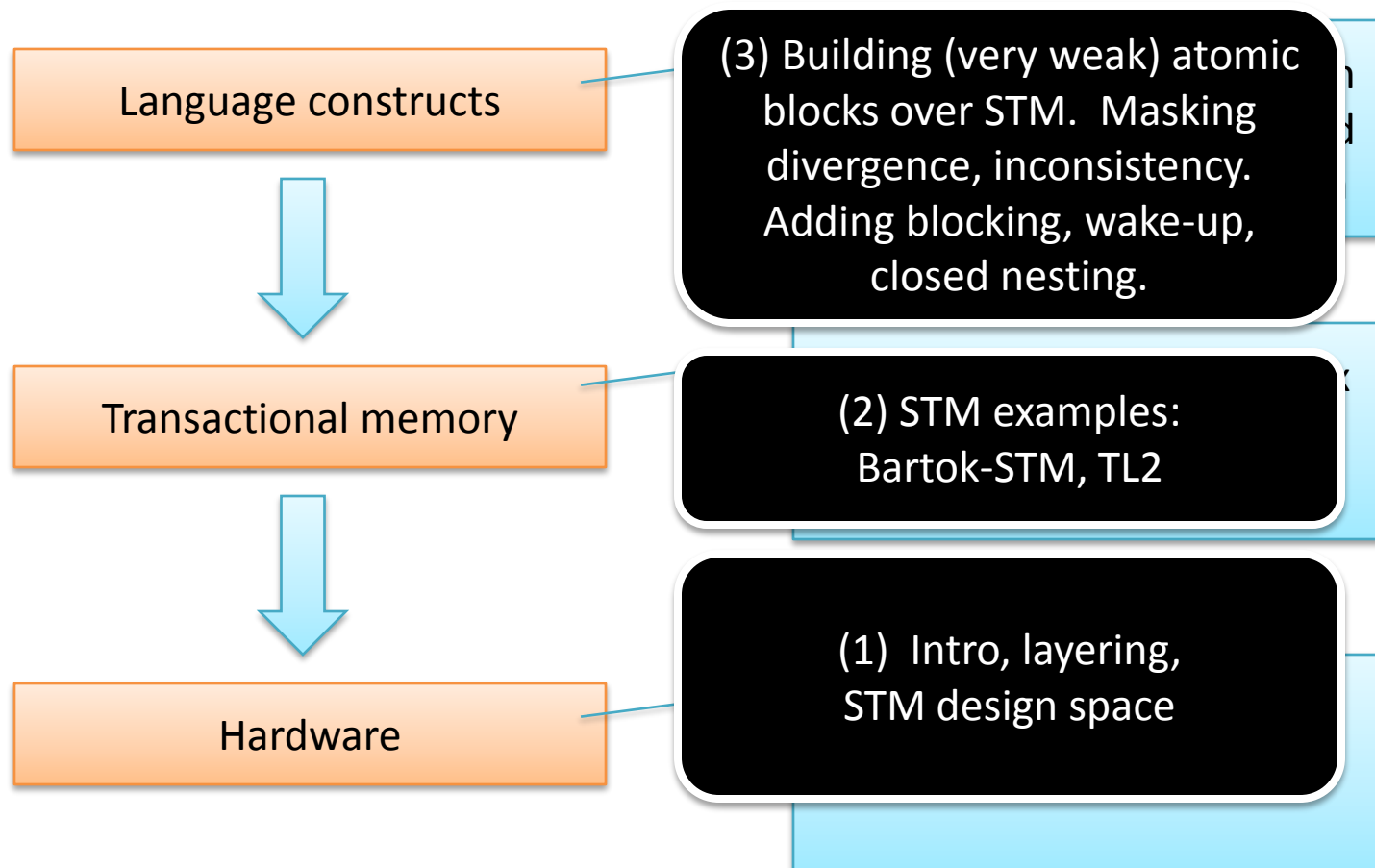
Initial assumptions in these lectures



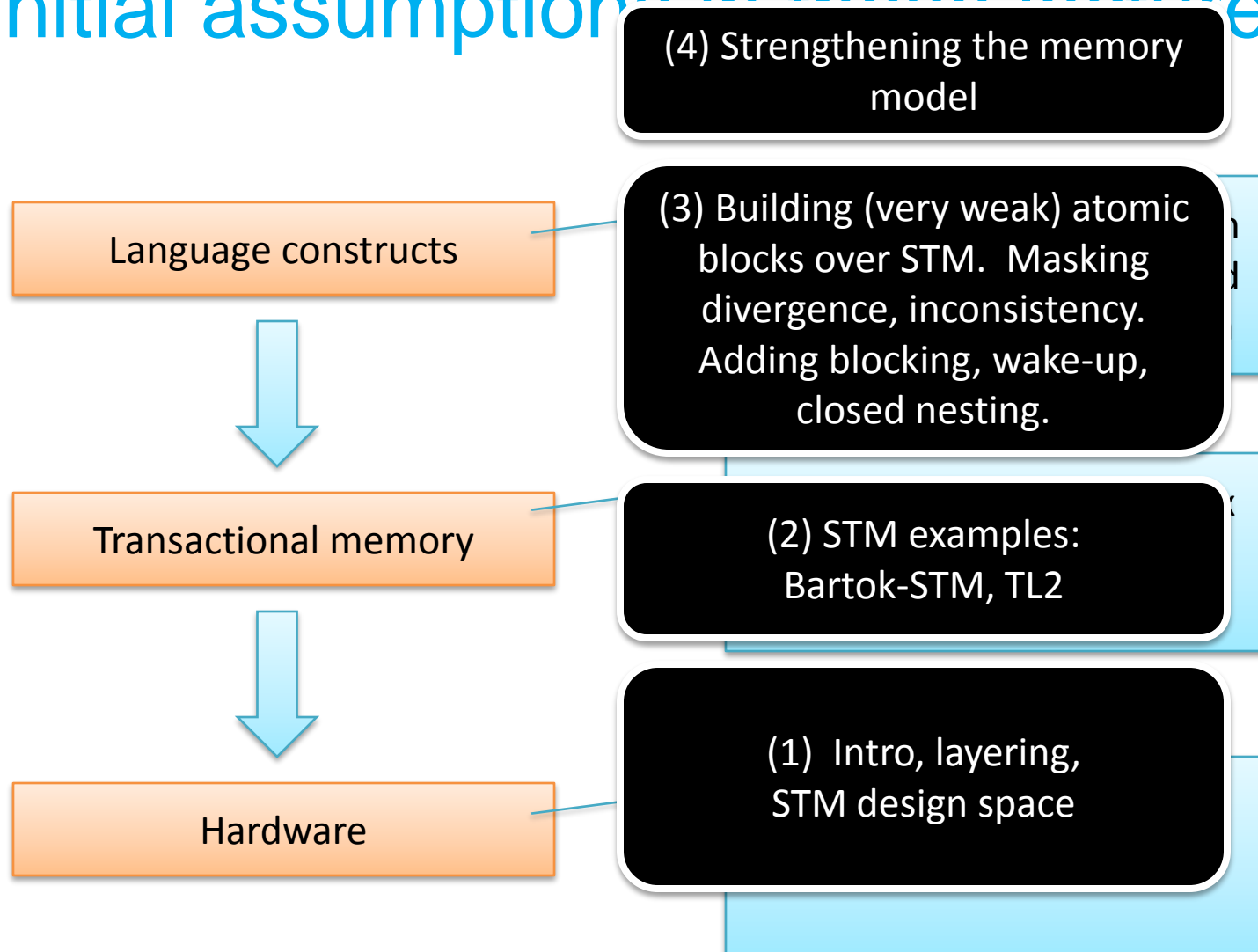
Initial assumptions in these lectures



Initial assumptions in these lectures



Initial assumptions in these lectures



Initial assumptions in these lectures

Language constructs



Transactional memory



Hardware

(5) Research directions:
improving scalability, progress
guarantees

(4) Strengthening the memory
model

(3) Building (very weak) atomic
blocks over STM. Masking
divergence, inconsistency.
Adding blocking, wake-up,
closed nesting.

(2) STM examples:
Bartok-STM, TL2

(1) Intro, layering,
STM design space

Desirable properties for atomic blocks

Fast

- Code inside an atomic block shouldn't run vastly slower than code outside
- What we lose on straight line speed we must make up for on parallelism
 - 10x slower, 100% in atomic blocks => need to use 10 cores
 - Unlikely in practice: Amdahl's law, contention in devices or the memory system

Desirable properties for atomic blocks

Fast

Scalable

- Formalised as disjoint-access parallelism
- We'd like non-conflicting atomic blocks to run in parallel
 1. They should be allowed to commit
 2. Their implementations shouldn't contend in the memory subsystem

Desirable properties for atomic blocks

Fast

Scalable

Predictable

- We may want the programmer to be able to anticipate the likely performance of their code
- Other things being equal we'd like to avoid
 - Fragile static analyses
 - Performance cliffs

Desirable properties for atomic blocks

Fast

Scalable

Predictable

Strong
semantics

- Atomic blocks that “do what they say on the tin”
- ...even if the same location is being accessed directly and inside an atomic block

Desirable properties for atomic blocks

Fast

Scalable

Predictable

Strong
semantics

Guaranteed
progress

- We may want a non-blocking progress guarantee
- (Informally) one thread executing an atomic block shouldn't preclude other threads executing atomic blocks

Desirable properties for atomic blocks

Fast

Scalable

Predictable

Strong
semantics

Guaranteed
progress

In these lectures we focus on the first four properties (a little more on progress in Lecture 5)

Desirable properties for atomic blocks

Fast

Scalable

Predictable

Strong
semantics

Guaranteed
progress

We'll initially concentrate on the first three, and return to strong semantics when looking at compilation

Desirable properties for STM

- **Fast:**
 - Individual primitives should be fast
- **Scalable:**
 - Primitives should scale internally (e.g. operations on different locations don't conflict in memory)
 - Transactions build over these primitives should scale (e.g. one transaction should not be forced to roll-back by non-conflicting accesses in another transaction)
- **Predictable:**
 - Cases where the implementation introduces costs or false conflicts should be able to be anticipated

Example STM primitive API

- `tx = StartTx()`
- `b = CommitTx(tx)`
- `b = ValidateTx(tx)`
- `AbortTx(tx)`

- `v = ReadTx(tx, addr)`
- `WriteTx(tx, addr, v)`

Transaction management: start a transaction, attempt to commit one, force a transaction to abort, test if a transaction is valid

All transacted memory accesses use explicit `ReadTx / WriteTx` operations

Atomic swap

```
void Swap(int *a, int *b)
{
    do {
        tx = StartTx();
        va = ReadTx(tx, &a);
        vb = ReadTx(tx, &b);
        writeTx(tx, &a, vb);
        writeTx(tx, &b, va);
    } while (!CommitTx());
}
```


Linked-list insertion

```
bool Insert(int newVal) {
    do {
        tx = StartTx();
        prev = head;
        n = ReadTx(tx, &prev.next);
        v = ReadTx(tx, &n.val);
        while (v < newVal) {
            prev = n;
            n = ReadTx(tx, &prev.next);
            v = ReadTx(tx, &n.val);
        }
        newCell = new Cell(newVal, n);
        WriteTx(tx, &prev.next, newCell);
    } while (!CommitTx());
}
```

What are the problems here?

Taxonomy: consistency during tx

- Gold standard:
 - During execution a transaction runs against a consistent view of memory
 - Won't be “tricked” into looping, etc.
 - “Opacity”
- What are the advantages / disadvantages when compared with an implementation giving weaker guarantees?

Taxonomy: lazy/eager versioning

- We need some way to manage the tentative updates that a transaction is making
 - Where are they stored?
 - How does the implementation find them (so a transaction's read sees an earlier write)?
- Lazy versioning: only make “real” updates when a transaction commits
- Eager versioning: make updates as a transaction runs, roll them back on abort
- What are the advantages, disadvantages?

Taxonomy: lazy/eager conflict detection

- We need to detect when two transactions conflict with one another
- Lazy conflict detection: detect conflicts at commit time
- Eager conflict detection: detect conflicts as transactions run
- Again, what are the advantages, disadvantages?

Taxonomy: word/object based

- What granularity are conflicts detected at?
- Object-based:
 - Access to programmer-defined structures (e.g. objects)
- Word-based:
 - Access to words (or sets of words, e.g. cache lines)
 - Possibly after mapping under a hash function
- What are the advantages and disadvantages of these approaches?

Next lecture: two examples

- Bartok-STM
 - Eager version management, generally lazy conflict detection (eager for write-write – “encounter time locking”)
 - Reads do not reflect a consistent view of the heap
- TL2
 - Lazy version management, lazy conflict detection, commit-time locking
 - Reads reflect a consistent view of the heap

Summary

- Distinguishing language features (atomic) from implementation techniques (STM)
- In building “atomic” we can
 - Build an STM with strong properties
 - Build strong properties over a weaker STM
- Axes for distinguishing STM primitives:
 - Consistency during execution
 - Version management mechanism
 - Conflict detection point
 - Conflict granularity

Further reading

- “Transactional memory: architectural support for lock-free data structures”, Maurice Herlihy, J Eliot B Moss. ISCA 1993. *Introduced transactional memory.*
- “Software transactional memory”, Nir Shavit, Dan Touitou. PODC 1995. *Introduced STM.*
- “Software transactional memory for dynamic-sized data structure”, Maurice Herlihy, Victor Luchangco, Mark Moir, William N Scherer III. PODC 2003. *One of the first practically-focused object-based STMs.*
- “Concurrent programming without locks”, Keir Fraser, Tim Harris. ACM TOCS, May 2007. *Early practically-focused word-based and object-based STMs.*
- “LogTM: log-based transactional memory”, Kevin E Moore, Jayaram Bobba, Michelle J Morovan, Mark D Hill, David A Wood. HPCA 2006. *Introduced the eager/lazy taxonomy for versioning and conflict detection.*
- “On the correctness of transactional memory”, Rachid Guerraoui, Michal Kapalka. PPOPP 2008. *Introduced the idea of “opacity”.*
- Rochester Software Transactional Memory (<http://www.cs.rochester.edu/research/synchronization/rstm/>) a research prototype STM API for C++.
- SXM and DSTM (<http://www.cs.brown.edu/~mph/>) research prototype STM APIs for C# and Java.

Lecture 2

STM design

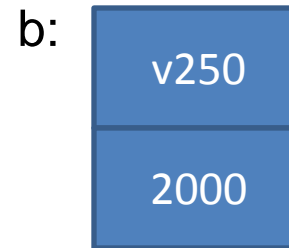
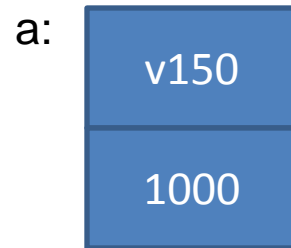
Two examples

- Bartok-STM
 - Eager version management, generally lazy conflict detection (eager for write-write – “encounter time locking”)
 - Reads do not reflect a consistent view of the heap
- TL2
 - Lazy version management, lazy conflict detection, commit-time locking
 - Reads reflect a consistent view of the heap

Bartok-STM

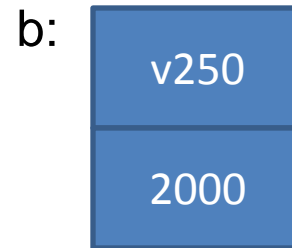
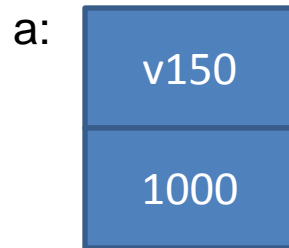
- Use per-object meta-data (“TMWs”)
- Each TMW is either:
 - Locked, holding a pointer to the transaction that has the object open for update
 - Available, holding a version number indicating how many times the object has been locked
- Writers eagerly lock TMWs to gain access to the object, using eager version management
 - Maintain an undo log in case of roll-back
- Readers log the version numbers they see and perform lazy conflict detection at commit time

Example: uncontended swap

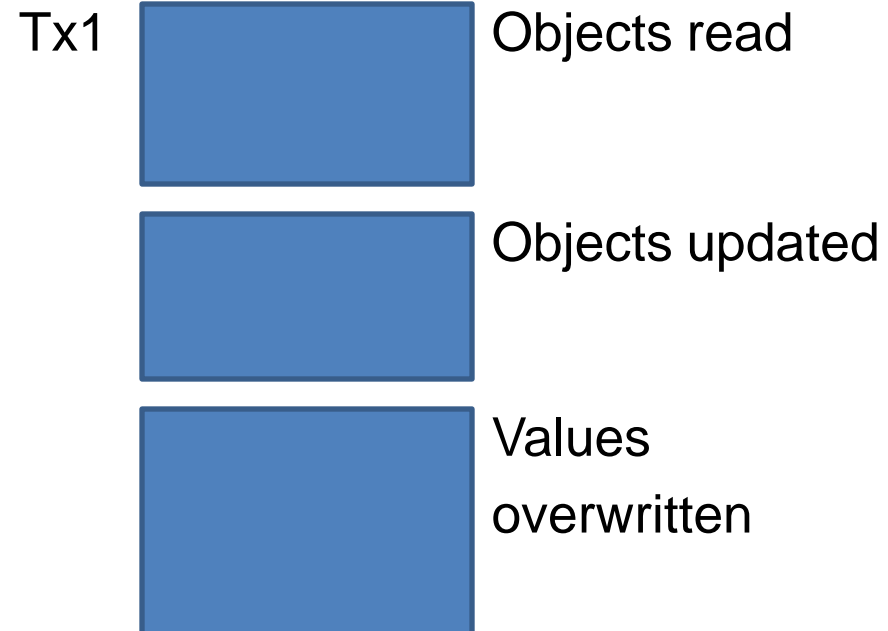


```
void Swap(int *a, int *b)
{
  do {
    tx = StartTx();
    va = ReadTx(tx, &a);
    vb = ReadTx(tx, &b);
    writeTx(tx, &a, vb);
    writeTx(tx, &b, va);
  } while (!CommitTx());
}
```

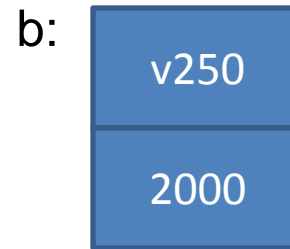
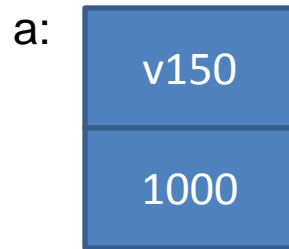
Example: uncontended swap



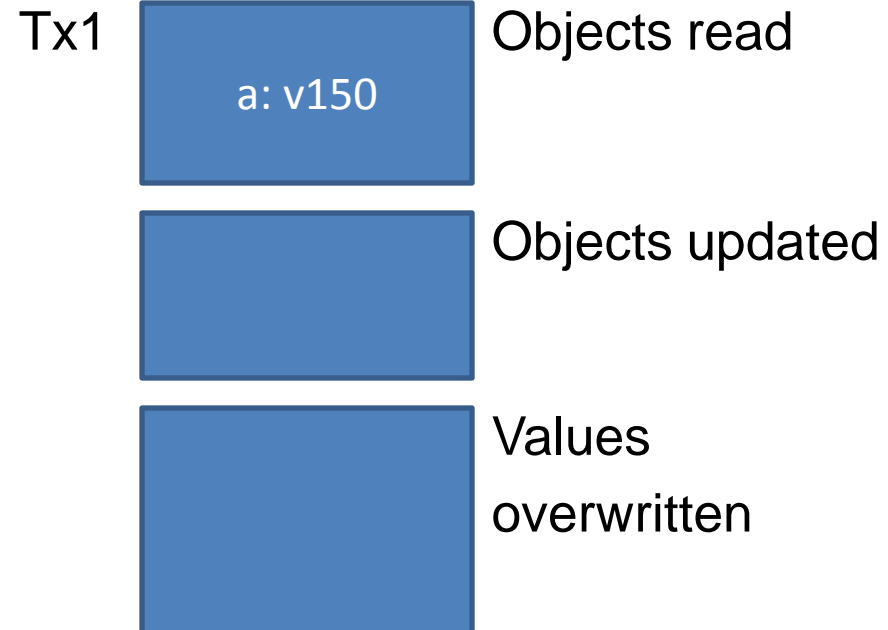
```
void Swap(int *a, int *b)
{
  do {
    tx = StartTx();
    va = ReadTx(tx, &a);
    vb = ReadTx(tx, &b);
    writeTx(tx, &a, vb);
    writeTx(tx, &b, va);
  } while (!CommitTx());
}
```



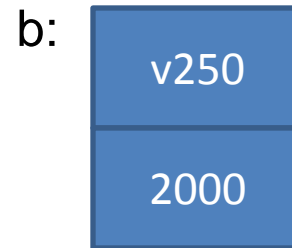
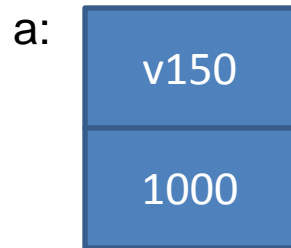
Example: uncontended swap



```
void Swap(int *a, int *b)
{
  do {
    tx = StartTx();
    va = ReadTx(tx, &a);
    vb = ReadTx(tx, &b);
    writeTx(tx, &a, vb);
    writeTx(tx, &b, va);
  } while (!CommitTx());
}
```

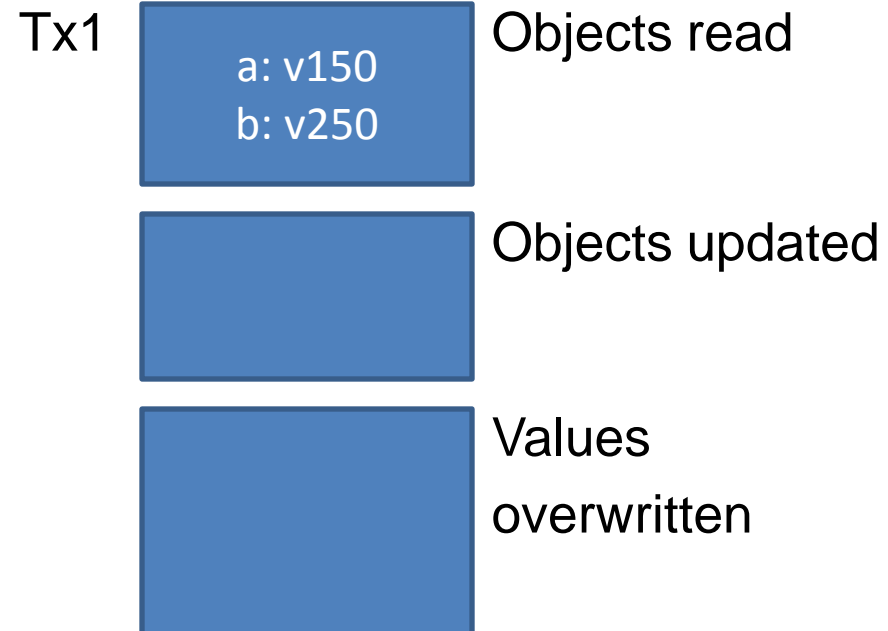


Example: uncontended swap

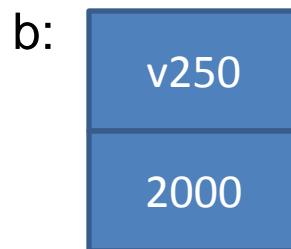
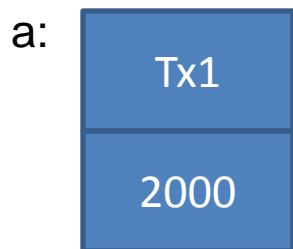


```

void Swap(int *a, int *b)
{
  do {
    tx = StartTx();
    va = ReadTx(tx, &a);
    vb = ReadTx(tx, &b);
    writeTx(tx, &a, vb);
    writeTx(tx, &b, va);
  } while (!CommitTx());
}
    
```

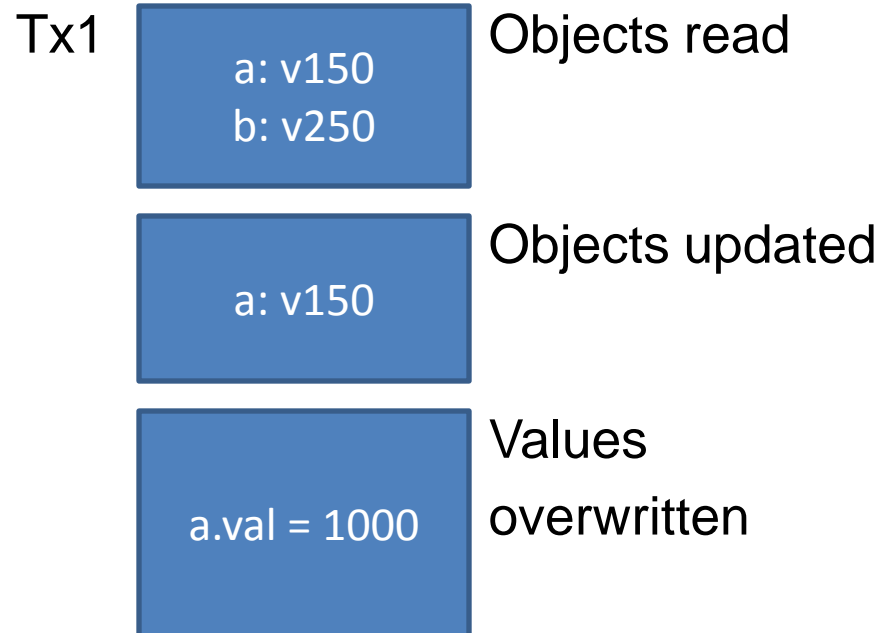


Example: uncontended swap

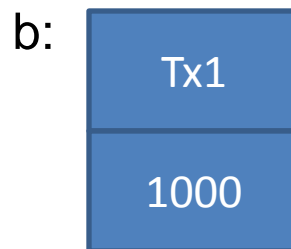
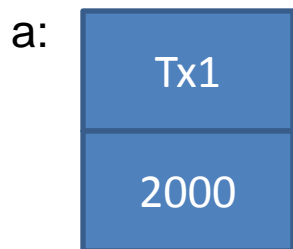


```

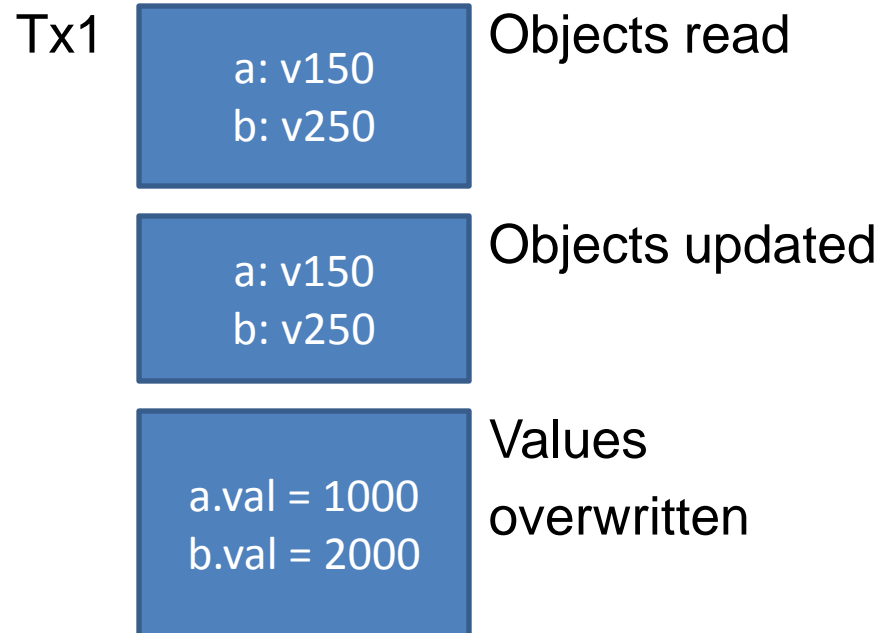
void Swap(int *a, int *b)
{
  do {
    tx = StartTx();
    va = ReadTx(tx, &a);
    vb = ReadTx(tx, &b);
    writeTx(tx, &a, vb);
    writeTx(tx, &b, va);
  } while (!CommitTx());
}
    
```



Example: uncontended swap

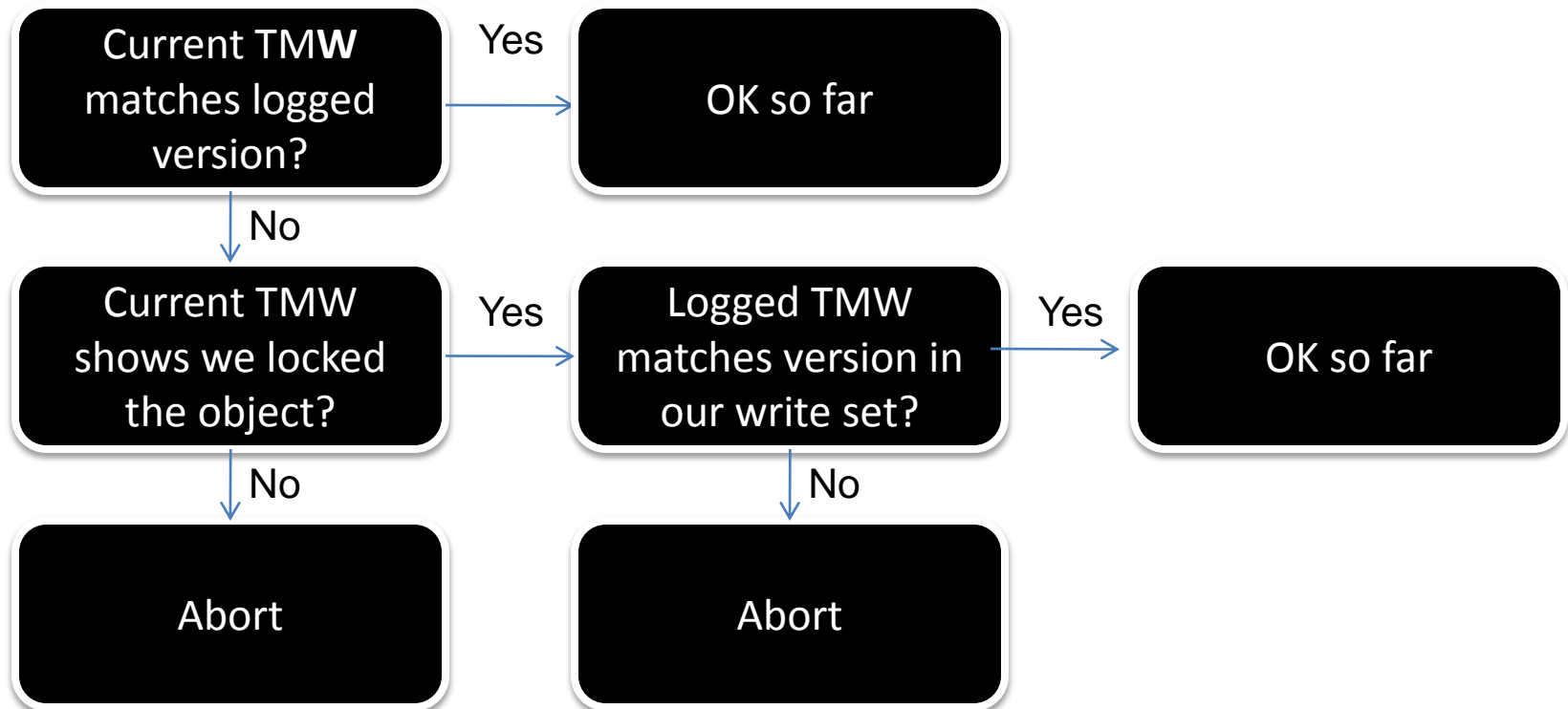


```
void Swap(int *a, int *b)
{
  do {
    tx = StartTx();
    va = ReadTx(tx, &a);
    vb = ReadTx(tx, &b);
    writeTx(tx, &a, vb);
    writeTx(tx, &b, va);
  } while (!CommitTx());
}
```

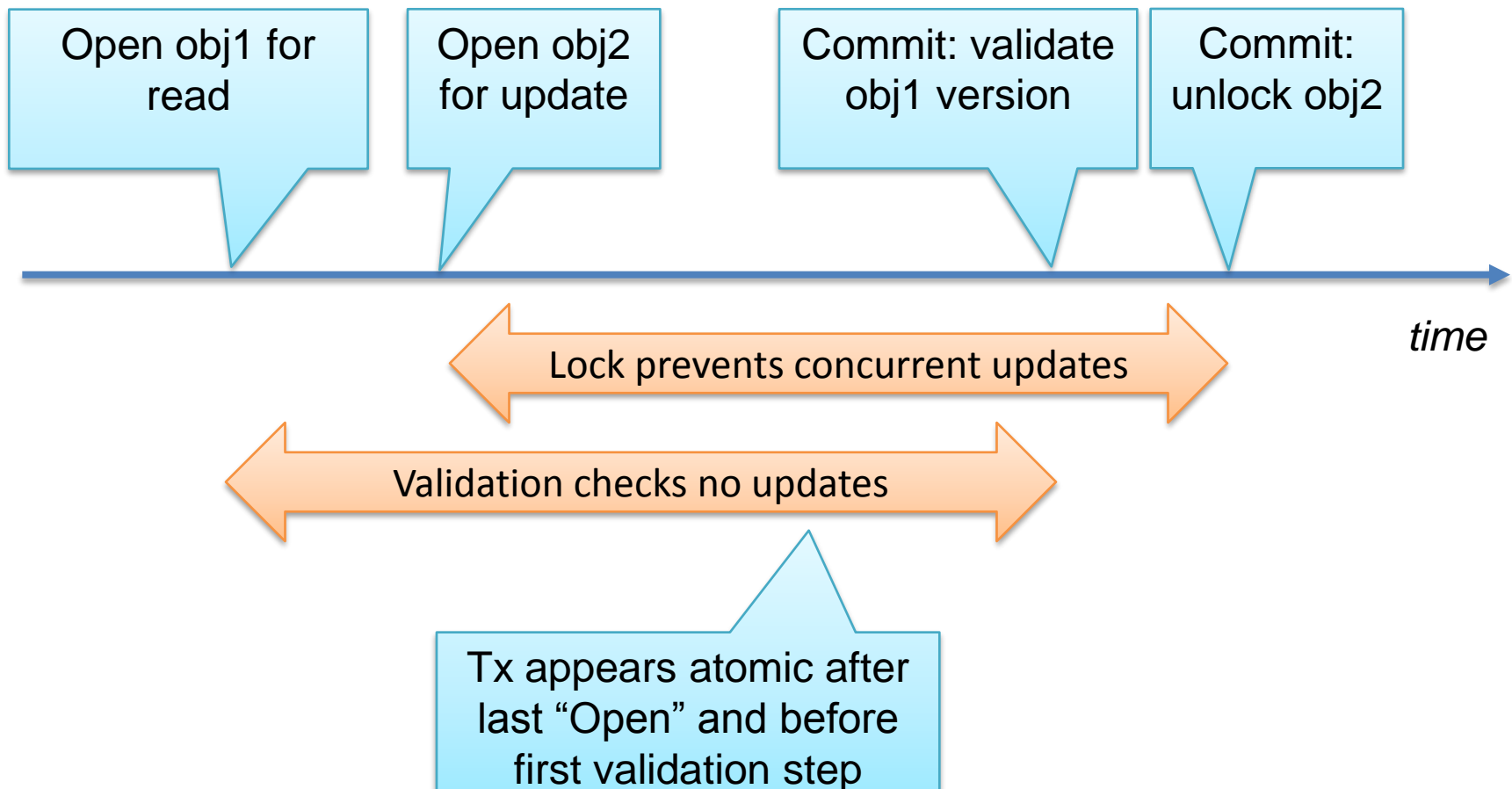


Commit in Bartok-STM

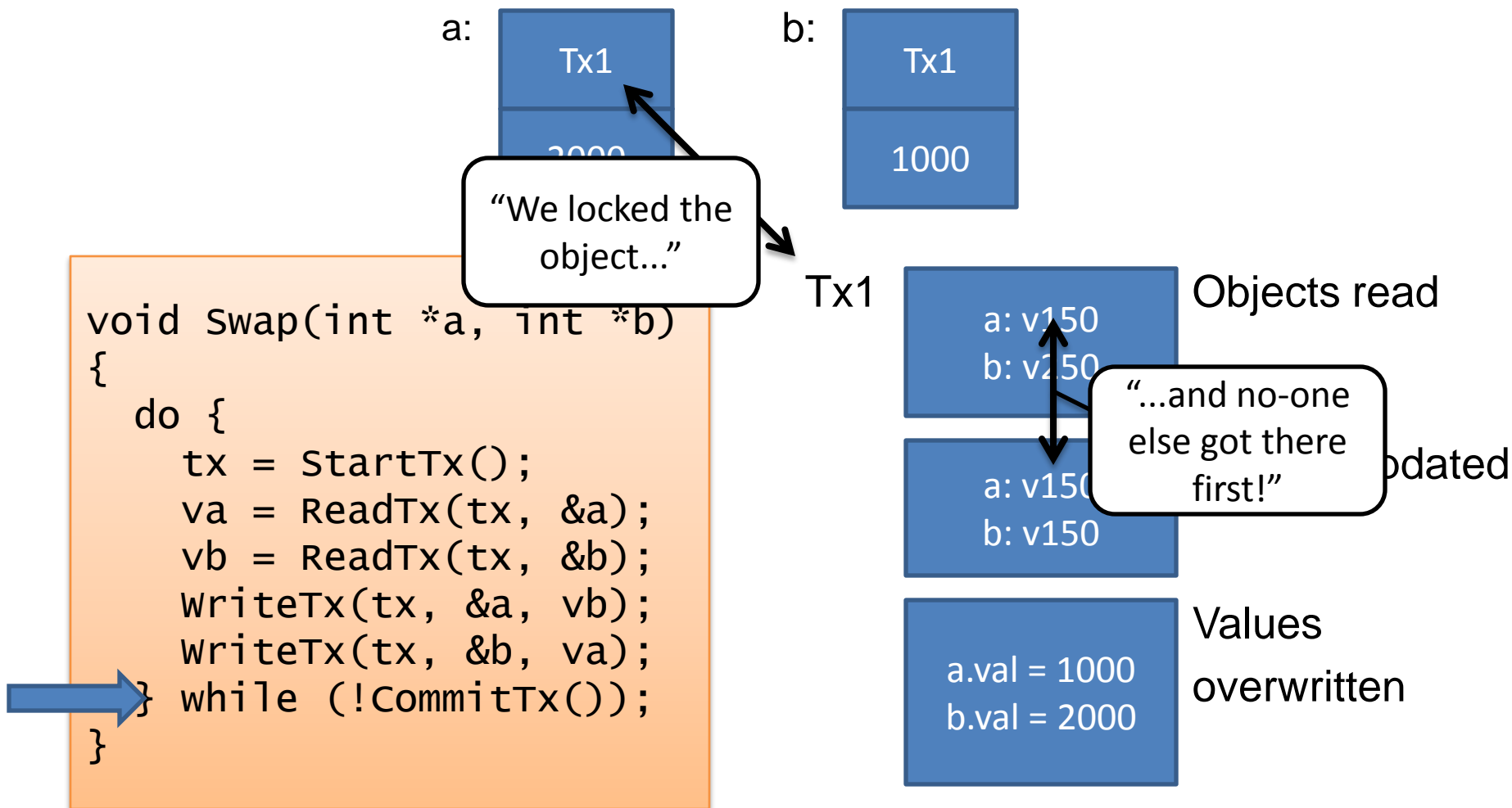
Iterate over
the read set:



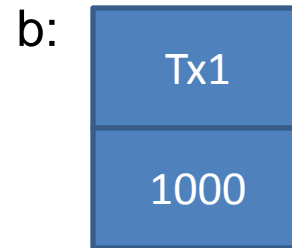
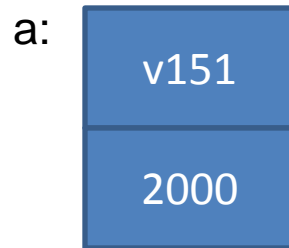
Correctness sketch



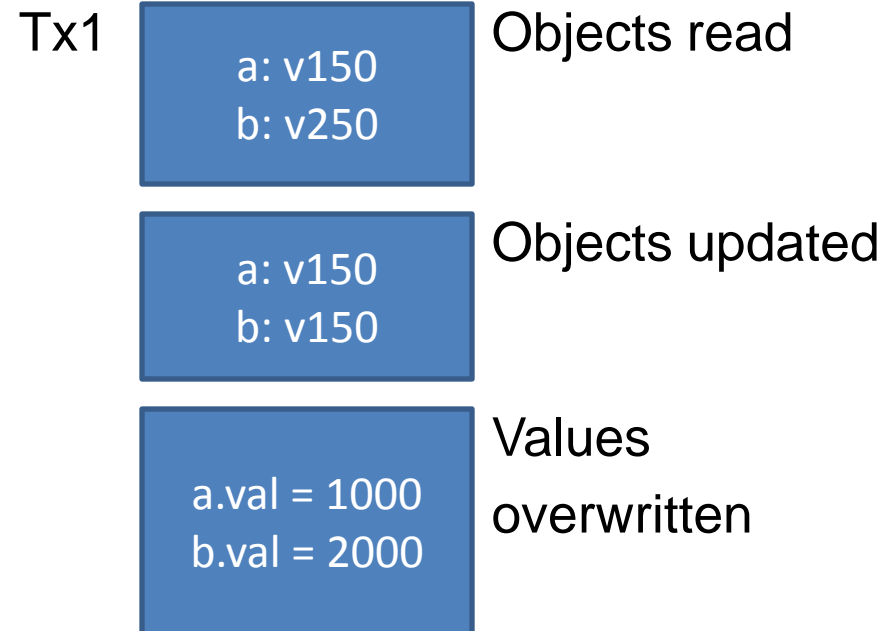
Example: uncontended swap



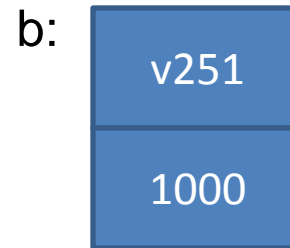
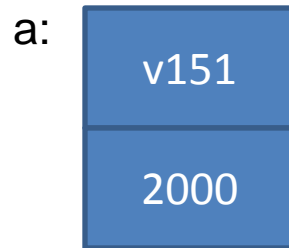
Example: uncontended swap



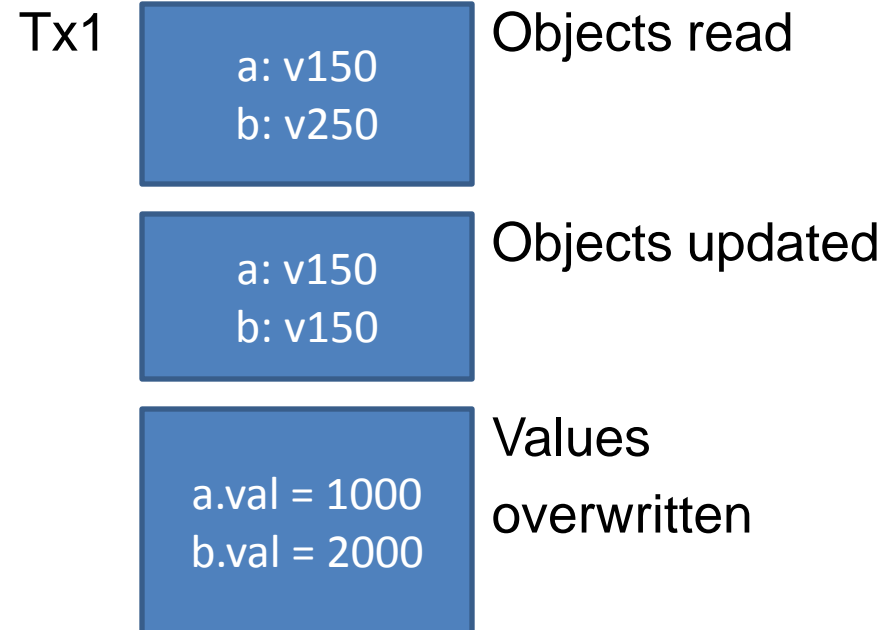
```
void Swap(int *a, int *b)
{
  do {
    tx = StartTx();
    va = ReadTx(tx, &a);
    vb = ReadTx(tx, &b);
    writeTx(tx, &a, vb);
    writeTx(tx, &b, va);
  } while (!CommitTx());
}
```



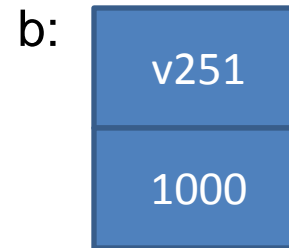
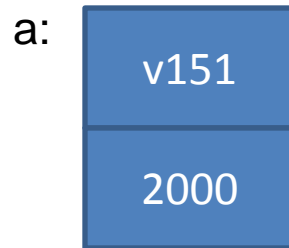
Example: uncontended swap




```
void Swap(int *a, int *b)
{
  do {
    tx = StartTx();
    va = ReadTx(tx, &a);
    vb = ReadTx(tx, &b);
    writeTx(tx, &a, vb);
    writeTx(tx, &b, va);
  } while (!CommitTx());
}
```



Example: uncontended swap



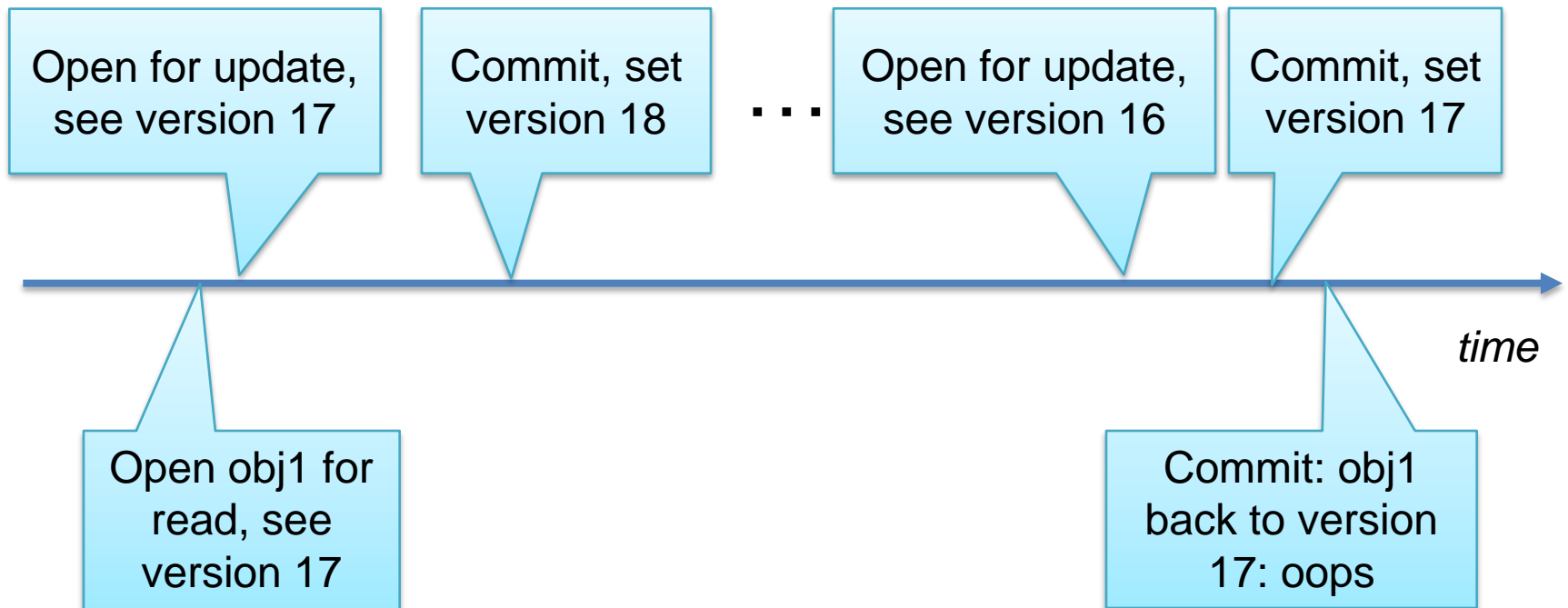
```
void Swap(int *a, int *b)
{
  do {
    tx = StartTx();
    va = ReadTx(tx, &a);
    vb = ReadTx(tx, &b);
    writeTx(tx, &a, vb);
    writeTx(tx, &b, va);
  } while (!CommitTx());
```



Tx-tx interaction in Bartok-STM

- Read-read: no problem, both readers see the same version number and verify it at commit time
- Read-write: reader sees that the writer has the object locked. Reader always defers to writer
- Write-write: competition for lock serializes writers (drop locks, then spin to avoid deadlock)

Version overflow in Bartok-STM



- At best this is distasteful; at worst incorrect

Avoiding version overflow

- A single object cycled through the whole version number space during a single tx
- Each tx can increment an object's version number at most once
- Bound the number of concurrent tx that can execute during a single tx
 - If a tx is about to observe this bound being reached then it just needs to validate itself

Bartok-STM

- Designed to work well on low-contention workloads
 - Eager version management to reduce commit costs
 - Eager locking to support eager version management
- Primitives do not guarantee that transactions see a consistent view of the heap while running
 - Can be sandboxed in managed code...
 - ...harder in native code

TL2

- Global “clock” incremented upon every commit
- Tx log the global clock when they start – “read version”, rv
- Tx perform lazy version management
- On reads, check the object’s timestamp $\leq rv$
- At commit:
 1. lock tentatively-updated locations
 2. increment clock (“write version”, wv)
 3. validate reads again (none older than our rv number)
 4. write back updates & store wv into updated objects

Example: uncontended swap

500

a:

v450
1000

b:

v350
2000

```
void Swap(int *a, int *b)
{
do {
    tx = StartTx();
    va = ReadTx(tx, &a);
    vb = ReadTx(tx, &b);
    writeTx(tx, &a, vb);
    writeTx(tx, &b, va);
} while (!CommitTx());
}
```

Example: uncontended swap

500

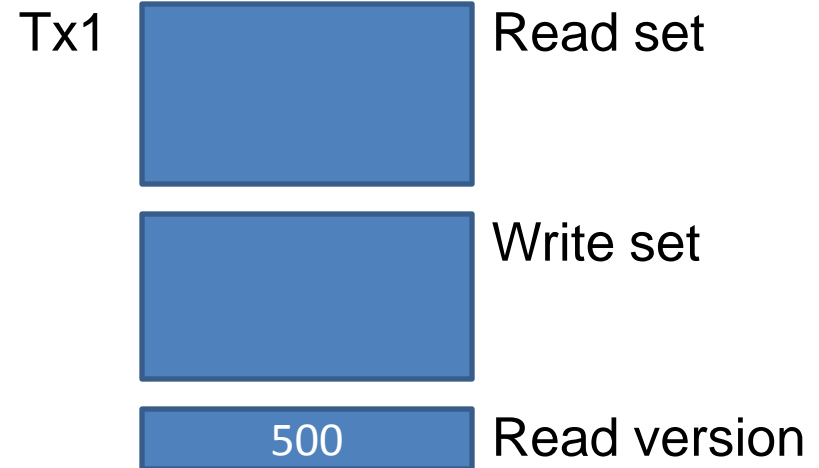
a:

v450
1000

b:

v350
2000

```
void Swap(int *a, int *b)
{
  do {
    tx = StartTx();
    va = ReadTx(tx, &a);
    vb = ReadTx(tx, &b);
    writeTx(tx, &a, vb);
    writeTx(tx, &b, va);
  } while (!CommitTx());
}
```



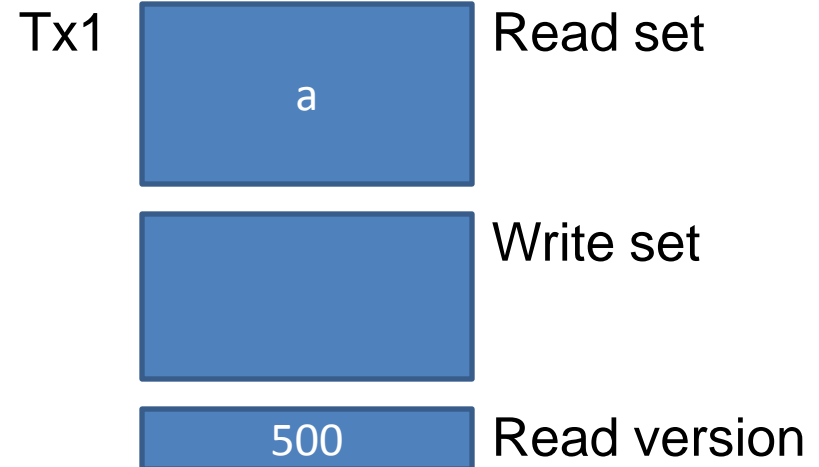
Example: uncontended swap

500

a:
v450
1000

b:
v350
2000

```
void Swap(int *a, int *b)
{
  do {
    tx = StartTx();
    va = ReadTx(tx, &a);
    vb = ReadTx(tx, &b);
    writeTx(tx, &a, vb);
    writeTx(tx, &b, va);
  } while (!CommitTx());
}
```



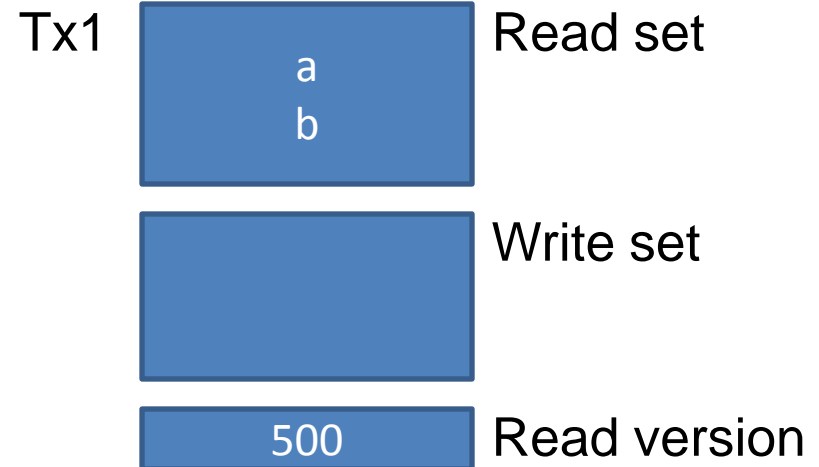
Example: uncontended swap

500

a:
v450
1000

b:
v350
2000

```
void Swap(int *a, int *b)
{
  do {
    tx = StartTx();
    va = ReadTx(tx, &a);
    vb = ReadTx(tx, &b);
    writeTx(tx, &a, vb);
    writeTx(tx, &b, va);
  } while (!CommitTx());
}
```



Example: uncontended swap

500

a:
v450
1000

b:
v350
2000

```
void Swap(int *a, int *b)
{
  do {
    tx = StartTx();
    va = ReadTx(tx, &a);
    vb = ReadTx(tx, &b);
    writeTx(tx, &a, vb);
    writeTx(tx, &b, va);
  } while (!CommitTx());
}
```

Tx1

a b	Read set
a.val = 2000	Write set
500	Read version

Example: uncontended swap

500

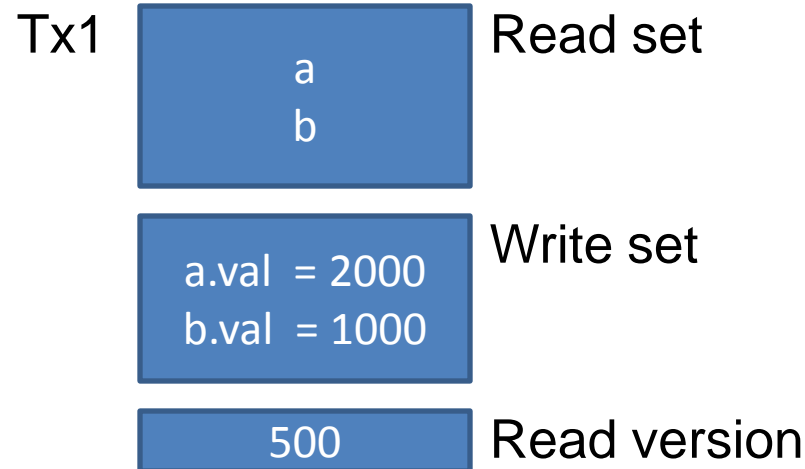
a:

v450
1000

b:

v350
2000

```
void Swap(int *a, int *b)
{
  do {
    tx = StartTx();
    va = ReadTx(tx, &a);
    vb = ReadTx(tx, &b);
    writeTx(tx, &a, vb);
    writeTx(tx, &b, va);
  } while (!CommitTx());
}
```



Example: uncontended swap

600

a: Tx1,
v450

1000

b: v350

2000

```
void swap(int *a, int *b)
{
  do {
    tx = StartTx();
    va = ReadTx(tx, &a);
    vb = ReadTx(tx, &b);
    writeTx(tx, &a, vb);
    writeTx(tx, &b, va);
  } while (!CommitTx());
}
```

Tx1

a b	Read set
a.val = 2000 b.val = 1000	Write set
500	Read version

Example: uncontended swap

600

a: Tx1,
v450

1000

b: Tx1,
v350

2000

```
void swap(int *a, int *b)
{
  do {
    tx = StartTx();
    va = ReadTx(tx, &a);
    vb = ReadTx(tx, &b);
    writeTx(tx, &a, vb);
    writeTx(tx, &b, va);
  } while (!CommitTx());
}
```

Tx1

a b	Read set
a.val = 2000 b.val = 1000	Write set
500	Read version

Example: uncontended swap

601

a: Tx1,
v450

1000

b: Tx1,
v350

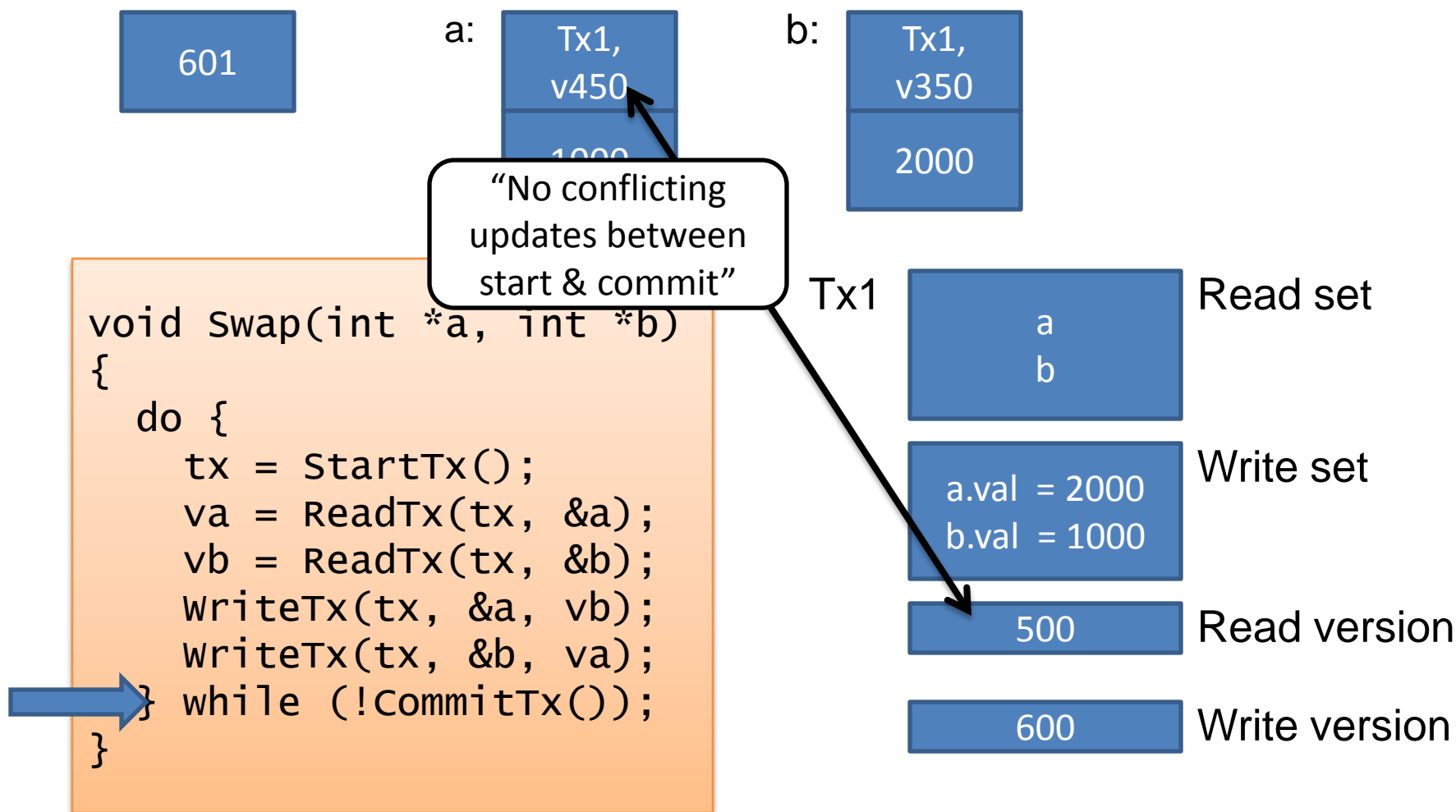
2000

```
void swap(int *a, int *b)
{
  do {
    tx = StartTx();
    va = ReadTx(tx, &a);
    vb = ReadTx(tx, &b);
    writeTx(tx, &a, vb);
    writeTx(tx, &b, va);
  } while (!CommitTx());
}
```

Tx1

a b	Read set
a.val = 2000 b.val = 1000	Write set
500	Read version
600	Write version

Example: uncontended swap



Example: uncontended swap

601

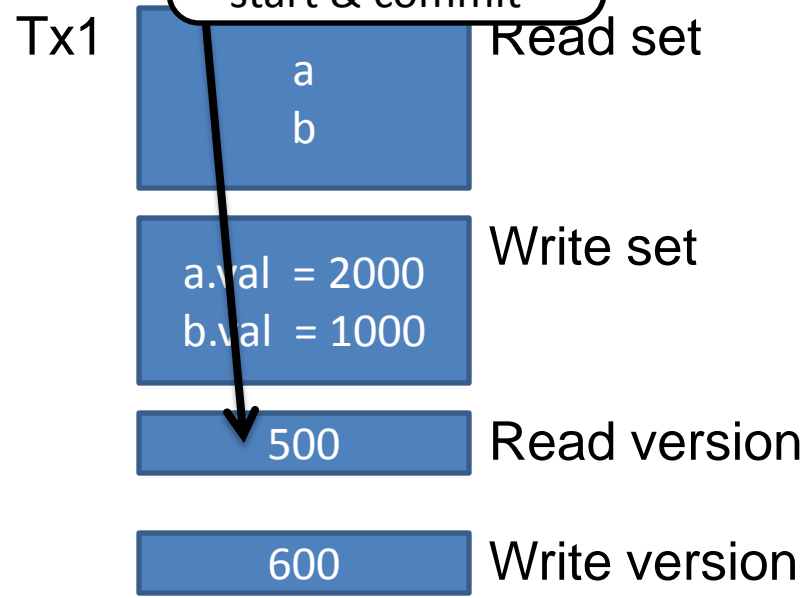
a: Tx1,
v450

1000

b: Tx1,
v350

2000

“No conflicting updates between start & commit”



```
void Swap(int *a, int *b)
{
  do {
    tx = StartTx();
    va = ReadTx(tx, &a);
    vb = ReadTx(tx, &b);
    writeTx(tx, &a, vb);
    writeTx(tx, &b, va);
  } while (!CommitTx());
}
```



Example: uncontended swap

601

a: Tx1,
v450
2000

b: Tx1,
v350
2000

```
void swap(int *a, int *b)
{
  do {
    tx = StartTx();
    va = ReadTx(tx, &a);
    vb = ReadTx(tx, &b);
    writeTx(tx, &a, vb);
    writeTx(tx, &b, va);
  } while (!CommitTx());
}
```

Tx1

a b	Read set
a.val = 2000 b.val = 1000	Write set
500	Read version
600	Write version

Example: uncontended swap

601

a: Tx1,
v450
2000

b: Tx1,
v350
1000

```
void swap(int *a, int *b)
{
  do {
    tx = StartTx();
    va = ReadTx(tx, &a);
    vb = ReadTx(tx, &b);
    writeTx(tx, &a, vb);
    writeTx(tx, &b, va);
  } while (!CommitTx());
}
```

Tx1

a b	Read set
a.val = 2000 b.val = 1000	Write set
500	Read version
600	Write version

Example: uncontended swap

601

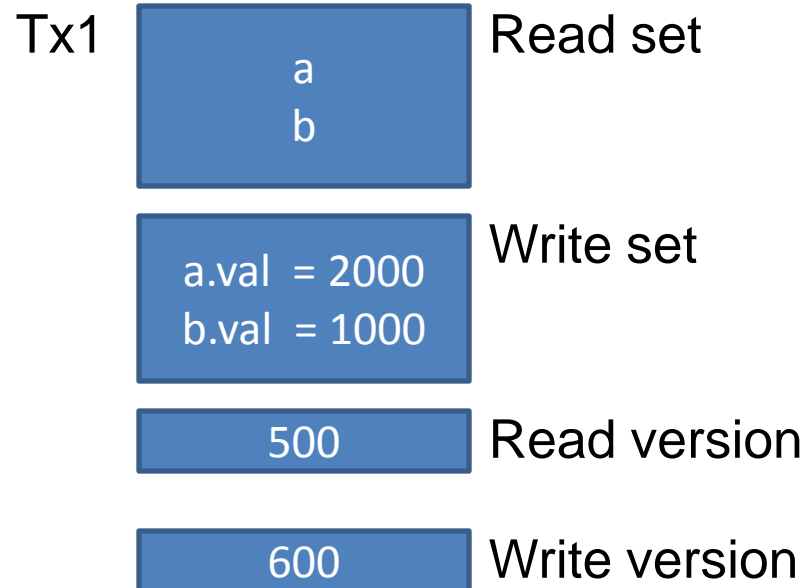
a:

v600
2000

b:

v600
1000

```
void swap(int *a, int *b)
{
  do {
    tx = StartTx();
    va = ReadTx(tx, &a);
    vb = ReadTx(tx, &b);
    writeTx(tx, &a, vb);
    writeTx(tx, &b, va);
  } while (!CommitTx());
}
```



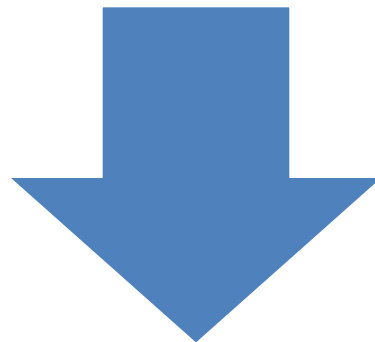
Tx-tx interaction in TL2

- Read-read: no problem, both readers see the same version number and verify it at commit time
- Reader-tentative-write: commit-time locking means that the reader can commit if they finish before the writer
- Reader-committed-writer: reader aborts
- Write-write: competition for lock serializes writers at commit time

Low-level optimizations in TL2

- Accelerate look-aside into log:
 - Use Bloom filter to check if the location may be in the write set
 - Can use hashing mechanisms to reduce log searching
- Don't need the read set in read-only transactions
- Don't need to validate if $\text{write_version} = \text{read_version} + 1$
- Reduce contention on the global version number:
 - Combine all version numbers with a thread ID of the last modifier
 - Only increment global version number if it differs from our thread's last write-version
 - Otherwise the global version number and our thread ID are globally unique

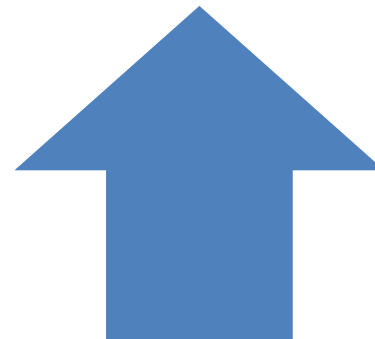
Performance trade-offs & dangers



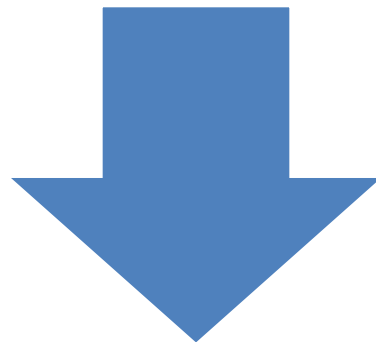
Improve concurrency between transactions (e.g. detect that an execution is linearizable, even though conflicting transactions overlapped in time)



Improve the speed of the STM primitives, e.g. by performing less book-keeping



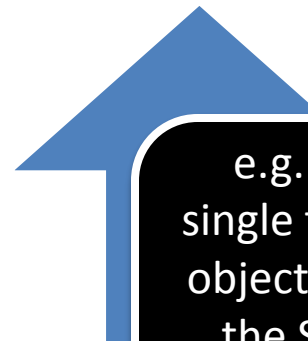
Performance trade-offs & dangers



Improve concurrency between transactions (e.g. detect that an execution is linearizable, even though conflicting transactions overlapped in time)



Improve the speed of the STM primitives, e.g. by performing less book-keeping



e.g. Bartok-STM allows a single tentative writer to each object: is the simplification in the STM worth the loss of concurrency between transactions?

Summary

- Common design features:
 - Use locking to arbitrate between writers (e.g. compare with the complexity of early non-blocking commits) and version numbers to detect conflicts on reads
 - Correctness arguments can both be based on identifying a point at which a transaction appears atomic
- TL2's use of a global version number gives stronger guarantees to the programmer building over it
- Many implementation variants possible, e.g. addr-to-TMW via a hash function, writers linked from an object header, indirection to object versions

Further reading

- “McRT-STM: a high performance software transactional memory system for a multi-core runtime”, Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L Hudson, Chi Cao Minh, Ben Hertzberg. PPOPP 2006. *Introduced McRT-STM and explored many design choices.*
- “Optimizing memory transactions”, Tim Harris, Mark Plesko, Avraham Shinnar, David Tarditi. PLDI 2006. *Bartok-STM.*
- “Transactional Locking II”, Dave Dice, Ori Shalev, Nir Shavit. DISC 2006. *TL2.*
- “Time-based transactional memory with scalable time bases”, Torvald Riegel, Christof Fetzer, Pascal Felber. SPAA 2007. *Supporting more concurrency between transactions by not having a fixed linearization point within them.*
- “NZTM: nonblocking zero-indirection transactional memory”, Fuad Tabba, Cong “James” Wang, James R Goodman, Mark Moir. TRANSACT 2007. *Explores maintaining lock-free progress while still providing good straight-line performance.*

Lecture 3

Building “atomic” over “STM”

Example semantics

- Atomic blocks appear to execute exactly once
- Atomic blocks run atomically wrt other atomic blocks and normal code (“strong atomicity”)
- Exceptions propagate normally out of an atomic block (erasure semantics; informally “atomic X” == “X” in single-threaded code)
- No access to native code

```
void Swap(Pair p) {  
    atomic {  
        va = p.a;  
        vb = p.b;  
        p.a = vb;  
        p.b = va;  
    }  
}
```

Why these semantics?

- Many of these are engineering decisions
 - I'm aiming to keep the definition simple
 - I'm aiming to provide a model that allows many implementations
 - I'm focussing on shared-memory-data-structure scenarios rather than atomic blocks for failure atomicity or to group I/O
- Different choices may be better in other settings
 - E.g. Based on programmer skills, other language features, possible implementation complexity,...
- Different choices may prove to be better as we gain experience using atomic blocks

Compilation

Source to bytecode compiler;
typically “csc” in C#, “javac” for Java

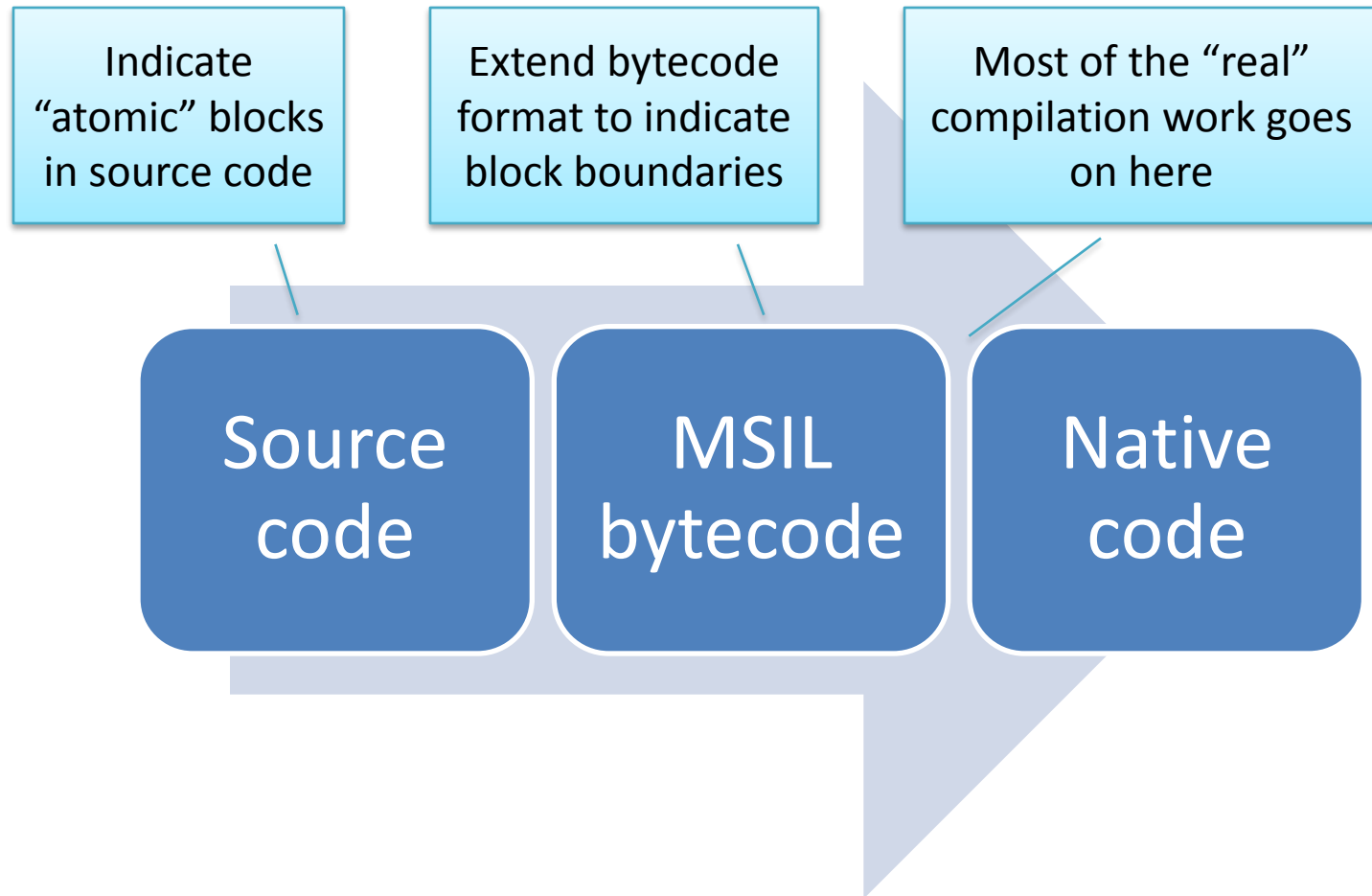
Bytecode-to-native compiler;
JIT or traditional compilation

Source
code

MSIL
bytecode

Native
code

Compilation



Why divide things this way?

- Little information loss from source code to bytecode
- Source-to-bytecode works a file at a time, bytecode-to-native can see the whole program (or, at least, see all of the parts needed so far in execution)
- Lower level transformations possible at bytecode-to-native
- Integration between the STM and other parts of the runtime system

```
void Swap(Pair p) {  
    try {  
        va = p.a;  
        vb = p.b;  
        p.a = vb;  
        p.b = va;  
    } catch (AtomicException) {  
    }  
}
```


Boilerplate around transactions

```
void swap(Pair p) {  
  do {  
    done = true;  
    try {  
      try {  
        tx = StartTx();  
        va = p.a;  
        vb = p.b;  
        p.a = vb;  
        p.b = va;  
      } finally {  
        CommitTx();  
      }  
    } catch (TxInvalid) {  
      done = false;  
    }  
  } while (!done);  
}
```

Keep running the atomic block in a fresh tx each time

Commit (on normal or exn exit)

Commit fails by raising a TxInvalid exception; re-execute

(I'm using source code examples for clarity; in reality this would be in the compiler's internal intermediate code)

Naïve expansion of data accesses

```
void swap(Pair p) {
  do {
    done = true;
    try {
      try {
        tx = StartTx();
        TxWrite(tx, &va, TxRead(tx, &p.a));
        TxWrite(tx, &vb, TxRead(tx, &p.b));
        TxWrite(tx, &p.a, TxRead(tx, &vb));
        TxWrite(tx, &p.b, TxRead(tx, &va));
      } finally {
        CommitTx();
      }
    } catch (TxInvalid) {
      done = false;
    }
  } while (!done);
}
```

What are the problems here?

- Using the STM for thread-private local variables
- Repeatedly mapping from addresses to concurrency control info
- Duplicating concurrency control work if it's implemented at a per-object granularity

Decomposed STM primitive API

- `OpenForRead(tx, obj)`
- `OpenForRead(tx, addr)`
- `OpenForUpdate(tx, obj)`
- `OpenForUpdate(tx, addr)`

- `LogForUndo(tx, addr)`

Indicate intent to read from an object or from a given address

Indicate intent to update a specific address (& optional size)

Using the decomposed API

```
x = p.a;
```



```
OpenForRead(tx, p);  
x = p.a;
```

```
p.b = y;
```



```
OpenForUpdate(tx, p);  
LogForUndo(tx, &p.b);  
p.b = y;
```

Implementation using decomposed API

```
...  
OpenForUpdate(tx, p);  
OpenForRead(tx, p);  
va = p.a;  
OpenForRead(tx, p);  
vb = p.b;  
OpenForUpdate(tx, p);  
LogForUndo(tx, &p.a);  
p.a = vb;  
OpenForUpdate(tx, p);  
LogForUndo(tx, &p.b);  
p.b = va;  
...
```

Always need update access:
get it first

Second OpenForRead made
unnecessary by first

Second OpenForUpdate
made unnecessary by first

Improved expansion of data accesses

```
void Swap(Pair p) {
  do {
    done = true;
    try {
      try {
        tx = StartTx();
        OpenForUpdate(tx, p);
        va = p.a;
        vb = p.b;
        LogForUndo(tx, &p.a);
        p.a = vb;
        LogForUndo(tx, &p.b);
        p.b = va;
      } finally {
        CommitTx();
      }
    } catch (TxInvalid) {
      done = false;
    }
  } while (!done);
}
```

Are we done?

- Local variables
- By-ref parameters
- Method calls
- Sandboxing invalid transactions
- Keeping optimizations safe
- GC integration
- Finalizers
- Condition synchronization

Local variables

- Won't see conflicts (in the safe subset of C#)
- May need to roll-back on re-execution
- Explicitly snapshot on entry to an atomic block, explicitly roll-back
- Optimization: just do this with locals live on entry to the atomic block

Method calls

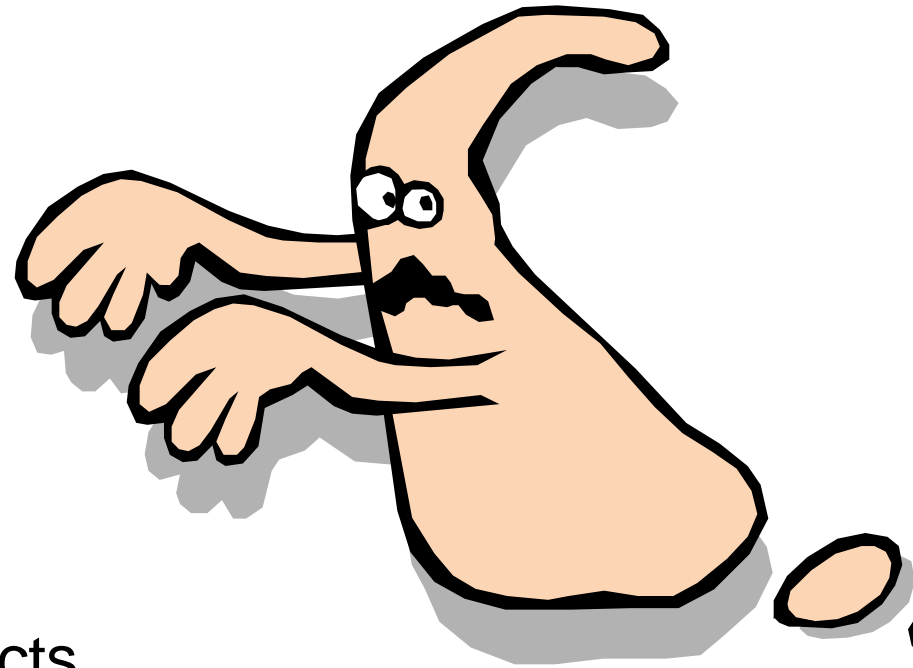
- Build transacted versions of methods called within atomic blocks
- Implemented in Bartok-STM by a special suffix on the name and adjustments to the class hierarchy
- Alternatively JIT them on demand using a transacted vs normal method table on each object

By-ref paramters

- Target may be:
 - Local variable
 - Caller's variable
 - Heap object (possibly shared)
- Snapshot SP on start of atomic block, log writes to older frames
 - Why not just log them all?

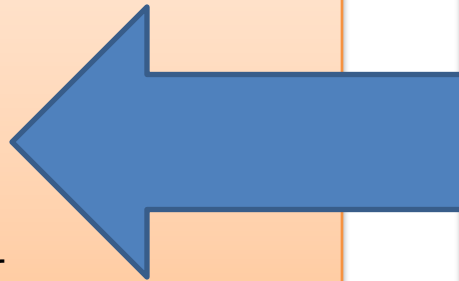
Sandboxing zombie transactions

- Those that have become invalid but don't yet know it
- May access memory
- May raise exceptions
- May attempt system calls etc
- General principle – validate before revealing any tx's effects outside the STM world



Looping / slow zombies

```
void Method1(Pair p) {  
    atomic {  
        ta = p.a;  
  
        tb = p.b;  
        if (ta != tb) {  
            while (true) {  
            }  
        }  
    }  
}
```



```
void Method2(Pair p) {  
    atomic {  
        p.a = 100;  
        p.b = 100;  
    }  
}
```

- Method2 runs between Method1's memory accesses
- The transaction running Method1 becomes a zombie... but never attempts to commit

Looping / slow zombies

- Add new API function “ValidationTick”
- ValidationTick guarantees: it will eventually detect if its calling transaction is invalid
- Call it in any loop not otherwise calling a TM API function
- Optimize ValidationTick so it only does “real” validation occasionally
- (Could also optimize the placement of ValidationTick calls)

```
OpenForRead(p);  
ta = p.a;  
tb = p.b;  
if (ta != tb) {  
    while (true) {  
        validationTick();  
    }  
}
```

Keeping optimizations safe

Original (contrived) source code

```
void Clear_tx(Pair p) {  
    for (int i = 0; i < 10; i ++) {  
        p.a = 10;  
        p.b = i;  
    }  
}
```

Keeping optimizations safe

Expanded with decomposed API operations

```
void Clear_tx(Pair p) {  
    for (int i = 0; i < 10; i ++) {  
        OpenForUpdate(tx, p);  
        LogForUndo(tx, &p.a);  
        p.a = 10;  
        LogForUndo(tx, &p.b);  
        p.b = i;  
    }  
}
```


Keeping optimizations safe

Hoisting loop-invariant code

```
void Clear_tx(Pair p) {  
    p.a = 10;  
    for (int i = 0; i < 10; i++) {  
        OpenForUpdate(tx, p);  
        LogForUndo(tx, &p.a);  
        LogForUndo(tx, &p.b);  
        p.b = i;  
    }  
}
```

Keeping optimizations safe

Introduce dependencies

```
void Clear_tx(Pair p) {  
  for (int i = 0; i < 10; i ++)  
    tmp1 = OpenForUpdate(tx, p);  
    tmp2 = LogForUndo(tx, &p.a) <tmp1>;  
    p.a = 10 <tmp2>;  
    tmp3 = LogForUndo(tx, &p.b) <tmp1>;  
    p.b = i <tmp3>;  
  }  
}
```

Keeping optimizations safe

Transformations must respect dependencies

```
void Clear_tx(Pair p) {  
    tmp1 = OpenForUpdate(tx, p);  
    tmp2 = LogForUndo(tx, &p.a) <tmp1>;  
    tmp3 = LogForUndo(tx, &p.a) <tmp1>;  
    p.a = 10 <tmp2>;  
    for (int i = 0; i < 10; i++) {  
        p.b = i <tmp3>;  
    }  
}
```

Taming the logs

- We'd like the logs to grow with (at most) the volume of data the tx accesses
 - Not with the time that it runs for
- We must consider GC of temporaries (later today)
- We must avoid or recover from duplicates in the log

Avoiding duplication in Bartok-STM

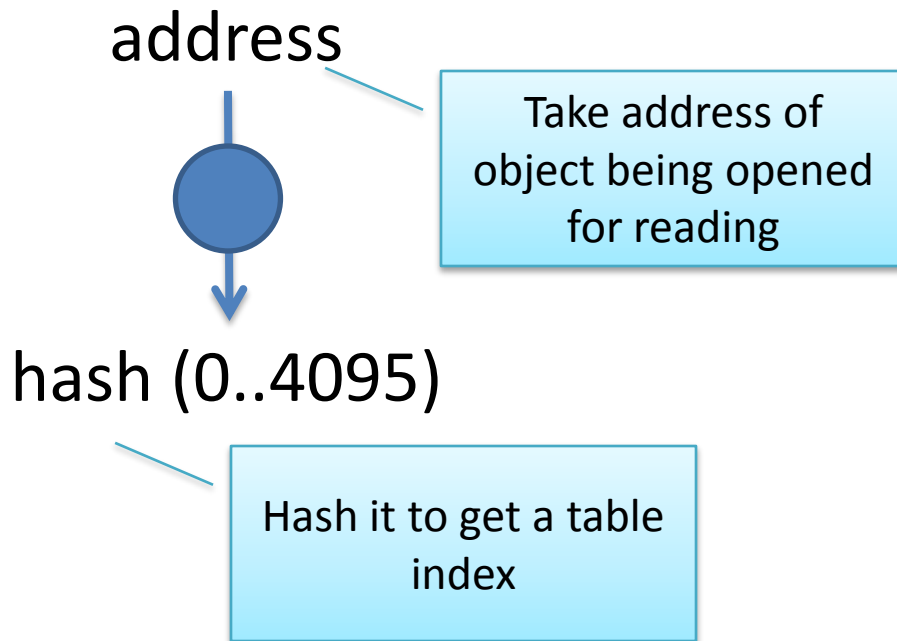
- Open-for-update log
 - Easy, updates are visible
- Undo log
 - Deterministically remove duplicates with a bit-map
 - Remember: updater has the object open for exclusive access
- Open-for-read log
 - Invisible reads
 - Probabilistically remove duplicates during tx
 - Deterministically remove the rest at GC

Removing read-log duplicates

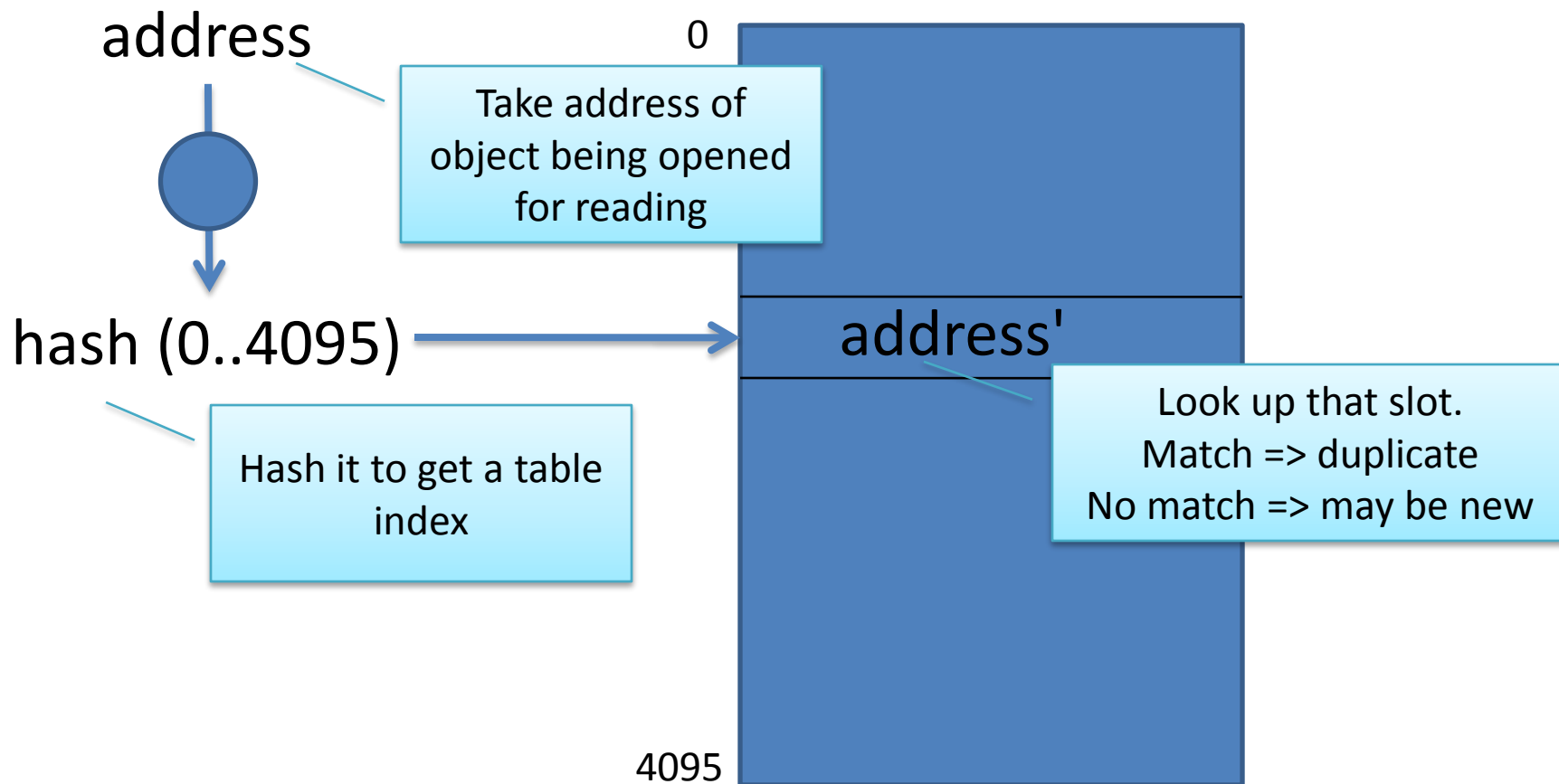
address

Take address of
object being opened
for reading

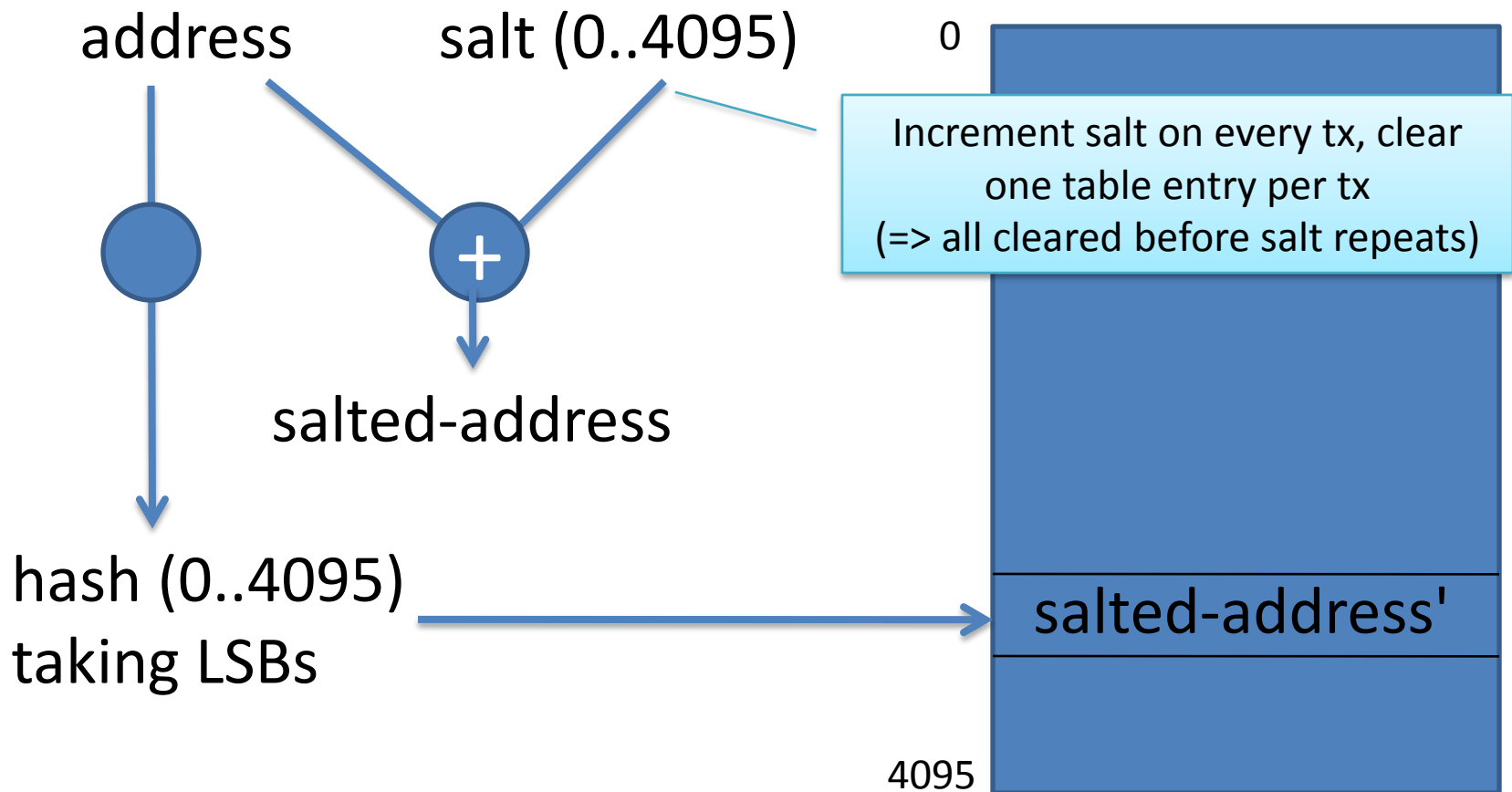
Removing read-log duplicates



Removing read-log duplicates



Incremental cleaning



GC integration

Another contrived program

```
void Temp() {  
    Pair result;  
    atomic {  
        for (int i = 0; i < 100000; i ++) {  
            result = new Pair();  
            result.a = i;  
        }  
    }  
    return result;  
}
```

Lots of temporary objects are allocated as the atomic block runs

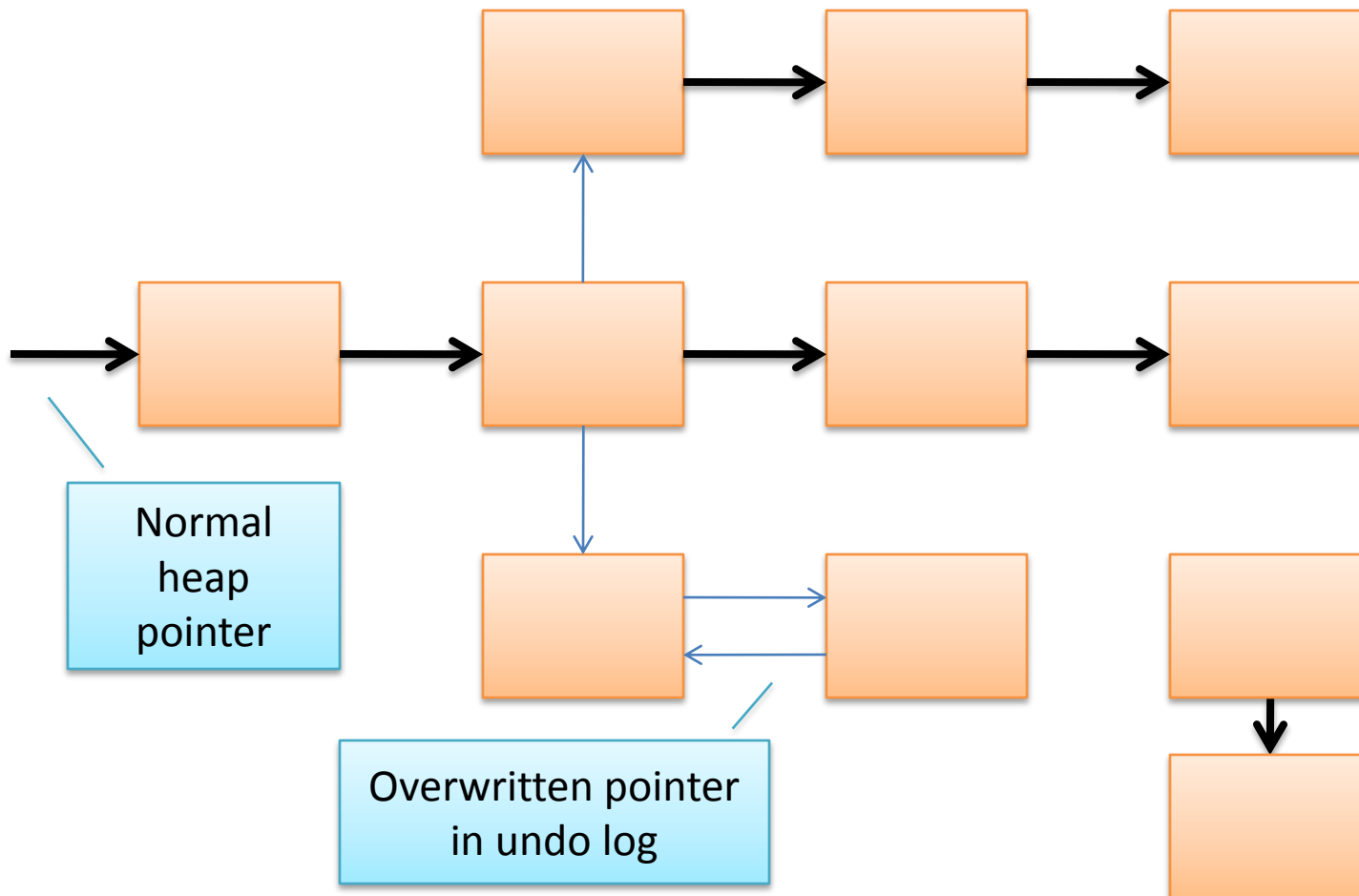
GC integration

- Abort all running tx on GC?
 - Not ideal: long running tx will not be able to commit
 - Is there a precedent for language features with this kind of perf?
 - (We also rely on GC as a fall-back for duplicate log elimination and to force the validations needed for safe version number overflow)
- Treat all the references from the logs as roots?
 - Not ideal: we'd keep all those temporaries

GC integration

- Principle:
 - Consider the possible heaps based on whether tx commit or abort
 - Retain an object if it is alive in any of these cases (ideally “iff”)
- Do we need to consider 2^n possibilities with n running tx?
 - No: validate all the tx first so we know they are not conflicting
 - Consider the world if they all commit, consider the world if they all roll back

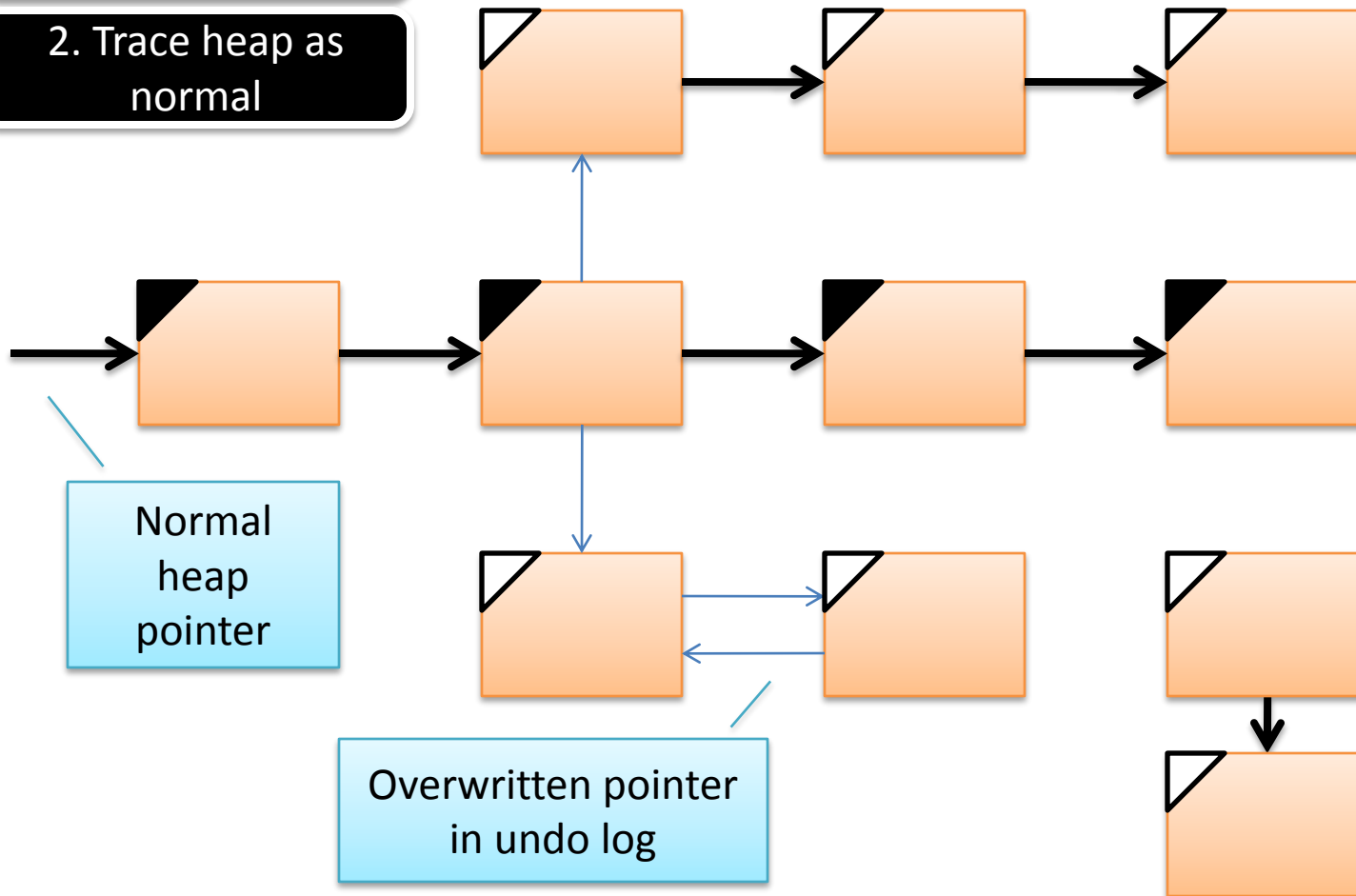
Example heap



Conservative algorithm

1. Validate tx

2. Trace heap as normal

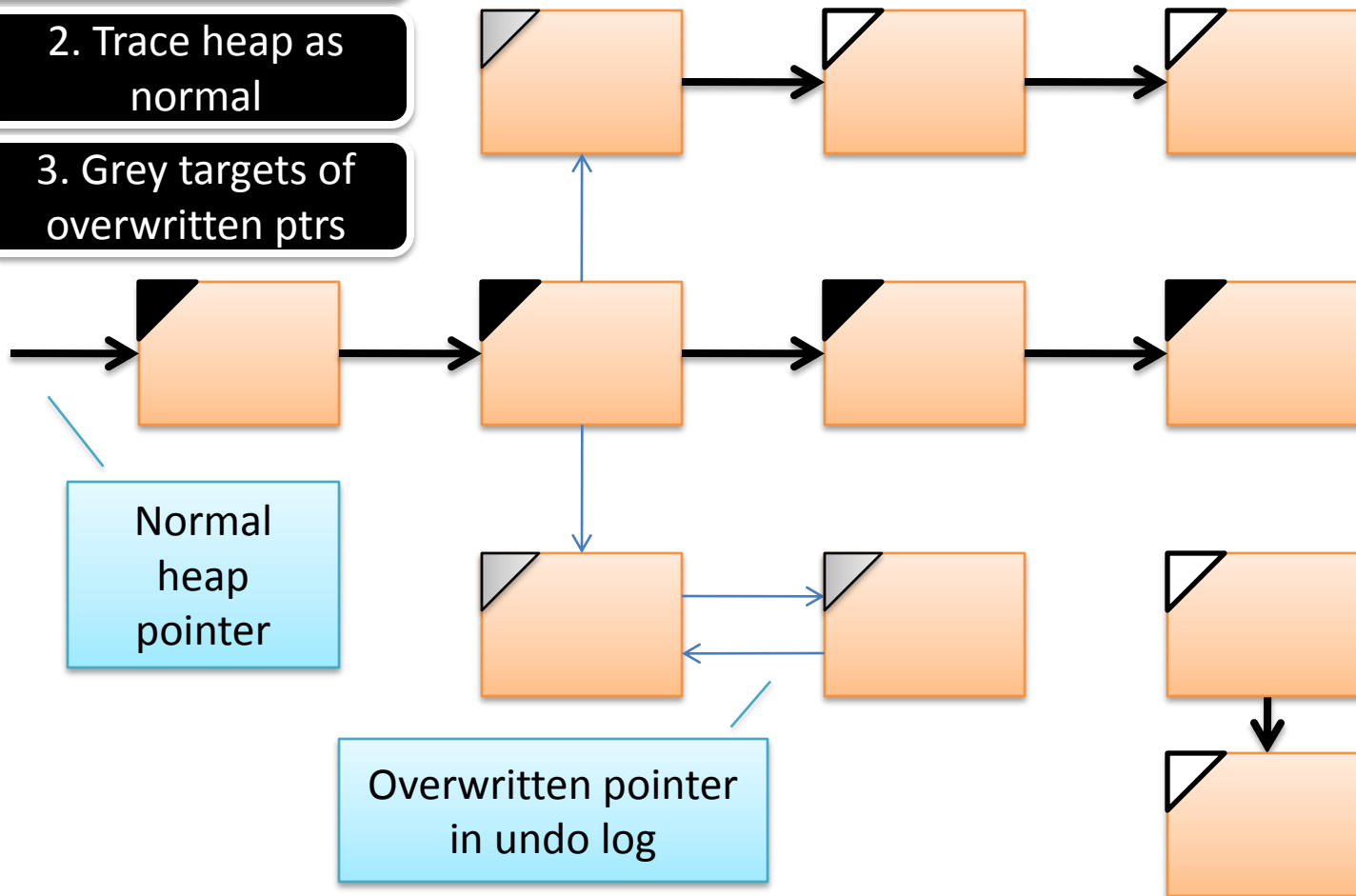


Conservative algorithm

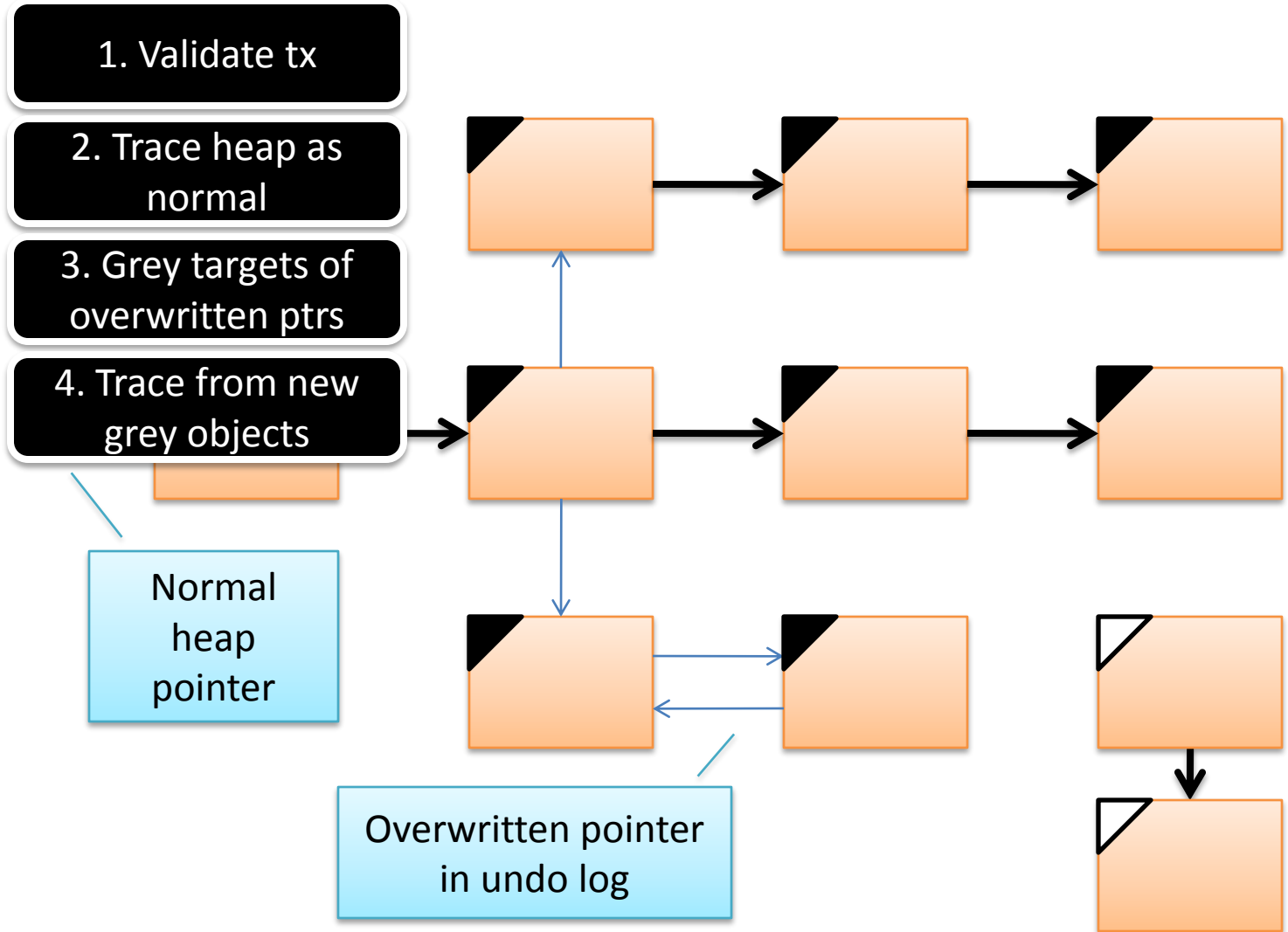
1. Validate tx

2. Trace heap as normal

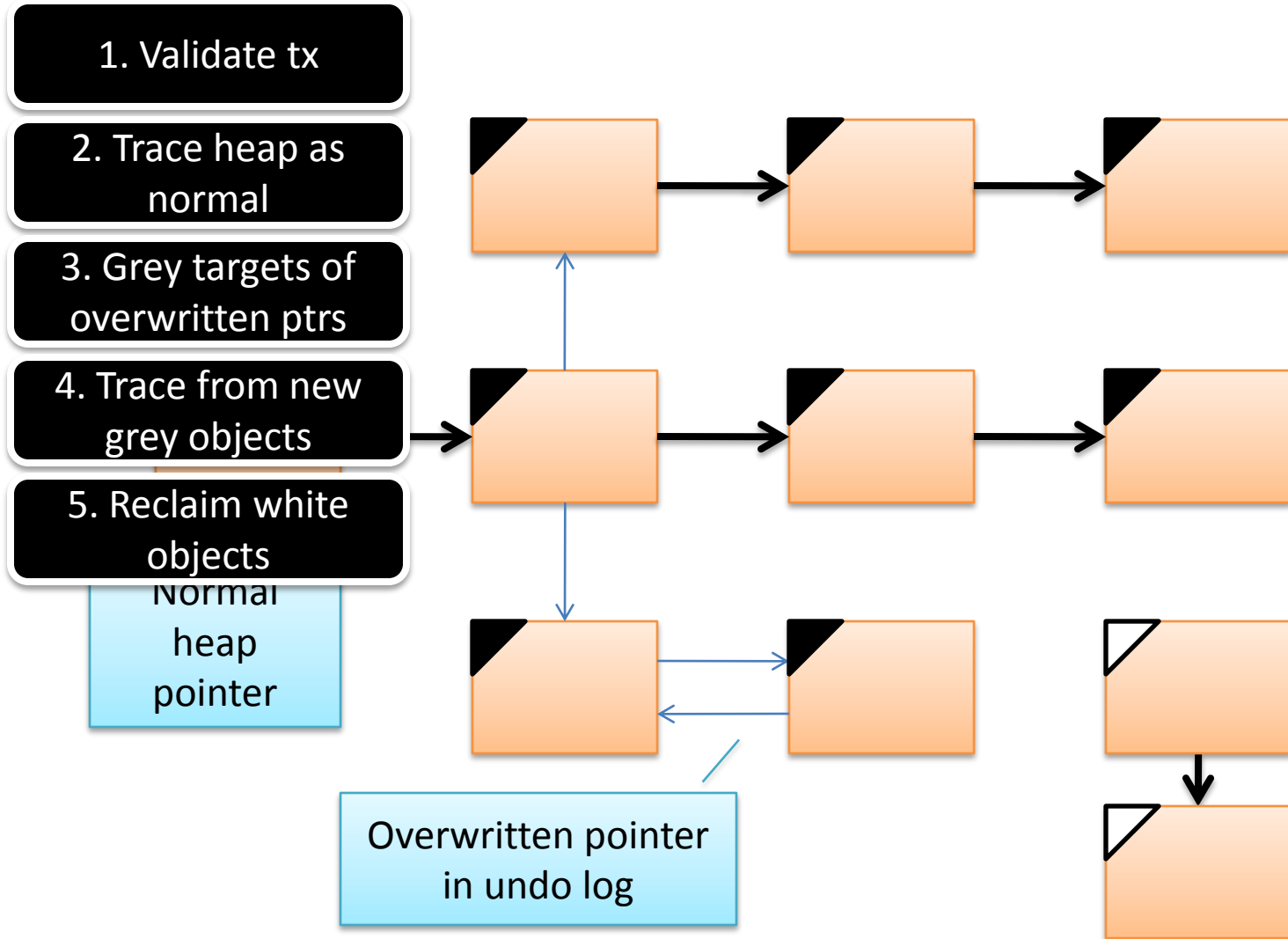
3. Grey targets of overwritten ptrs



Conservative algorithm



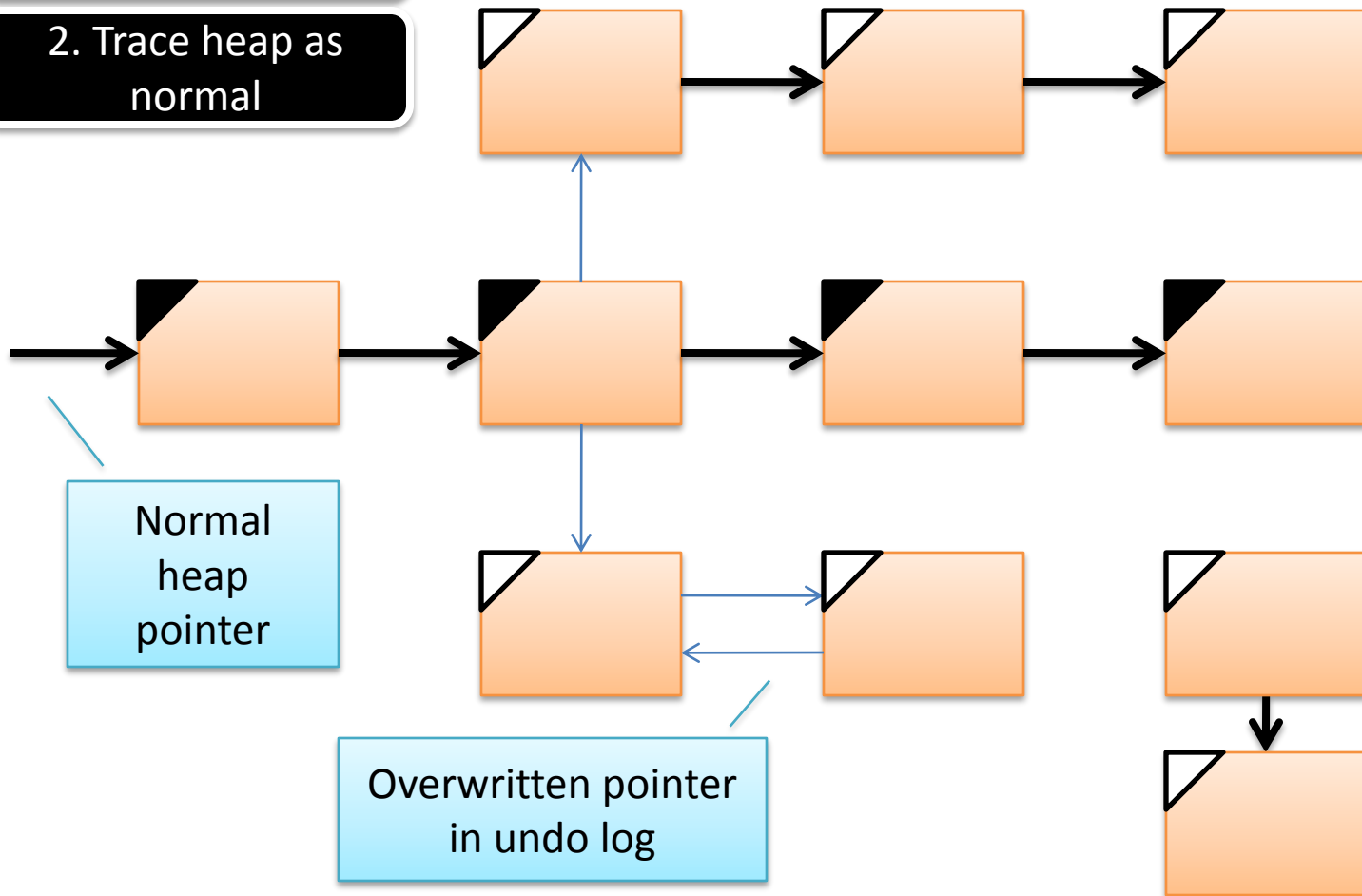
Conservative algorithm



Precise algorithm

1. Validate tx

2. Trace heap as normal

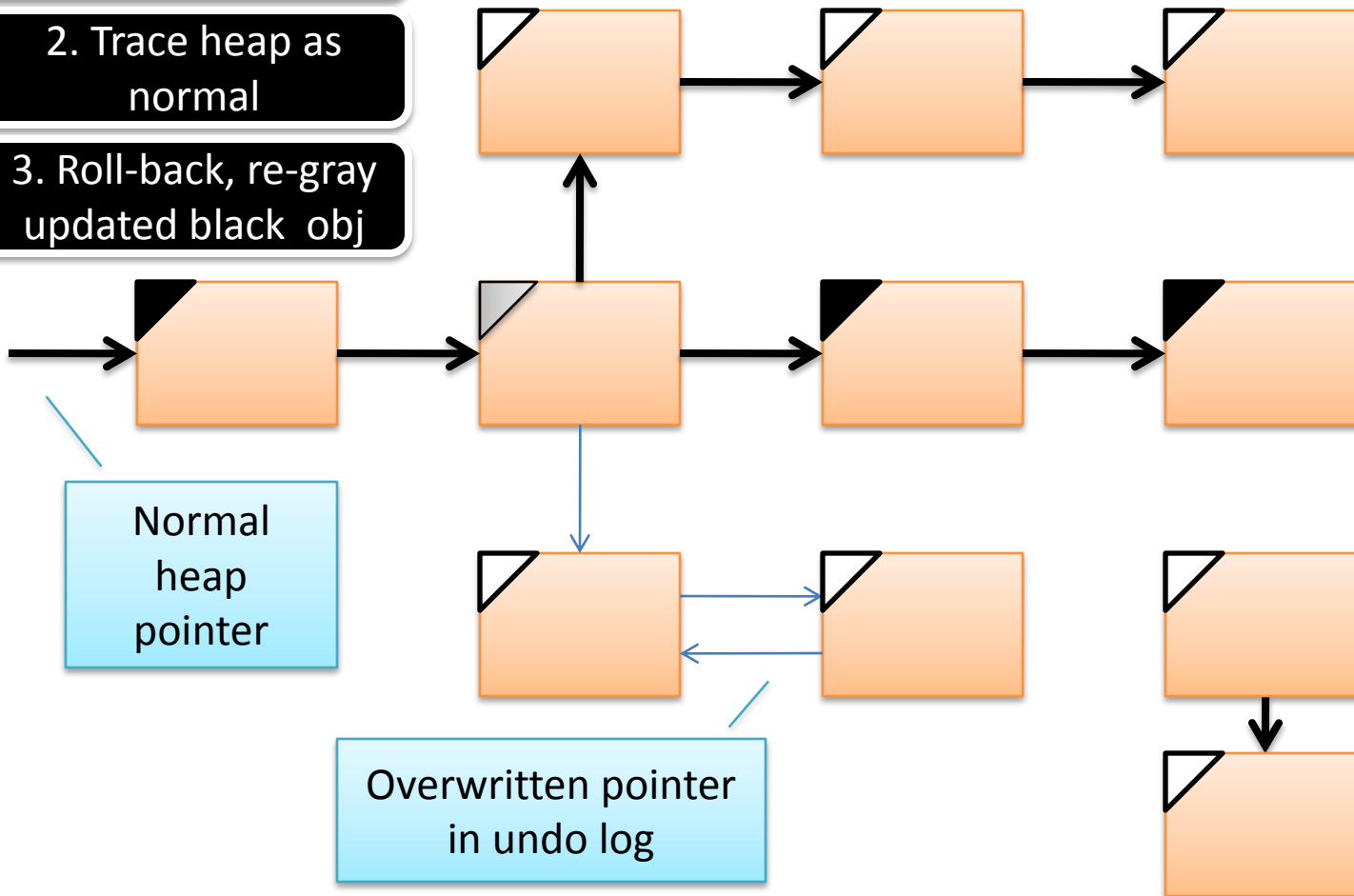


Precise algorithm

1. Validate tx

2. Trace heap as normal

3. Roll-back, re-gray updated black obj



Precise algorithm

1. Validate tx

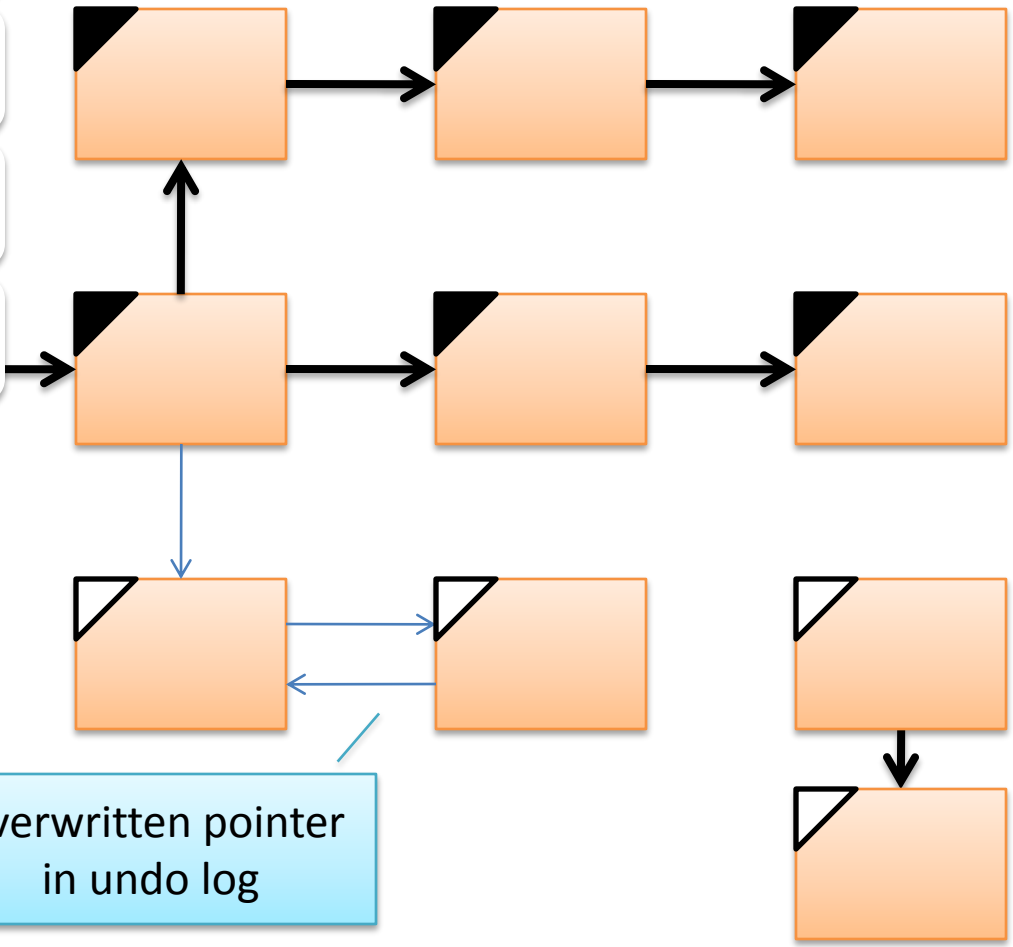
2. Trace heap as normal

3. Roll-back, re-gray updated black obj

4. Trace from gray objects

Normal heap pointer

Overwritten pointer in undo log



Precise algorithm

1. Validate tx

2. Trace heap as normal

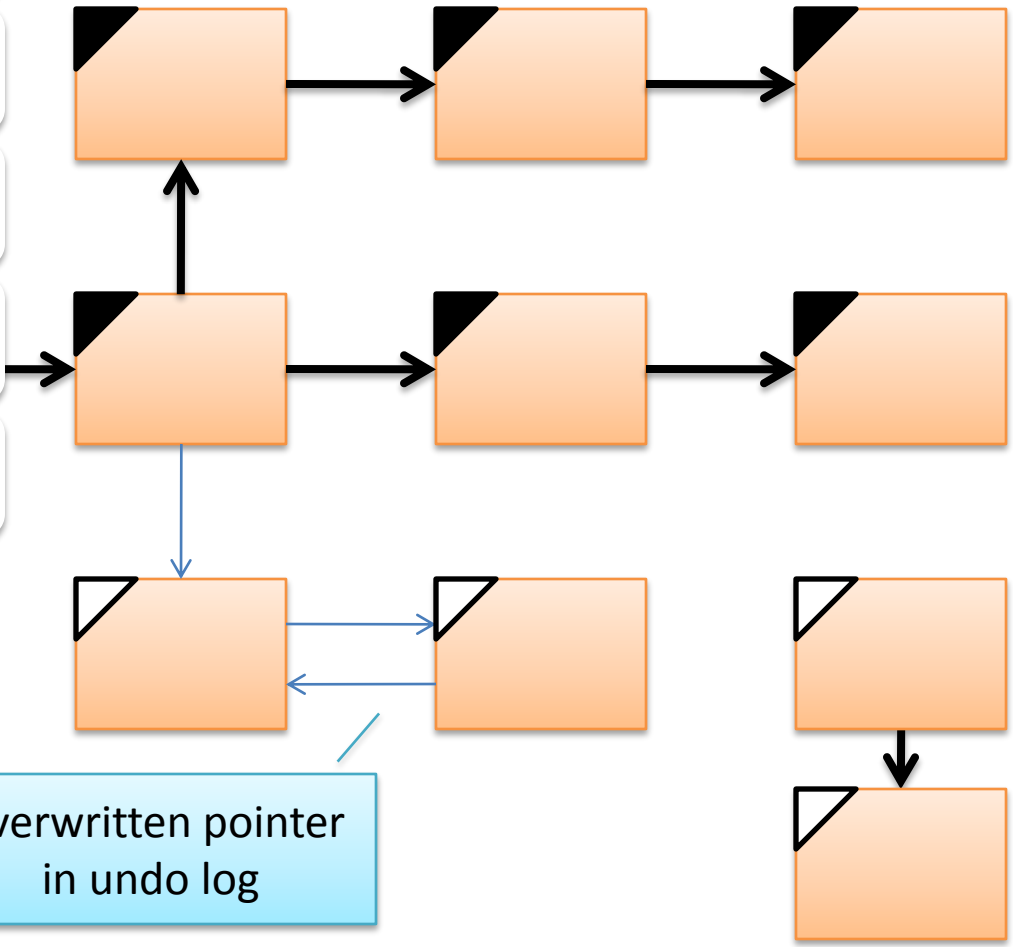
3. Roll-back, re-gray updated black obj

4. Trace from gray objects

5. Reclaim white objects

Normal
heap
pointer

Overwritten pointer
in undo log



Precise algorithm

1. Validate tx

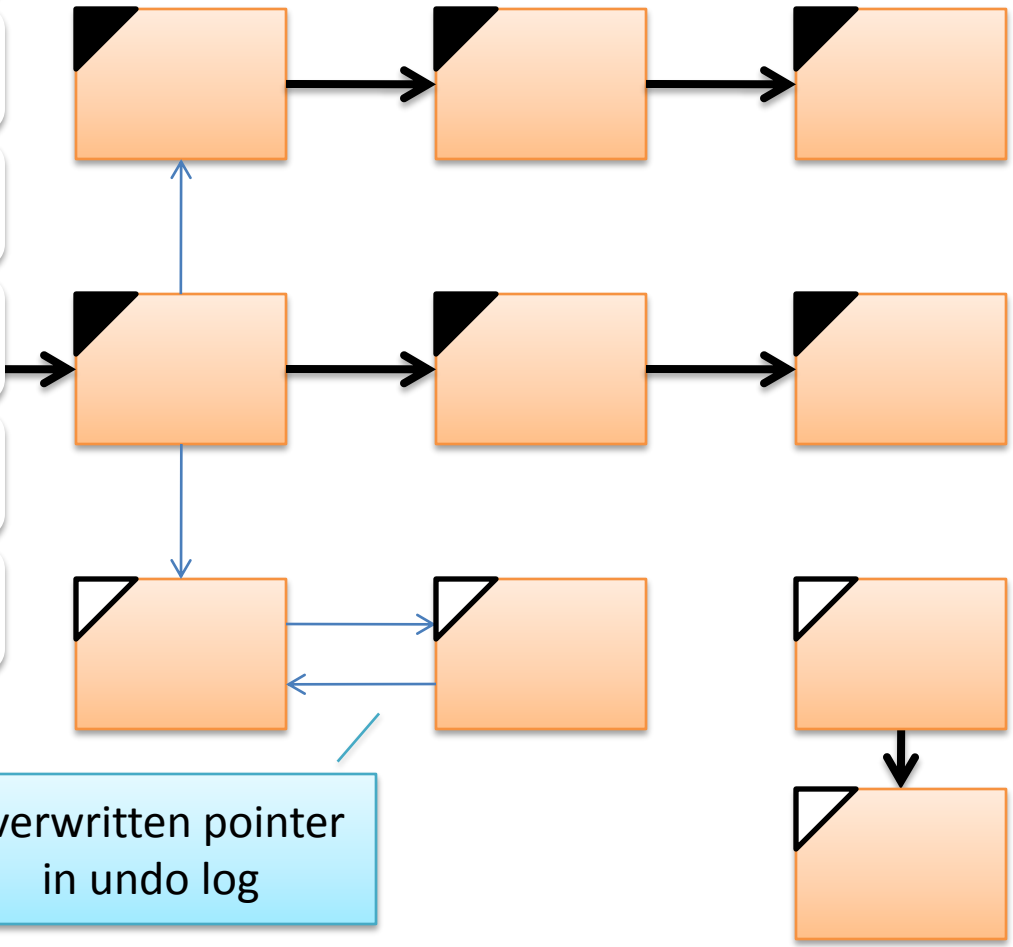
2. Trace heap as normal

3. Roll-back, re-gray updated black obj

4. Trace from gray objects

5. Reclaim white objects

6. Restore heap



Finalizers

```
Pair p;  
atomic {  
    p = new Pair();  
}
```

Suppose this block is attempted twice

How many times is this printed? (Or is this program wrong?)

```
Class Pair {  
    void Finalize() {  
        Console.Out.WriteLine("Hello world\n");  
    }  
}
```

Finalizers

- Remember the intended semantics:
 - Exactly once execution
- Transactionally-allocated objects are only eligible for finalization when the tx commits
- Tentative allocation, non-finalization, and (re-)execution remains entirely transparent

Condition synchronization

```
atomic {  
  buffer.data = 42;  
  buffer.full = true;  
}
```

This atomic block is
only ready to run
when buffer.full is true

```
atomic {  
  if (!buffer.full) {  
    retry;  
  }  
  result = buffer.data;  
  buffer.full = false;  
}
```

- Semantically: in STM-Haskell we required the scheduler to only run atomic blocks when they succeed without calling “retry”

Primitive for synchronization

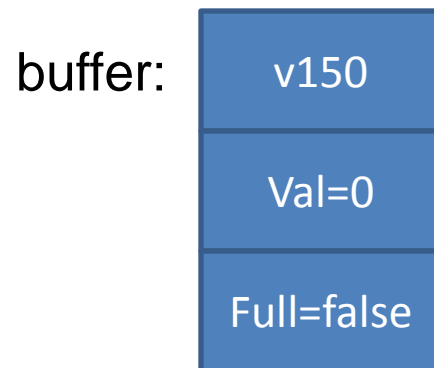
- `void WaitTX(tx)`
 - Semantically equivalent to `AbortTx`
 - Implementation may assume caller will immediately re-execute a (deterministic) tx
 - Implementation may introduce a delay to avoid unnecessary spinning
- Intuition:
 - No point re-executing the consumer until the producer has run

Compiling “retry” to “WaitTx”

```
atomic {  
    if (!buffer.full)  
        retry;  
}  
result = buffer.  
buffer.full =  
}
```

```
void Consume(Buffer b) {  
    do {  
        done = true;  
        try {  
            try {  
                tx = StartTx();  
                OpenForRead(tx, b);  
                if (!b.full) {  
                    waitTx();  
                }  
                OpenForUpdate(tx, b);  
                result = b.data;  
                LogForUndo(tx, &b.full);  
                b.full = false;  
            } finally {  
                CommitTx();  
            }  
        } catch (TxInvalid) {  
            done = false;  
        }  
    } while (!done);  
}
```

Implementing WaitTx



Implementing WaitTx

1. Extend object header
with list of waiters

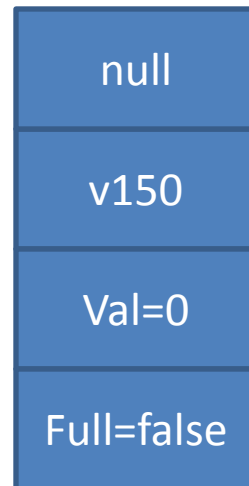
buffer:

null
v150
Val=0
Full=false

Implementing WaitTx

1. Extend object header with list of waiters

2. Extend tx records with a mutex & condvar pair

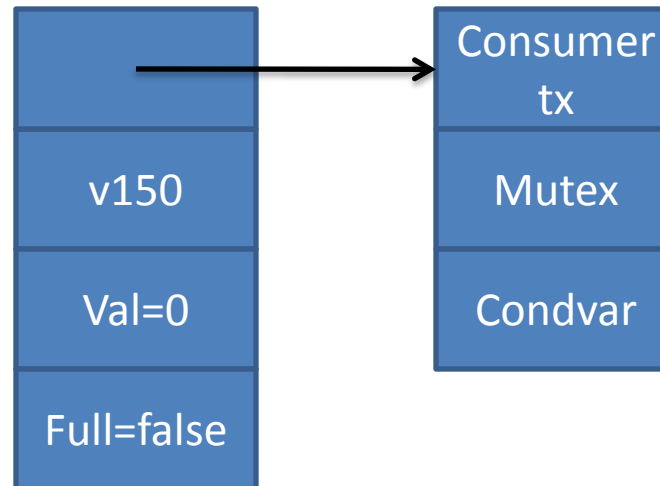


Implementing WaitTx

1. Extend object header with list of waiters

2. Extend tx records with a mutex & condvar pair

3. WaitTx links the consumer to the lists in its read set



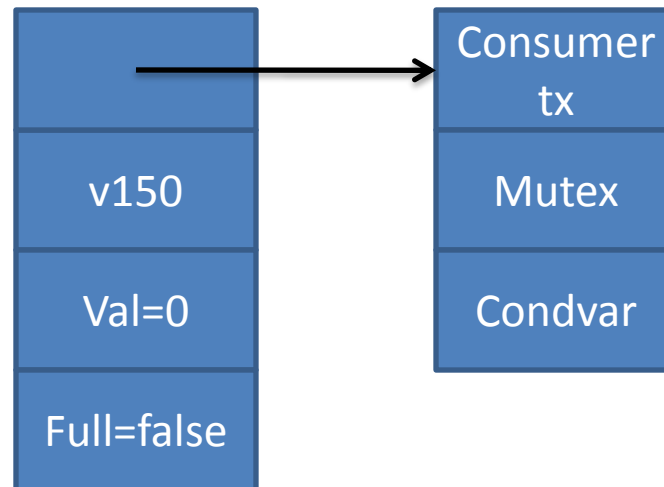
Implementing WaitTx

1. Extend object header with list of waiters

2. Extend tx records with a mutex & condvar pair

3. WaitTx links the consumer to the lists in its read set

4. WaitTx validates, locks the mutex, updates its status, blocks



Implementing WaitTx

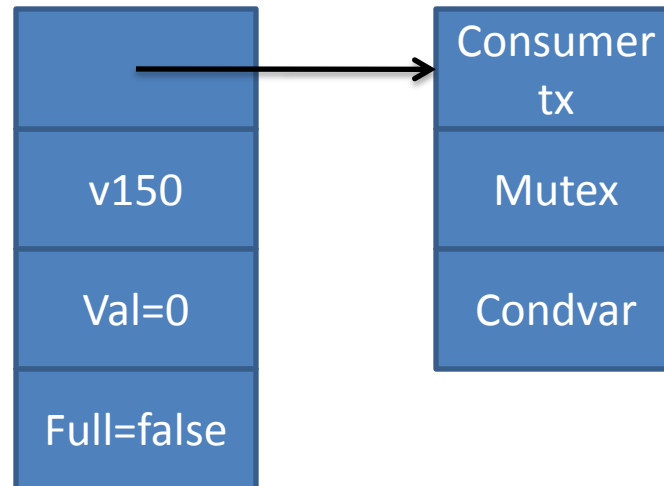
1. Extend object header with list of waiters

2. Extend tx records with a mutex & condvar pair

3. WaitTx links the consumer to the lists in its read set

4. WaitTx validates, locks the mutex, updates its status, blocks

5. CommitTx wakes waiters on objects in its write set

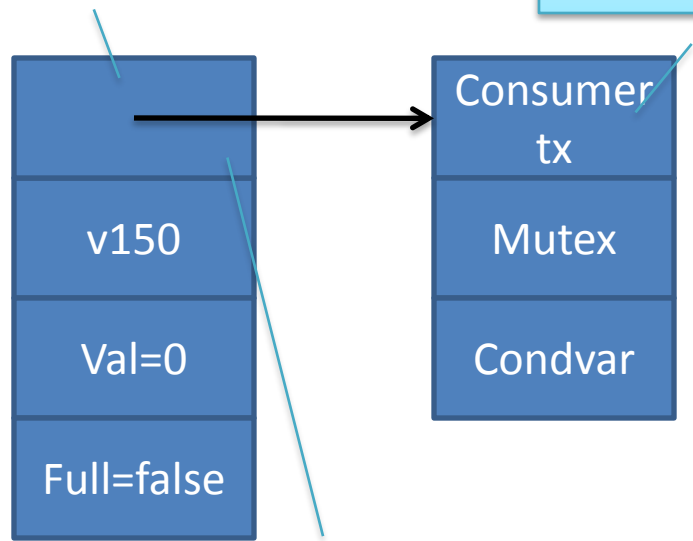


In WaitTx

1. Extend object header with list of waiters
2. Extend tx records with a mutex & condvar pair
3. WaitTx links the consumer to the lists in its read set
4. WaitTx validates, locks the mutex, updates its status, blocks
5. CommitTx wakes waiters on objects in its write set

Use "thin locks" style tricks to avoid fixed header word allocation

NB: many-to-many relationship, so probably use separate doubly linked list



Use latch in the header for concurrency control on the list

Conclusion

- Basic transformation to use STM operations look straightforward but there are lots of details
 - ...these will vary from language to language
- There's still a lot of scope for program transformations to reduce the number of STM operations
- Scalability and performance may rely on integration between the STM and runtime system

Further reading

- “Language support for lightweight transactions”, Tim Harris and Keir Fraser. OOPSLA 2003. *Introduced atomic blocks built over STM.*
- “Composable memory transactions”, Tim Harris, Maurice Herlihy, Simon Marlow, Simon Peyton Jones. PPOPP 2005. *Introduced “retry” and “orElse” for blocking.*
- “Optimizing memory transactions”, Tim Harris, Mark Plesko, Avraham Shinnar, David Tarditi. PLDI 2006. *Compiler and runtime optimizations for building atomic blocks over STM.*
- “Compiler and runtime support for efficient software transactional memory”, Ali-Reza Adl-Tabatabai, Brian T Lewis, Vijay Menon, Brian R Murphy, Bratin Saha, Tatiana Shpeisman. PLDI 2006. *Compiler and runtime optimizations for building atomic blocks over STM.*
- “A uniform transactional execution environment for Java”, Lukasz Ziarek, Adam Welc, Ali-Reza Adl-Tabatabai, Vijay Menon, Tatiana Shpeisman, Suresh Jagannathan. ECOOP 2008. *STM-based speculative lock elision in Java, combination of atomic blocks and existing language constructs.*

Lecture 4

Memory models

The Story So Far

- Atomic blocks built over STM
- Integration with the language RTS
- Blocking & wake-up
- Non-conflicting transaction run and commit in parallel
- Conflicts between transactions are detected
- But lots of STM-specific things leak out...



Undetected conflicting access

```
atomic {  
    x++;  
}
```

```
x = 100;
```

- Can we see “x == 1”?
- In most STM implementations a transaction does not detect a conflicting write from non-transactional code

Reading non-committed state

```
atomic {  
    x = 10;  
    <ABORT>  
}
```

```
temp = x;
```

- Can we see “temp == 10”?
- Eager versioning (“update-in-place”) STM implementations don’t restrict direct access to shared data

Zombie updates

```
atomic {  
    x = 1;  
    y = 1;  
}
```

```
atomic {  
    if (x != y) z = 1;  
}
```

```
temp = z;
```

- Can we see “temp == 1”?
- STMs using lazy conflict detection allow atomic blocks to keep running after conflicts
- Non-transactional code may see these zombies

Privatization

```
x_shared = true;    x = 0;
```

```
atomic {  
    if (x_shared) {  
        x = 100;  
    }  
}
```

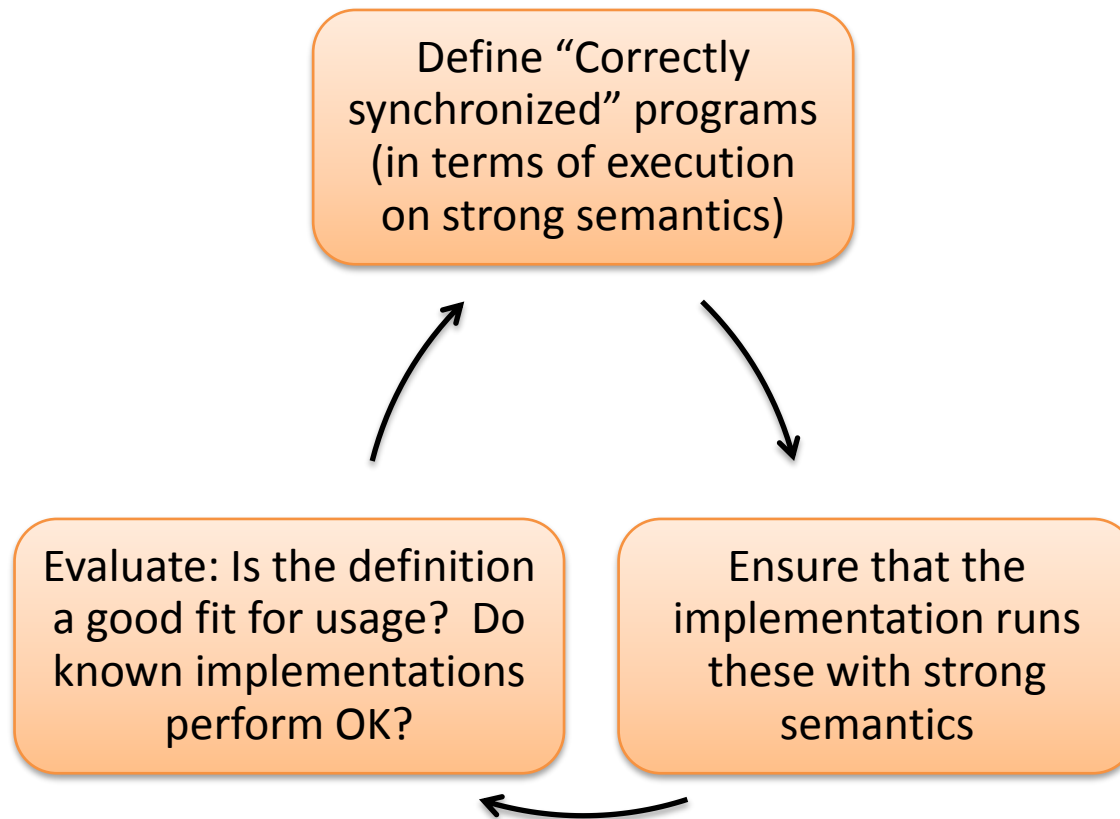
```
atomic {  
    x_shared = false;  
}  
// work on x privately
```

- Both eager-versioning and lazy-versioning STM implementations can allow the store of 100 to stomp on x during the private work

What should we do about this?

- Dealing with examples like these is the subject of ongoing research. There are many options:
 - Consider these examples incorrect programs (if so then what do we still guarantee if they're written?)
 - Strengthen implementations to run them correctly
 - Develop type systems to prevent them being written
- These lectures reflect my thoughts on the problem, and some of the approaches from the research literature in mid-2008

Dealing with this methodically



Provide strong semantics for...

More implementation flexibility;
possibly better performance

All
programs

Violation-free
programs

Those obeying
dynamic separation

Those obeying
static separation

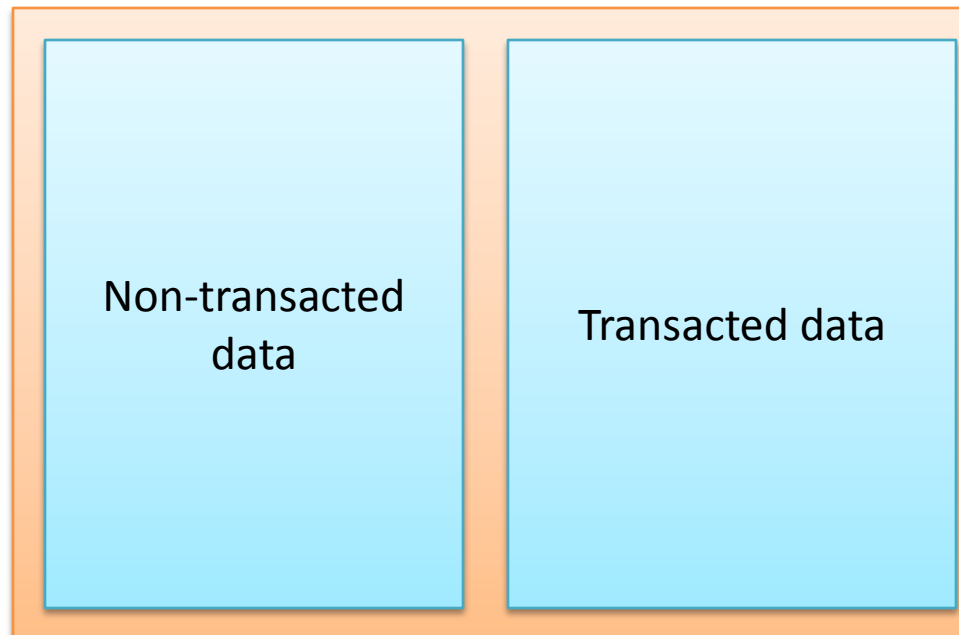
Possibly more implementation
complexity; possibly lower performance

These “possibilities” are important

- The example STMs both run programs obeying static separation with strong semantics
- Known implementations that support violation freedom add extra complexity and/or synchronization
- Known implementations of strong atomicity in software add extra complexity and/or overhead on non-transacted code

Does these aspects of current implementations hint at fundamental impossibility results? Or are there alternative implementations we've not yet invented?

Static separation



Static separation

- + Very permissive in terms of STM implementation – e.g. Bartok-STM and TL2 are both OK
- + Can be enforced by a simple type system
- + Perhaps we should distinguish shared data in some way
- Simple type system rules out many “sensible” programs
- ...and may require code duplication or further annotations

Static separation in Haskell

- Distinguish between three kinds of operation:
 - Pure computation – can do this anywhere
 - Operations with STM side-effects – can compose these with one another, and express blocking and alternatives
 - Operation with arbitrary side-effects – cannot form part of atomic actions (external observers or other threads may see them)
- Write ‘atomic’ *only when a compound atomic action needs to be performed*
 - Typically when it needs to be composed with IO

Static separation in Haskell

- Impure Haskell code is distinguished through the type system

```
readString :: IO String
```

When evaluated, has 'IO' side effects and returns a string

```
writeString :: String -> IO ()
```

Takes a string, has an 'IO' side effect, void return type

- A function with an 'ordinary' type (`Int -> String`) is *guaranteed* to be pure

Static separation in Haskell

- Extend the type system to distinguish reversible side effects to memory from “real” IO:

`newTVar :: a -> STM TVar a`

`readTVar :: TVar a -> STM a`

`writeTVar :: TVar a -> a -> STM a`

`retry :: STM a`

`orElse :: STM a -> STM a -> STM a`

`atomic :: STM a -> IO a`

Explicit access to
transactional variables

Composable blocking

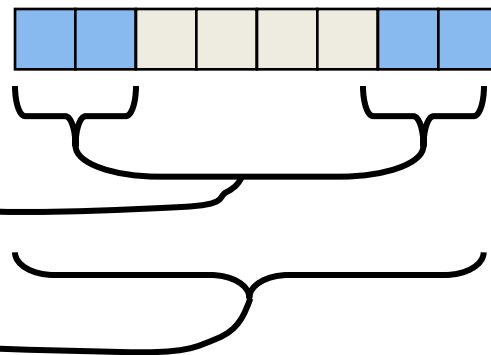
Run an atomic action so it can be
composed with external IO

Example: an array-based queue

A queue holding elements of type 'e'

```
data ArrayBlockingQueueSTM e = ArrayBlockingQueueSTM {
  shead :: TVar Int,
  stail :: TVar Int,
  sused :: TVar Int,
  slen :: Int,
  sa :: Array Int (TVar e)
}
```

An integer-indexed array, holding values of type 'TVar e'



Example: an array-based queue

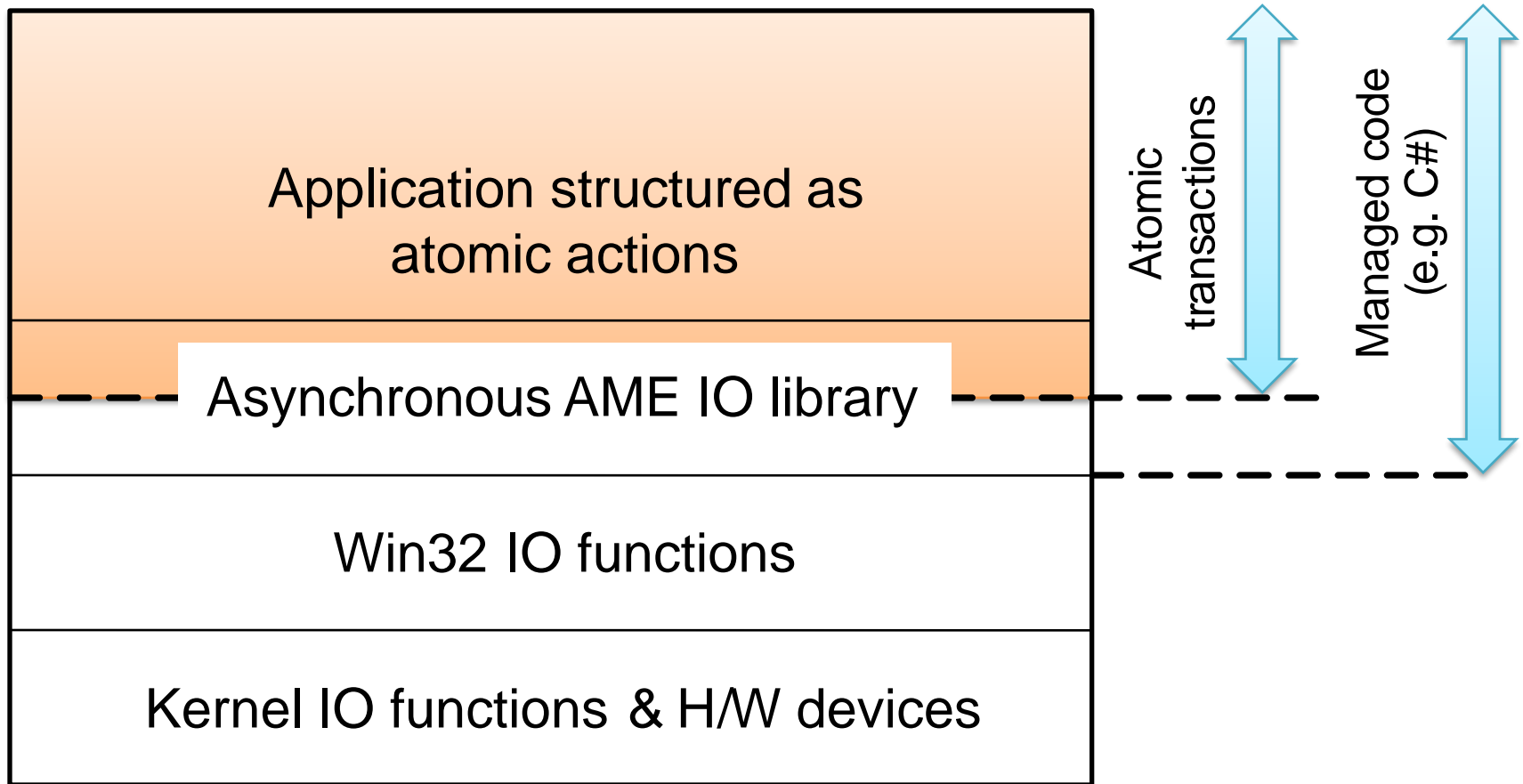
```
readHeadElementSTM :: ArrayBlockingQueueSTM e
                    -> Bool -> STM e

readHeadElementSTM abq remove
= do u <- readTVar (sused abq)
    if u == 0
    then retry
    else do h <- readTVar (shead abq)
        let tv = sa abq ! h
            e <- readTVar tv
        if remove
        then do
            let len = slen abq
                newh = (h+1) `mod` len
            writeTVar (shead abq) $! newh
            writeTVar (sused abq) $! (u-1)
        else return ()
    return e
```

Static separation in Haskell

- Why does this work in Haskell?
- Do you think it would work in C# or Java?

Dynamic separation in AME




```
// Atomic action A1
if (status == Idle) {
  <populate buffer b>

  // IO library
  status = Requested;
  unprotected {
    // Unprotected code U1
    <check buffer>
    pin(b);
    unsafe {
      syscall(&b[0]);
    }
    unpin(b);
  }
  status = Complete;
}
```

Must make A1's updates must be visible to the syscall

```
// Atomic action A2
blockuntil(status == Complete);
<use results from buffer>
```

Syscall's updates to "b" must be seen here

```
// Atomic action A3
if (status == Idle) {
  <populate buffer b>
  status = Requested;
  ...
}
```

A3's implementation must not stomp on "b"

```
// Atomic action A1
if (status == Idle) {
    <populate buffer b>

    // IO library
    status = Requested;
    unprotected {
        // Unprotected code U1
        unprotect(b);
        <check buffer>
        pin(b);
        unsafe {
            syscall(&b[0]);
        }
        unpin(b);
        protect(b);
    }
    status = Complete;
}
```

```
// Atomic action A2
blockuntil(status == Complete);
<use results from buffer>
```

```
// Atomic action A3
if (status == Idle) {
    <populate buffer b>
    status = Requested;
    ...
}
```

– also `shareReadOnly(b)`

Using dynamic separation (DS)

- To write a correctly synchronized program:
 - Protected data must only be accessed in atomic actions
 - Unprotected data must only be accessed outside atomic actions
 - Shared read-only data can be read anywhere
- A contract between the programmer and language implementer:
 - If the program is correctly synchronized
 - Then it must be implemented with strong semantics

Correctly synchronized examples

```
b = new Buffer();
<popu
unpro
// Initially b_shared=true, b_shared protected, b protected

// Initially x=y=0, x protected, y protected, z unprotected

} // Atomic action A1          // Atomic action A2
x = 10;                        if (x != y) {
y = 10;                        z = 42;
<u                               }

// Unprotected code U1
unprotected {
  // Can we see z==42?
}
```

Supporting dynamic separation in Bartok

- Implementation invariant:
 - Transactions only update protected objects
- Add a 'protected' flag to the STM word
 - Check it in OpenForUpdate
 - Set? Proceed as normal
 - Clear? Validate transaction
 - Validation succeeds: program is not "good"
 - Validation fails: transaction was invalid
 - Update it in Protect/Unprotect/ShareRead only
 - Wait until the object is not OpenForUpdate
 - Update the flag

Dynamic separation in AME

- Why does dynamic separation work well for AME?
- Do you think it would work well for Java or C# with atomic blocks?

Violation freedom

- Informally: the program should not try to access the same data inside and outside an atomic block at the same time
- Remember: we reason about this under the strong semantics; think about “atomic”, not “STM” and transactions

Conflicting access; not violation-free

```
atomic {  
    x = 10;  
}
```

```
temp = x;
```

- Violation on x
- Under a “catch fire” style of definition we would say nothing about what this program may do
- (We might hope to say more, in practice)

Privatization is OK

```
x_shared = true;    x = 0;
```

```
atomic {  
    if (x_shared) {  
        x = 100;  
    }  
}
```

```
atomic {  
    x_shared = false;  
}  
// work on x privately
```

- Most STM-based implementations don't support this
- These implementations are wrong if we take supporting violation-free programs as our goal

Eager versioning, invisible readers

```
x_shared = true;    x = 0;
```

The tx implementing this atomic block cannot know about the concurrent reader...

```
atomic {  
  if (x_shared) {  
    x = 100;  
  }  
}
```

Run this atomic block, pre-empt just before write to "x"

```
atomic {  
  x_shared = false;  
}  
// work on x privately
```

...so it cannot help but commit its update and work on x

Lazy versioning, invisible readers

`x_shared = true; x = 0;`

2. Run this atomic block in its entirety

```
atomic {
    if (x_shared) {
        x = 100;
    }
}
```

1. Run this atomic block, pre-empt during write-back of updates

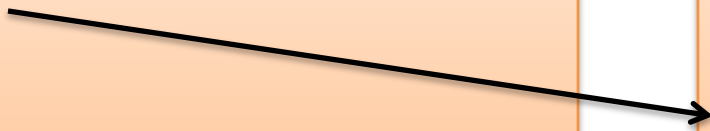
```
atomic {
    x_shared = false;
}
// work on x privately
```

3. These accesses to "x" race with the write-backs

Privatization

```
atomic {  
  S1;  
}
```

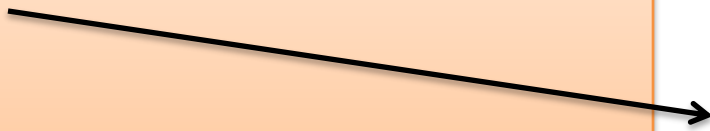
```
atomic {  
  S2;  
}  
S3;
```



- We want to be able to reason transitively:
 - S2 is serialized after conflicting tx S1
 - S3 comes after S2, so must see effects of S1

Publication

```
S1;  
atomic {  
  S2;  
}
```



```
atomic {  
  S3;  
}
```

- We want to be able to reason transitively:
 - S3 is serialized after conflicting tx running S2
 - S2 comes after S1, so S3 must see effects of S1

Supporting violation freedom

- (Assuming lazy versioning)
- Privatization:
 - If conflicting tx Tx1->Tx2 in serialization order then code after Tx2 can't start until Tx1's commit is done
- Publication:
 - If conflicting tx Tx1->Tx2 in serialization order then Tx2's execution must start after Tx1's starts

Violation-freedom and program transformations

```
atomic {  
  if (y) {  
    // Use x  
  }  
}
```

Can a compiler hoist the
read of "x" above the
read of "y"?



```
atomic {  
  temp = x;  
  if (y) {  
    // Use temp  
  }  
}
```

- A violation-free program becomes one with a violation!
- Either:
 - Prohibit this transformation (NB: does the transforming compiler know that the code is inside an atomic block?)
 - Improve the STM implementation to accept it (OK with TL2-style global versions, not OK with Bartok-style object versions)

Violation freedom

- There are many related notions and subtle examples here; these slides are intentionally informal
- Contrast “violation free programs run with strong semantics” with the notion of “single global lock atomicity”
- What does supporting violation-free programs with strong semantics mean for implementations?
 - Probably can’t use in-place updates & invisible readers
 - Probably can’t use invisible readers and retain disjoint-access-parallelism between the implementations of atomic blocks

Strong atomicity

- One way to view this:
 - Supporting violation-free programs requires us to deal with conflicts introduced by the STM implementation and the application
 - What if, instead, we deal with all conflicts between atomic blocks and the application?
- Perhaps pay a straight-line perf cost or implementation complexity cost
 - But retain disjoint-access parallelism, or flexibility between STM choices
- Get more intuitive semantics for “racy” programs as well

Basic approach for strong atomicity

- Expand all memory accesses into short atomic blocks
- Avoid doing this on never-transactionally-accessed locations (reasoning in terms of the STM implementation)
- Provide streamlined implementations and/or combine blocks to amortise entry/exit costs

Dynamic optimization

- JIT-based implementation
- Invariant:
 - No conflicts are possible from direct accesses in non-transacted code
- When compiling a tx, the JIT invalidates code that might violate this invariant. Recompile it with barriers
- Can combine the basic approach with more precise static analyses to rule out conflicts

Conclusion

- Many subtleties here were not initially understood
- There is unlikely to be a “right” or “wrong” answer; it’s an engineering trade-off between the programs that are supported and the implementation techniques we can use
- Formal semantics help identify which programs are correctly synchronized and/or what might happen in programs that are not
- There’s relatively little work in this space in the broader programming language community – many concurrent languages have imprecise definitions of behavior in the presence of races

Further reading (1)

- “Subtleties of Transactional Memory Atomicity Semantics”, Colin Blundell, E Christopher Lewis, Milo M K Martin, Computer Architecture Letters, November 2006. *Weak and strong atomicity.*
- “The Java memory model”, Jeremy Manson, William Pugh, Sarita V Adve. POPL 2005. *Illustrates the idea of providing strong semantics (in their case sequential consistency) to a class of correctly synchronized program. (The idea itself goes back to Adve and Hill’s earlier work)*
- “What does atomic mean?”, Tim Harris. London Theoretical Computer Science Seminar, April 2007. *Identified the link from Adve & Hill’s work to STM.*
- “Privatization techniques for software transactional memory”, M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott. PODC 2007 (Brief announcement). *Concurrently identified the link from Adve & Hill’s work to STM, plus fence techniques for supporting privatization. A longer version is available as Rochester CS TR915.*
- “Enforcing isolation and ordering in STM”, Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steve Balensiefer, Dan Grossman, Richard Hudson, Katherine F Moore, Bratin Saha. PLDI 2007. *Study of tx/non-tx interactions. Basic strong atomicity implementation and optimizations.*

Further reading (2)

- “High-level small-step operational semantics for transactions”, Katherine Moore, Dan Grossman. POPL 2008. *Formal semantics and type system for static separation.*
- “Semantics of transactional memory and automatic mutual exclusion”, Martín Abadi, Andrew Birrell, Tim Harris and Michael Isard. POPL 2008. *Formal semantics and type system for static separation.*
- “A model of dynamic separation for transactional memory”, Martín Abadi, Tim Harris and Katherine Moore. CONCUR 2008. *Dynamic separation. Also see the practically-focussed tech report that accompanies this paper.*
- “Practical Weak-Atomicity Semantics for Java STM”, Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard Hudson, Bratin Saha and Adam Welc. SPAA 2008. *Design trade-offs when adding “atomic” blocks to Java and relating their behavior to that of locks.*
- “Dynamic Optimization for Efficient Strong Atomicity”, Florian Shneider, Vijay Menon, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai. OOPSLA 2008.

Lecture 5

Extensions and research
directions

Introduction

- This lecture looks at extensions to the basic model we've look at so far
- Much of this is ongoing research
 - There is limited experience building large applications with many of these constructs
 - Not all of the constructs have been implemented in the same setting
- Where relevant I'll include references to the original papers as we go along

Language features: explicit aborts

- We avoided this for simplicity
- Typically exposed by unhandled exceptions
 - Validate the transaction
 - Marshal the exception out in some way
 - Roll back the transaction's operations
- Compare this with a unilateral Tx.Abort – what are the advantages, disadvantages?

Abort on exceptions

- Advantages:
 - Manual clean-up after errors can be difficult
 - STM provides the basic mechanisms needed
- Disadvantages:
 - Incompatible with “erasure” semantics (add/remove “atomic” in sequential code with no effect)
 - Needs to be build manually when building atomic blocks out of locks

Language features: closed nesting

- Interacts with explicit aborts
- Interacts with retry: “does retry in S2 just restart S2 or S1”?
 - Look at the semantics
- Provide some robustness against long roll-backs (conflict during S2 doesn't require re-execution of S1)

```
atomic {  
  S1;  
  atomic {  
    S2;  
  }  
  S3;  
}
```

Language features: or else

- We have three “raise”-like operations
 - throw ~ catch
 - abort ~ atomic
 - retry ~ ...

Language features: orelse

- “{X} orelse {Y}”
 - From STM-Haskell
 - Attempt to run X, if it's not ready (calls “retry”), then attempt to run Y instead
- Composable blocking
 - Wait conditions remain in X and Y
 - Callers can select whether or not to block

Language features: always

- “always X”
 - From STM-Haskell
 - “X” is an expression that must yield true
 - Ensure that “X” is preserved by subsequent atomic blocks
- Design choice if “X” is violated:
 - Block? Fail?

Language features: checkpoints

- Separate any semantic effects of closed nesting from its use for performance
- Checkpoint state
 - No effect at level of (e.g.) STM-Haskell semantics
 - Roll-back to checkpoint before first read of data suffering a conflict

```
atomic {  
  S1;  
  checkpoint;  
  S2;  
  checkpoint;  
  S3;  
}
```


Language features: ANTs

- “Abstract nested transactions”
- No effect semantically
- Implementation:
 - Log S2’s execution independently so it can be re-executed in isolation
 - Conservatively: check S2’s read/write set is disjoint from enclosing tx, check interaction through locals

```
atomic {  
  S1;  
  ant {  
    S2;  
  }  
  S3;  
}
```

Language features: open nesting

- Provide a form of nested transaction that can commit (to the heap) before its parent
- Provide mechanisms to register handlers
 - Abort, Commit, Validation, Top-level commit
- Use to combine external resources with STM
 - 2PC between resource managers
 - Impacts STM (or STM must vote last)
- Use to get better scalability on shared-memory structures (e.g. counters incremented by concurrent tx)

Language features: boosting

- Support existing scalable data structures for use within transactions
- Data structure implements a properly specified interface
 - Inverses (for use upon tx roll-back)
 - Commutativity info (for use for concurrency control)

Language features: boosting

```
public class SkipListKey {
    ConcurrentSkipListSet<Integer> list ;
    LockKey lock;

    public boolean add(final int v) {
        lock . lock(v);
        boolean result = list .add(v);
        if ( result ) {
            Thread.onAbort(new Runnable() {
                public void run() { list . remove(v);}
            });
        }
        return result ;
    }
    ...
}
```

Tx-aware locking
prevents non-
commutative boosted
operations

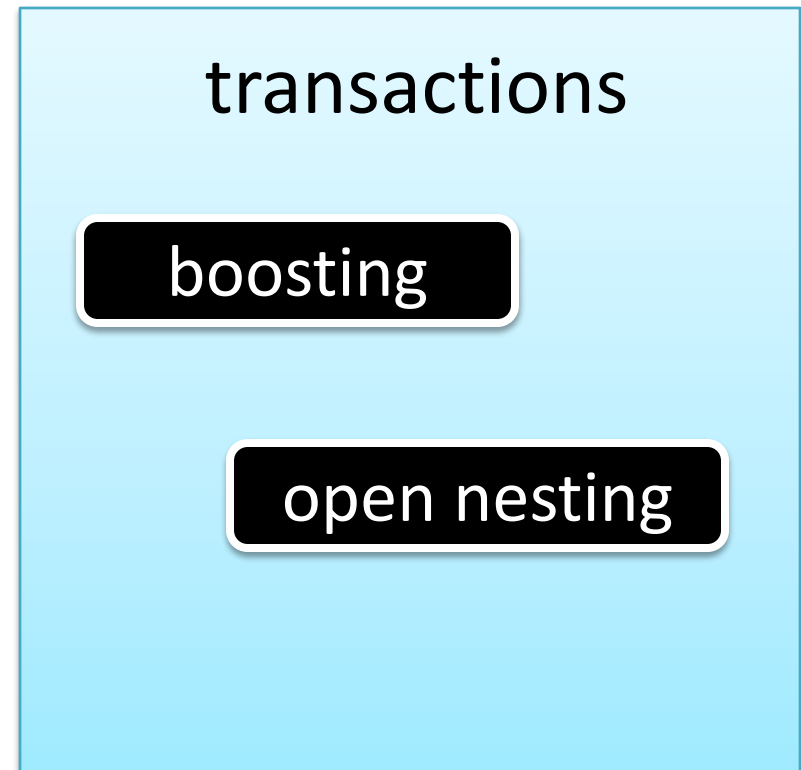
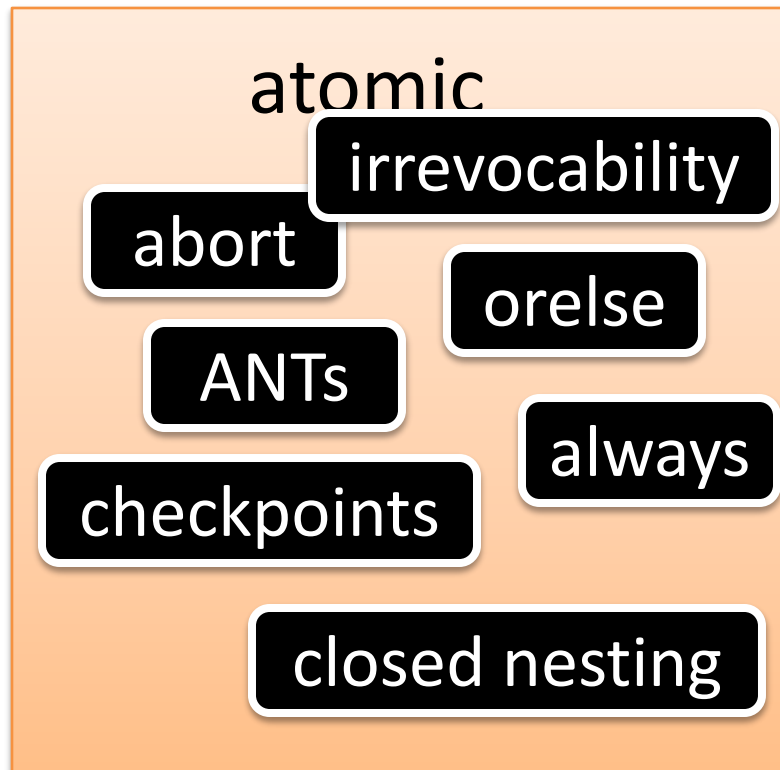
Code provides inverse:
remove any items
inserted

Language features: irrevocability

- Permit at most one transaction to be “irrevocable”
 - It must not be allowed to abort
 - Eases deployment path: let transactions make system calls
 - Can be used to signal that the implementation should favour certain atomic blocks
- Interaction with explicit aborts, condition synchronization, and STM implementation mechanisms
- Interacts with memory model
 - What if a system call accesses memory that the transaction has accessed?

Language features; discussion

- Distinguish based on level of definition?



Language features; discussion

- Another take on it: layering

Higher-level code is aware of “atomic”

Correct use of (e.g.) open nesting

Lower-level code is aware of transactions

- Much like other low-level parts of the system are aware of tx (e.g. the GC)

Progress – single-lock semantics?

```
// Thread 1
atomic {
  while (true) {
  }
}
```

```
// Thread 2
atomic {
  o1.x = 42;
}
```

- Is it OK for an implementation to reach a state from which Thread 2 can never execute its atomic block?
- This will never happen with atomic blocks built over Bartok-STM

Progress – ugly synchronization

```
// Thread 1
atomic {
  while (!o1.done) {
  }
}
```

```
// Thread 2
atomic {
  o1.done = true;
}
```

- Is it OK for an implementation to reach a state from which Thread 2 can never execute its atomic block?
- Again, this will never happen with atomic blocks built over Bartok-STM

Progress – uglier synchronization

```
// Thread 1
atomic {
  o1.count++;
  while (!o1.done) {
  }
}
```

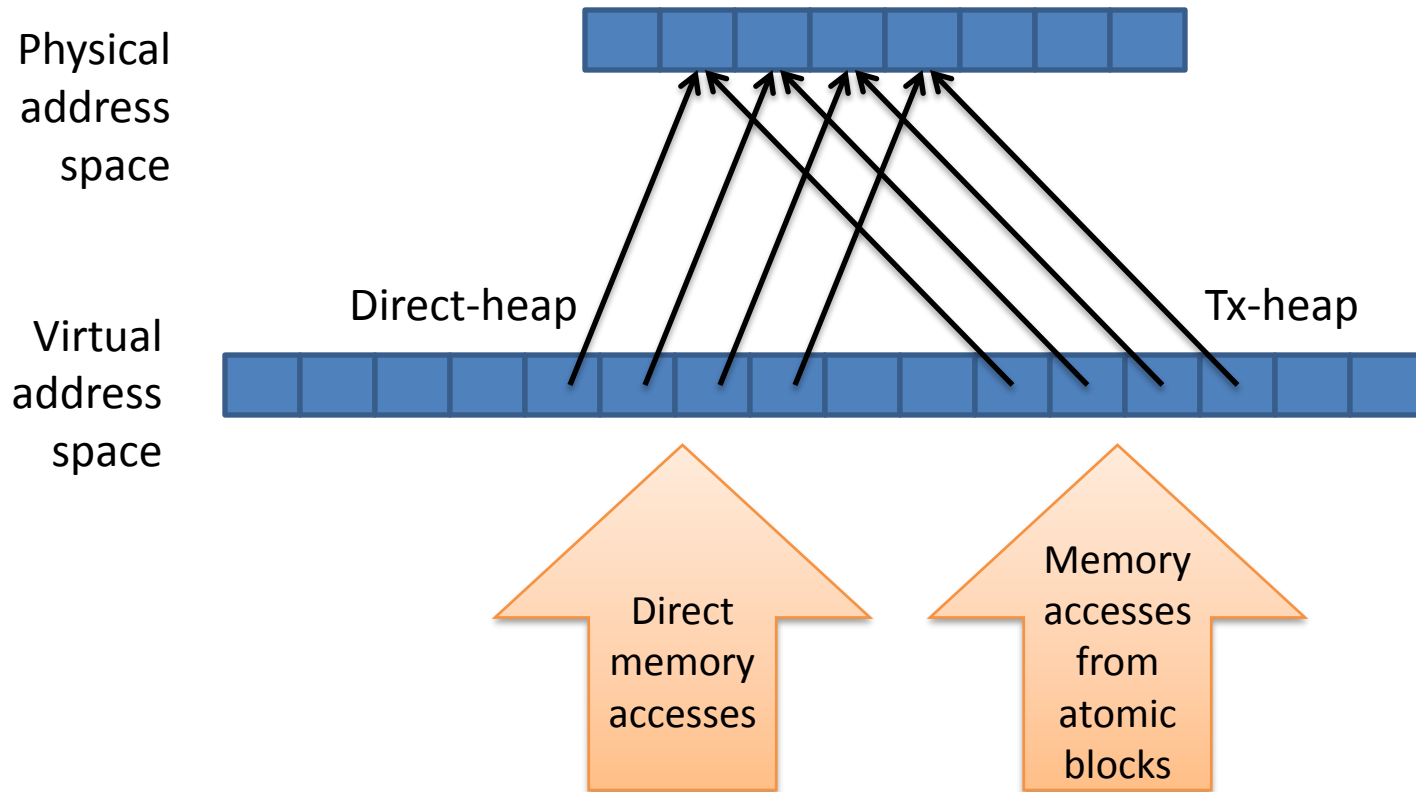
```
// Thread 2
atomic {
  o1.count++;
  o1.done = true;
}
```

- Is it OK for an implementation to reach a state from which Thread 2 can never execute its atomic block?
- Thread 2 *can* get stuck in Bartok-STM

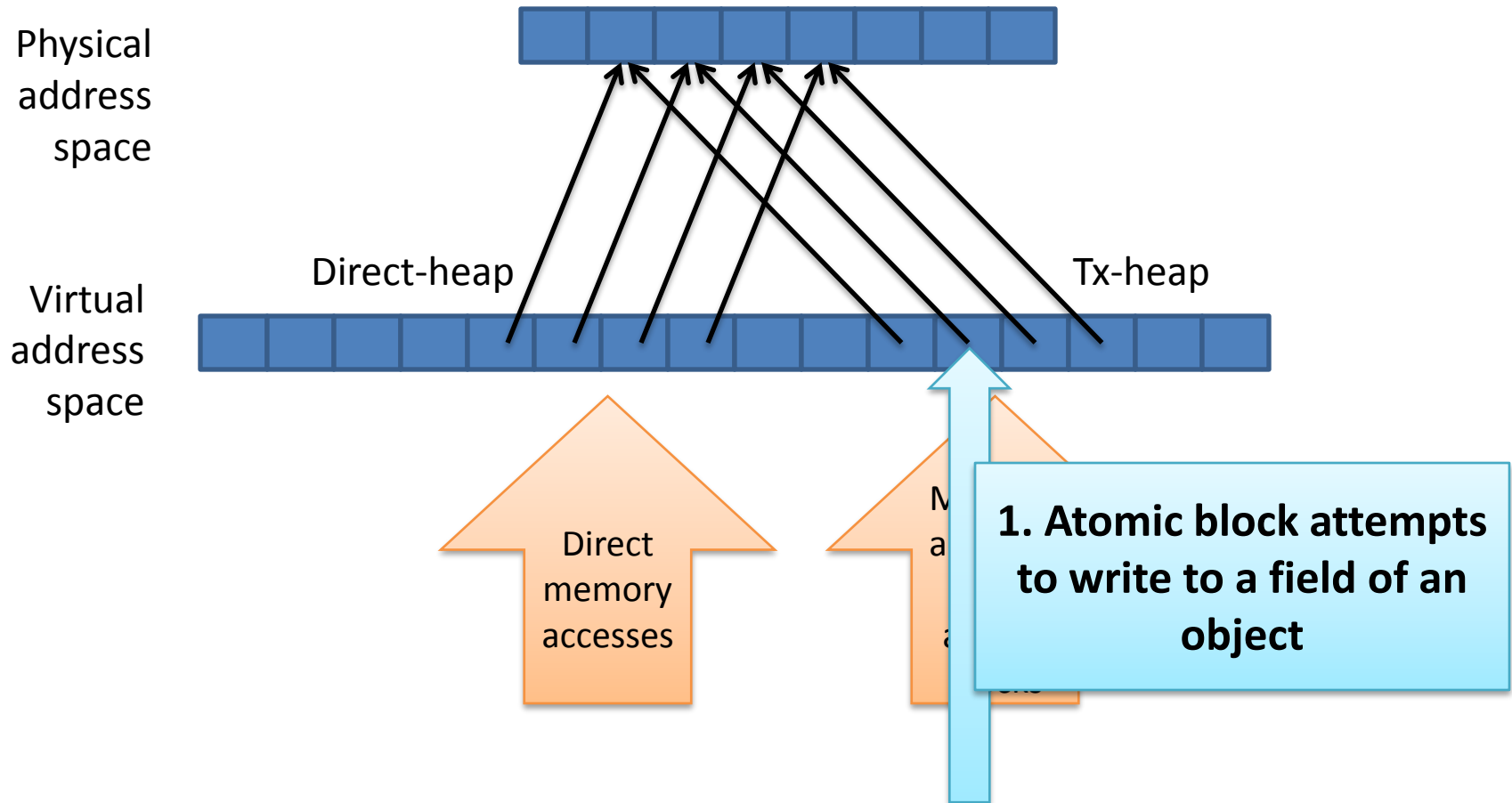
Progress

- Alternatively:
 - Require non-blocking atomic blocks
 - Build over non-blocking STM
- Can we come up with something weaker than this?
 - View single-global lock as a safety property
 - Provide some other specification for progress
- As usual we have a dilemma:
 - The programmer thinks at the source level
 - Conflicts are detected by the STM, possibly after program transformations

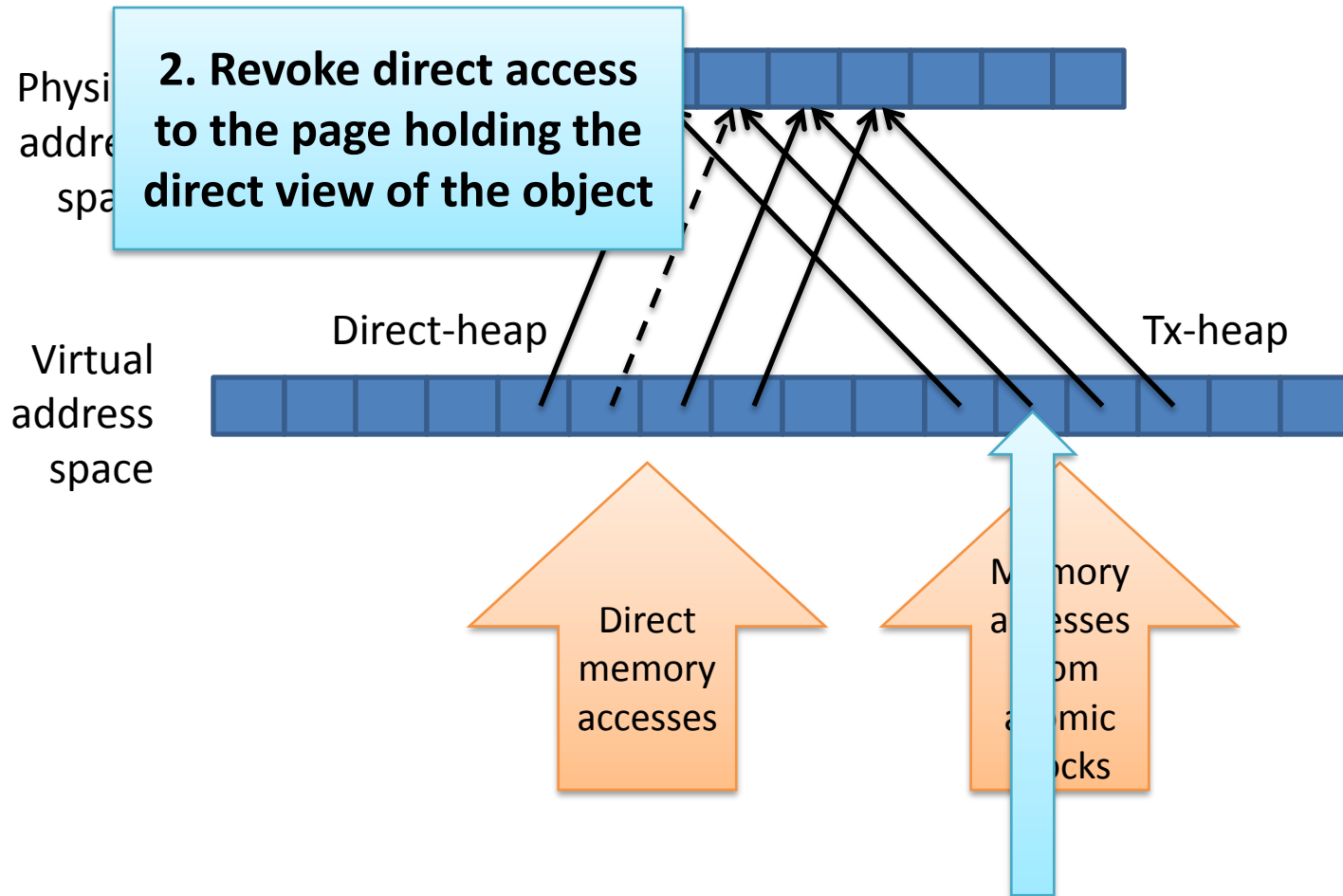
Exploiting page protection for strong atomicity



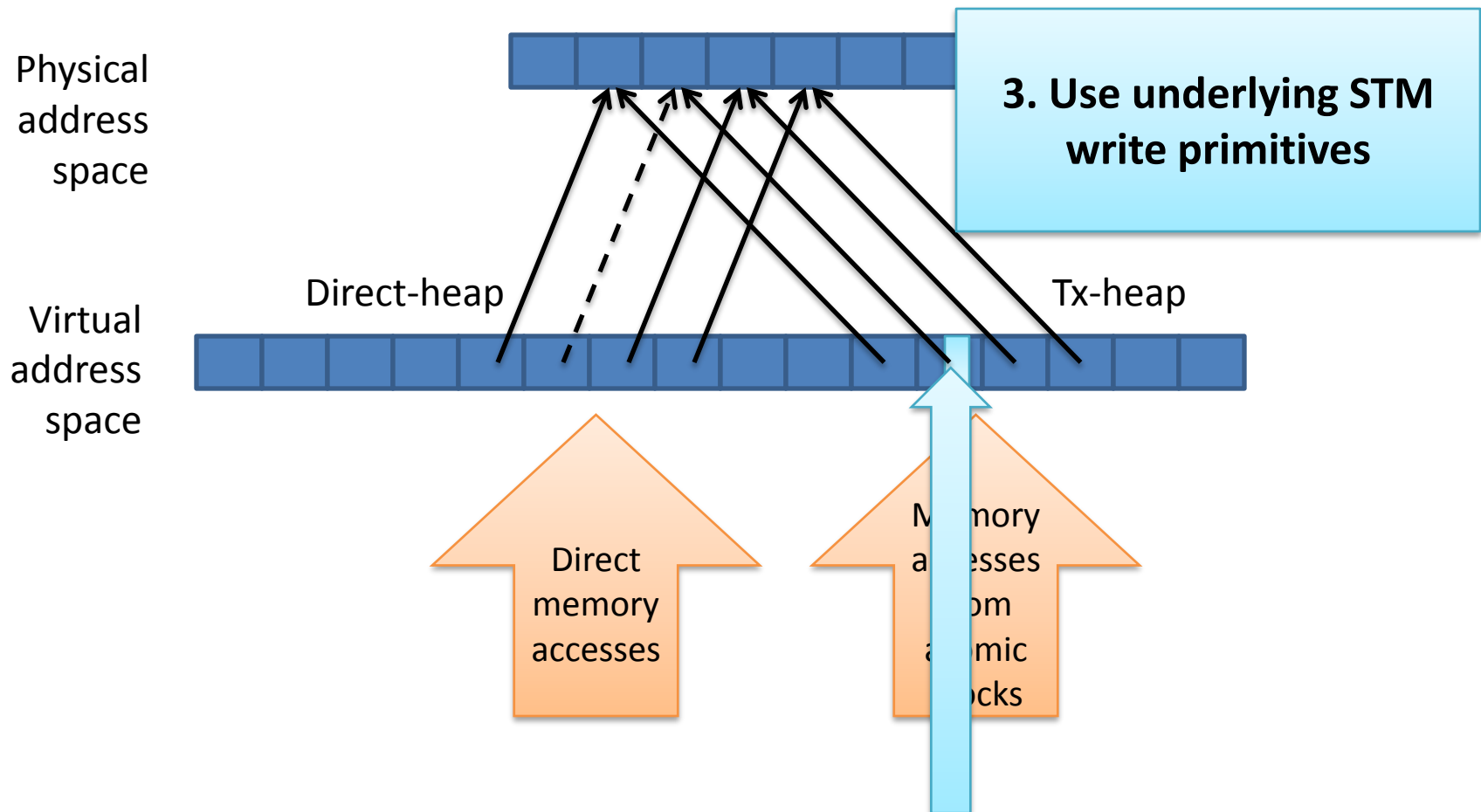
Writes from atomic blocks



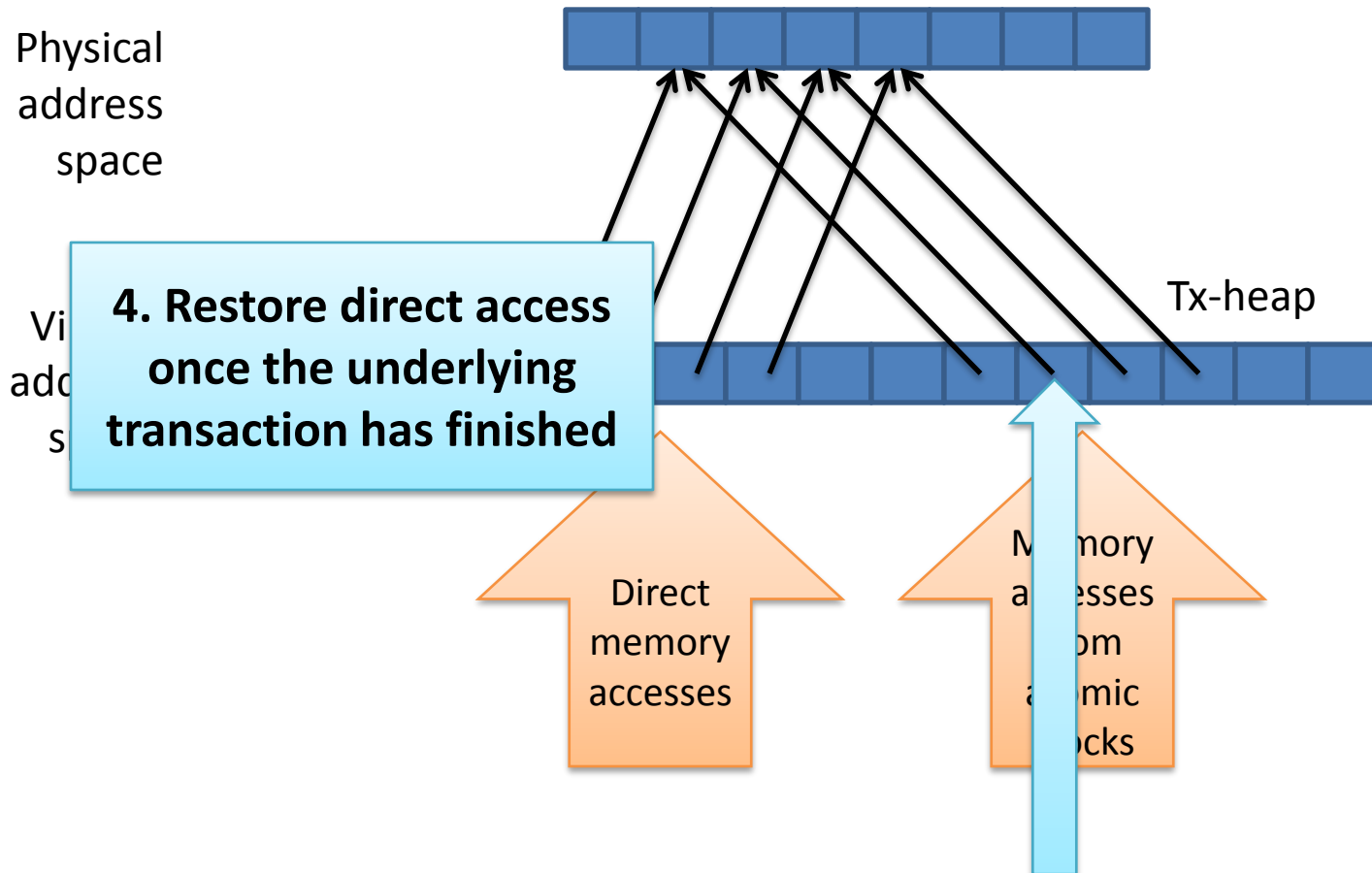
Writes from atomic blocks



Writes from atomic blocks



Writes from atomic blocks



Dealing with memory accesses

- Trap on memory fault
- OS redirects to C user-mode handler
- C handler builds fake call-frame to C# handler
- C# handler does a small transaction

Performance

- Make “safe” accesses directly to the atomic heap – vtables, array lengths, readonly, ...
- Make page protection changes lazily
- Heuristically segregate objects to reduce false sharing
 - At allocation time
 - At garbage collection time
- Recompile residual causes of access faults

Things I've not covered

- Non-linearizable transactions (e.g. snapshot isolation)
- Unmanaged code and system calls
- Parallelism within transactions
- Hardware implementations and acceleration
- Performance and usage examples

Summary

- I find it helpful to distinguish language features from implementation techniques
 - Can a language feature be defined independently of how it is implemented? Are alternative implementations possible?
- Formal foundations and specs are important:
 - Is an optimization correct?
 - Is a program correctly synchronized?
 - What should this program do?
- Benchmarks and workloads are important:
 - Is a program transformation genuinely worthwhile?
 - Which are the slow parts of an implementation?

