Microsoft Research

# Lock-free programming and transactional memory

Tim Harris

3/4

# Overview

Multi-core systems, and concurrent programming without locks

Transactional memory for managing shared-memory data structures

Integrating transactional memory into a modern, object-oriented programming language

Making sense of transactional memory: combining transactions with libraries, locking, and IO

Atomic blocks

Low-level API & optimizations

Sandboxing zombies

Runtime-system integration

Condition synchronization

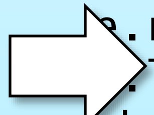
Microsoft Research

# Atomic blocks

```
Class Q {
  QElem l
  QElem r

  void pu
    atomi
      QEl
      e.r
      e.l
      thi
      thi
    }
  }

  ...
}
```



```
Class Q {
  QElem leftSentinel;
  QElem rightSentinel;

  void pushLeft(int item) {
    do {
      TxStart();
      QElem e = new QElem(item);
      TxWrite(&e.right, TxRead(&this.leftSentinel.right));
      TxWrite(&e.left, this.leftSentinel);
      TxWrite(&TxRead(&this.leftSentinel.right).left, e);
      TxWrite(&this.leftSentinel.right, e);
    } while (!TxCommit());
  }

  ...
}
```

# 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

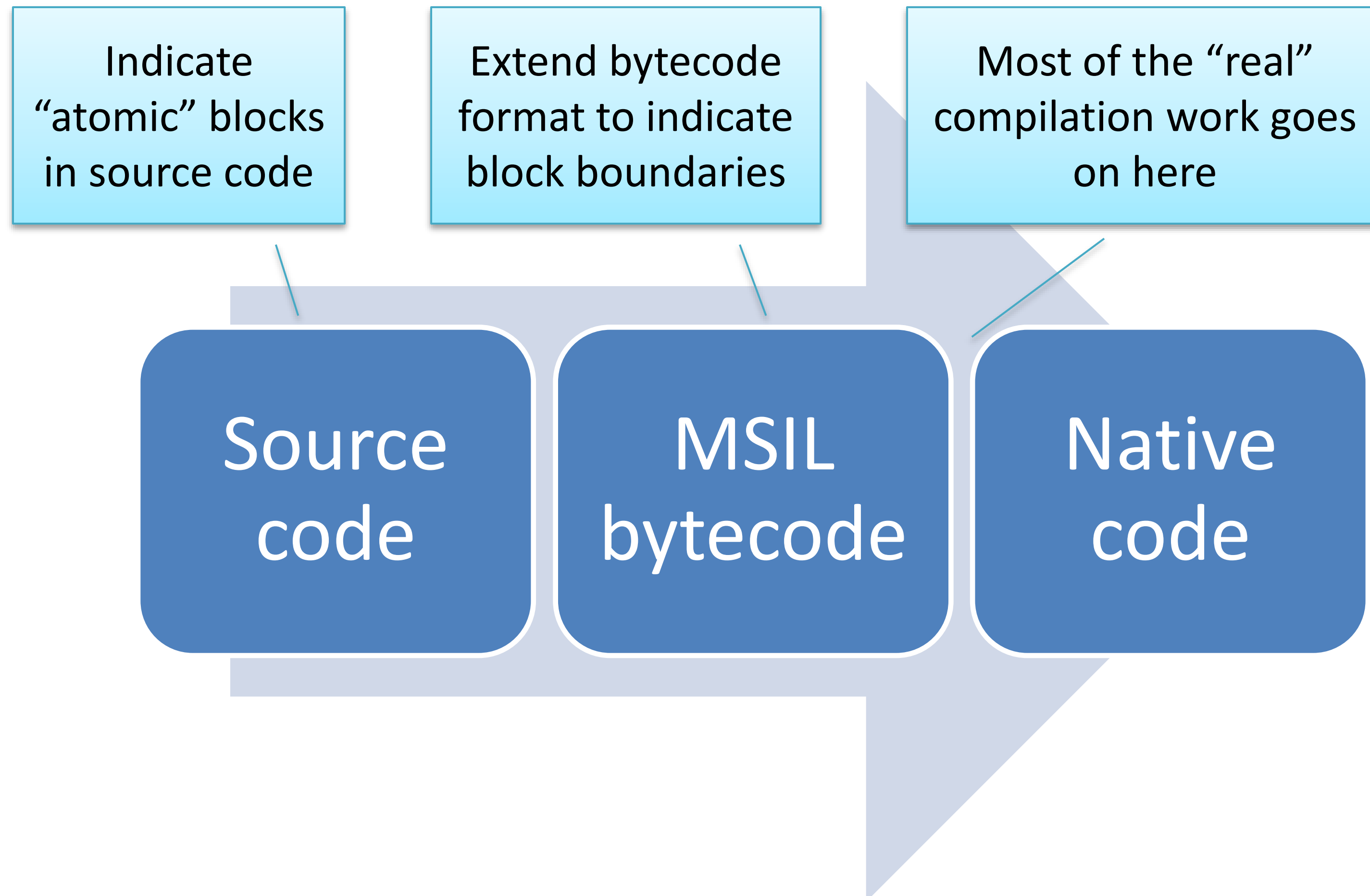
Source code

MSIL bytecode

Native code

Microsoft Research

# 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) {
  }
}
```

Microsoft Research

# 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);
}
```

Microsoft Research

Atomic blocks

Low-level API & optimizations

Sandboxing zombies

Runtime-system integration

Condition synchronization

# 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

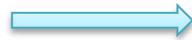
- OpenForRead(tx, obj)
- OpenForRead(tx, addr)
- OpenForUpdate(tx, obj)
- OpenForUpdate(tx, addr)
- LogForUndo(tx, addr)

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

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

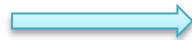
# Using the decomposed API

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



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

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



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

Microsoft Research

# 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

# 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;
  }
}
```

Microsoft Research

# 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;
  }
}
```

Microsoft Research

# 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>;
  }
}
```

Microsoft Research

# 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>;
   }
}
```

Microsoft Research

Atomic blocks

Low-level API & optimizations

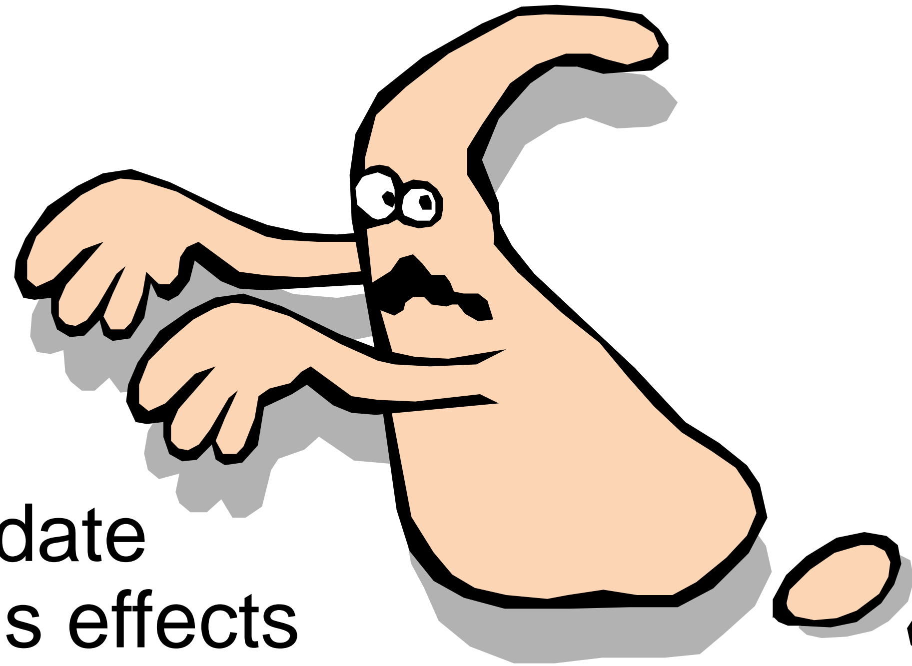
Sandboxing zombies

Runtime-system integration

Condition synchronization

# 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



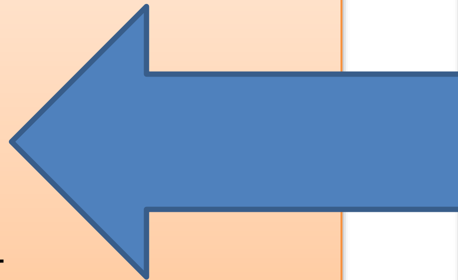
Microsoft Research

# 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

# Looping / slow zombies

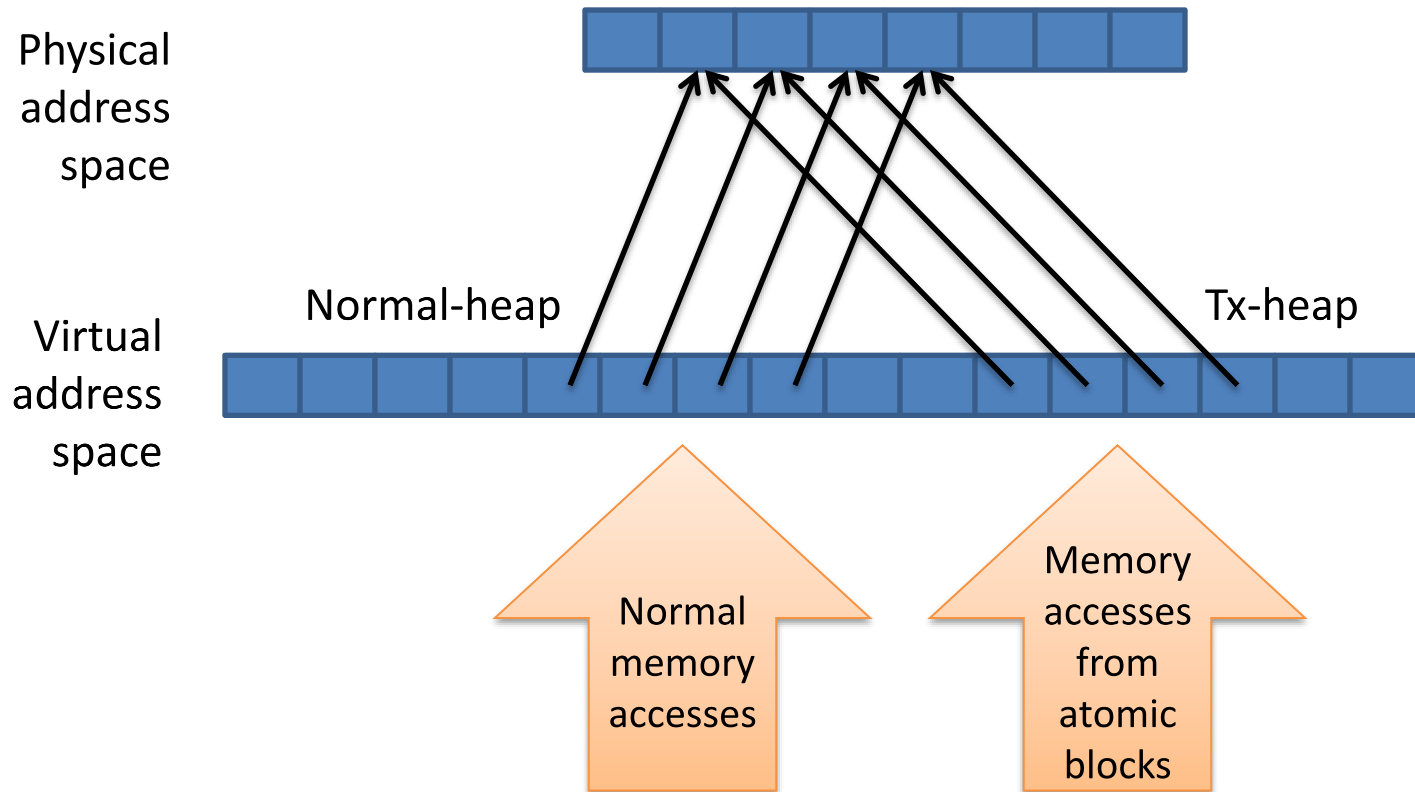
- Add new API function "ValidationTick"
- ValidationTick guarantees: it will eventually detect if its calling transaction is invalid
- Call it in any loop not otherwise calling a TM API function
- Optimize ValidationTick so it only does "real" validation occasionally
- (Could also optimize the placement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

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

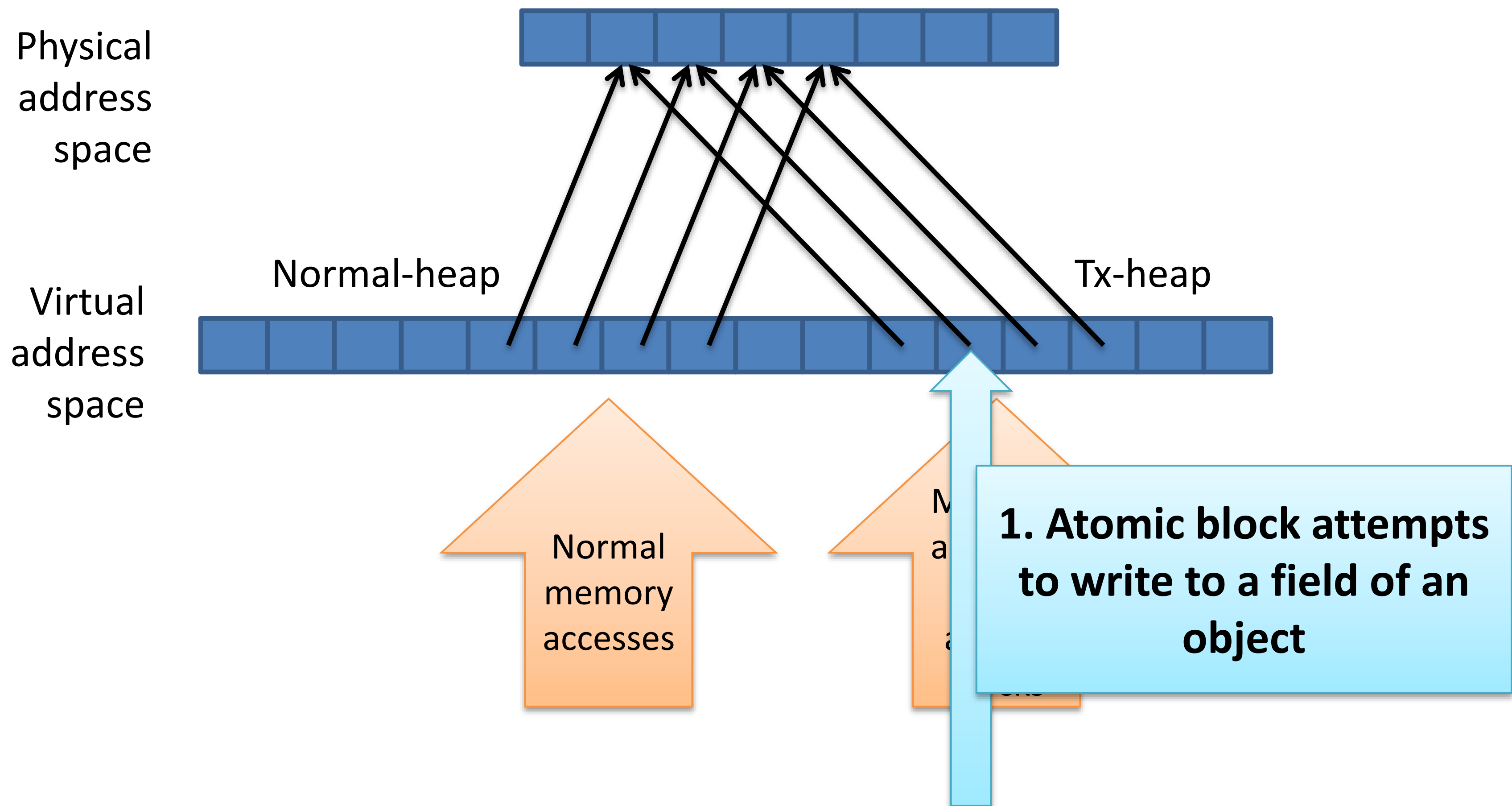
# Strong atomicity

- Add a mechanism to detect conflicts between
  tx and normal accesses
- We would like:
  - No overhead on direct accesses
  - Predictable performance
  - Little overhead over weak atomicity

# Strong atomicity: implementation

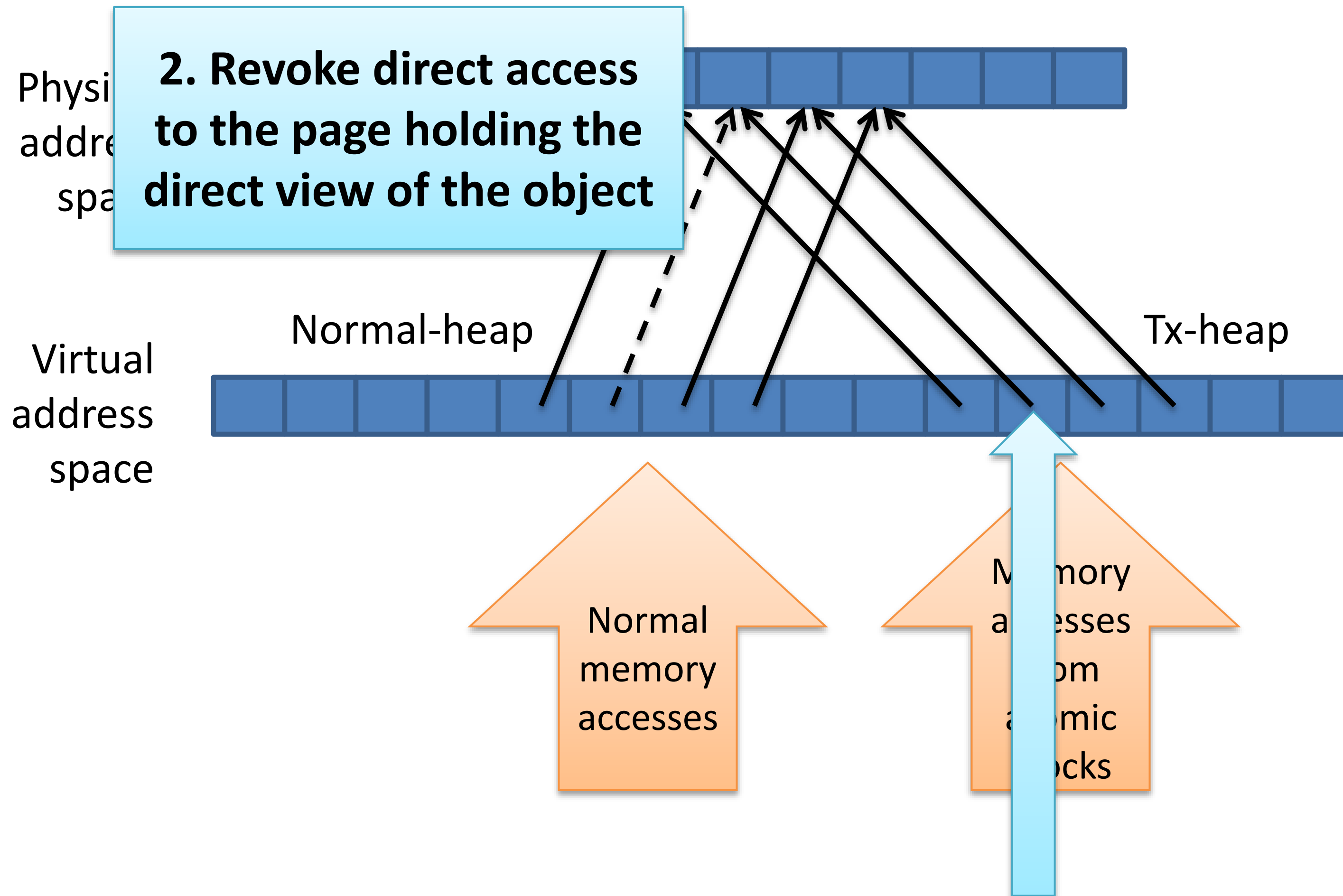


# Writes from atomic blocks

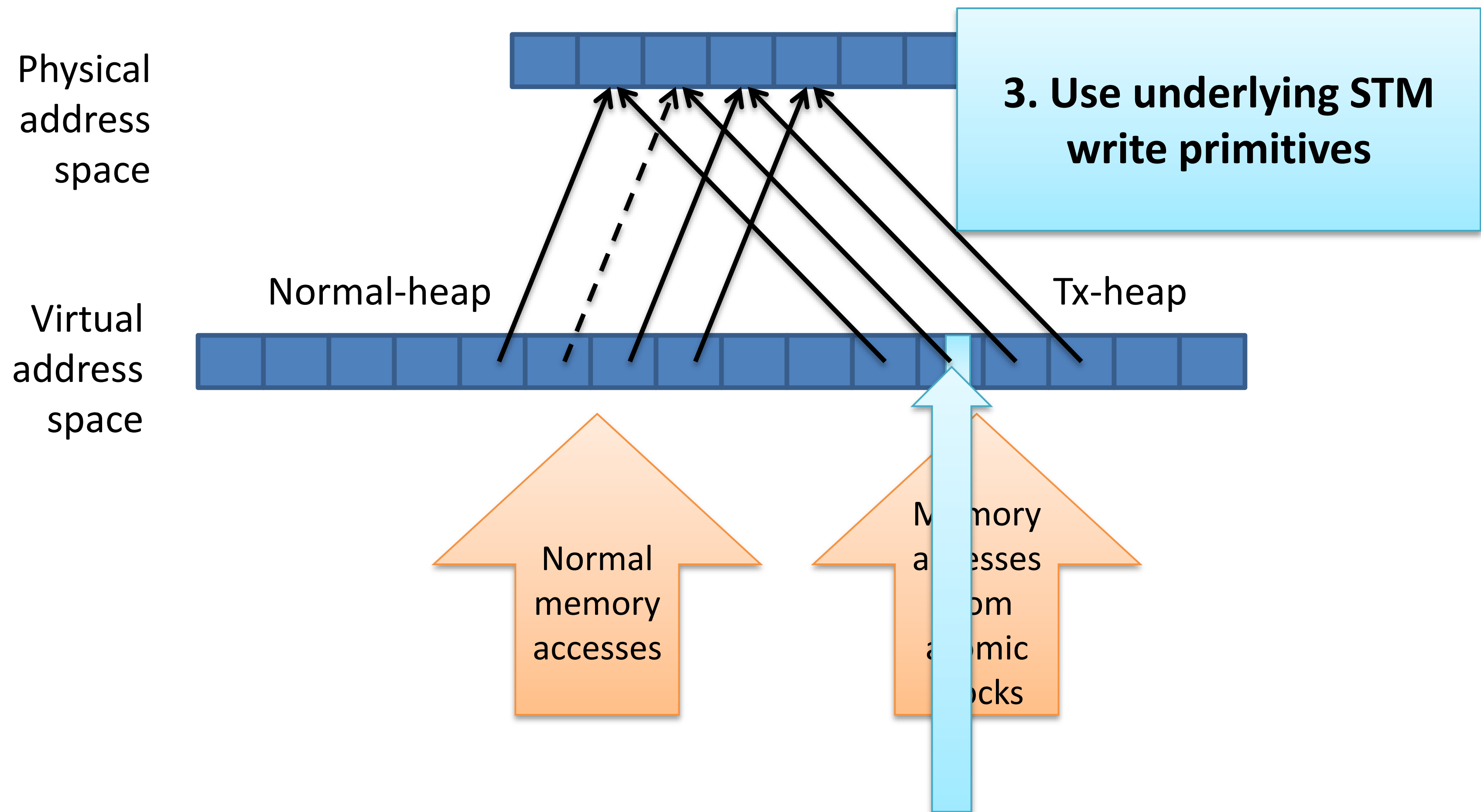


Microsoft Research

# Writes from atomic blocks



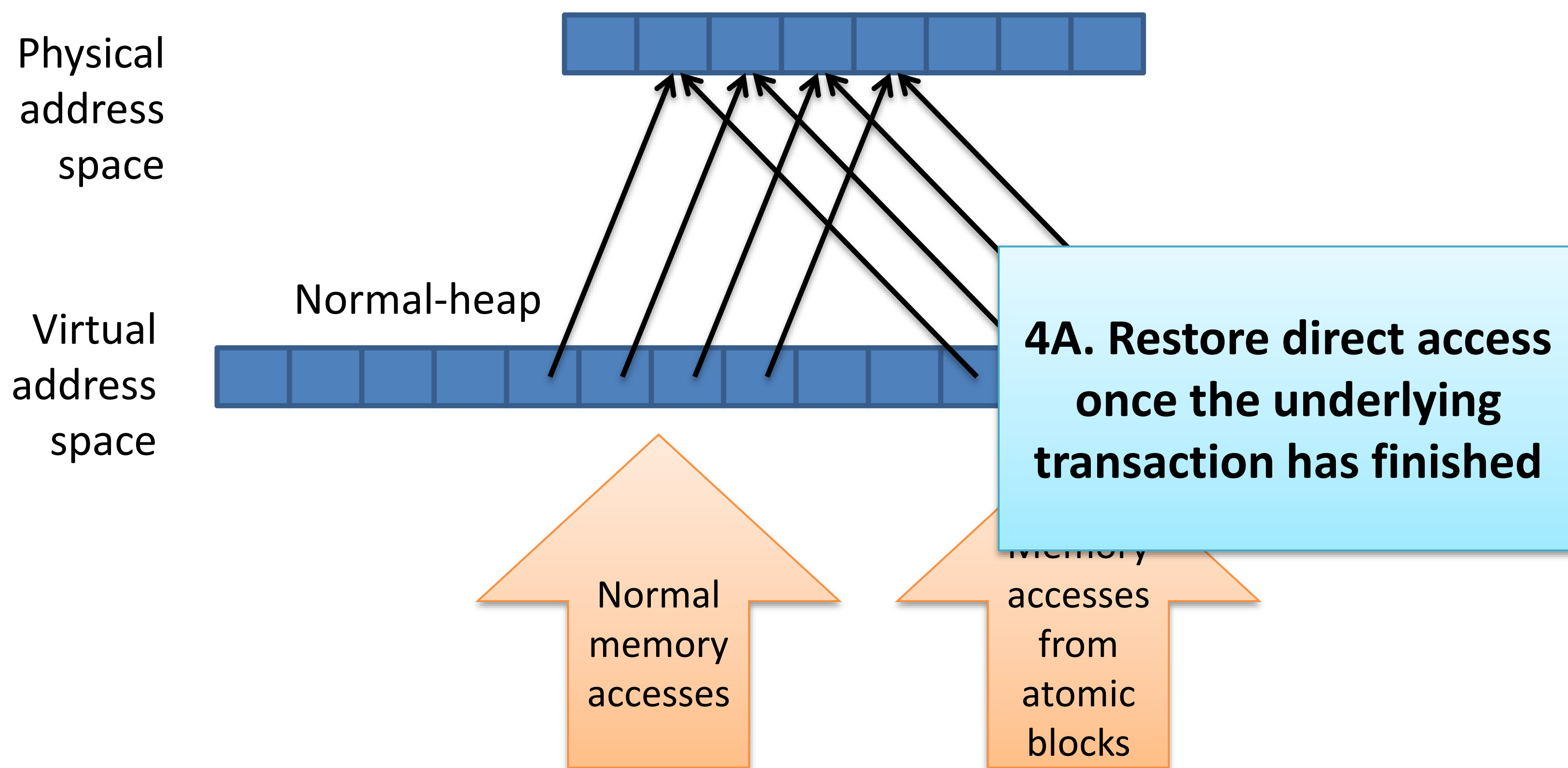
# Writes from atomic blocks



Case 1

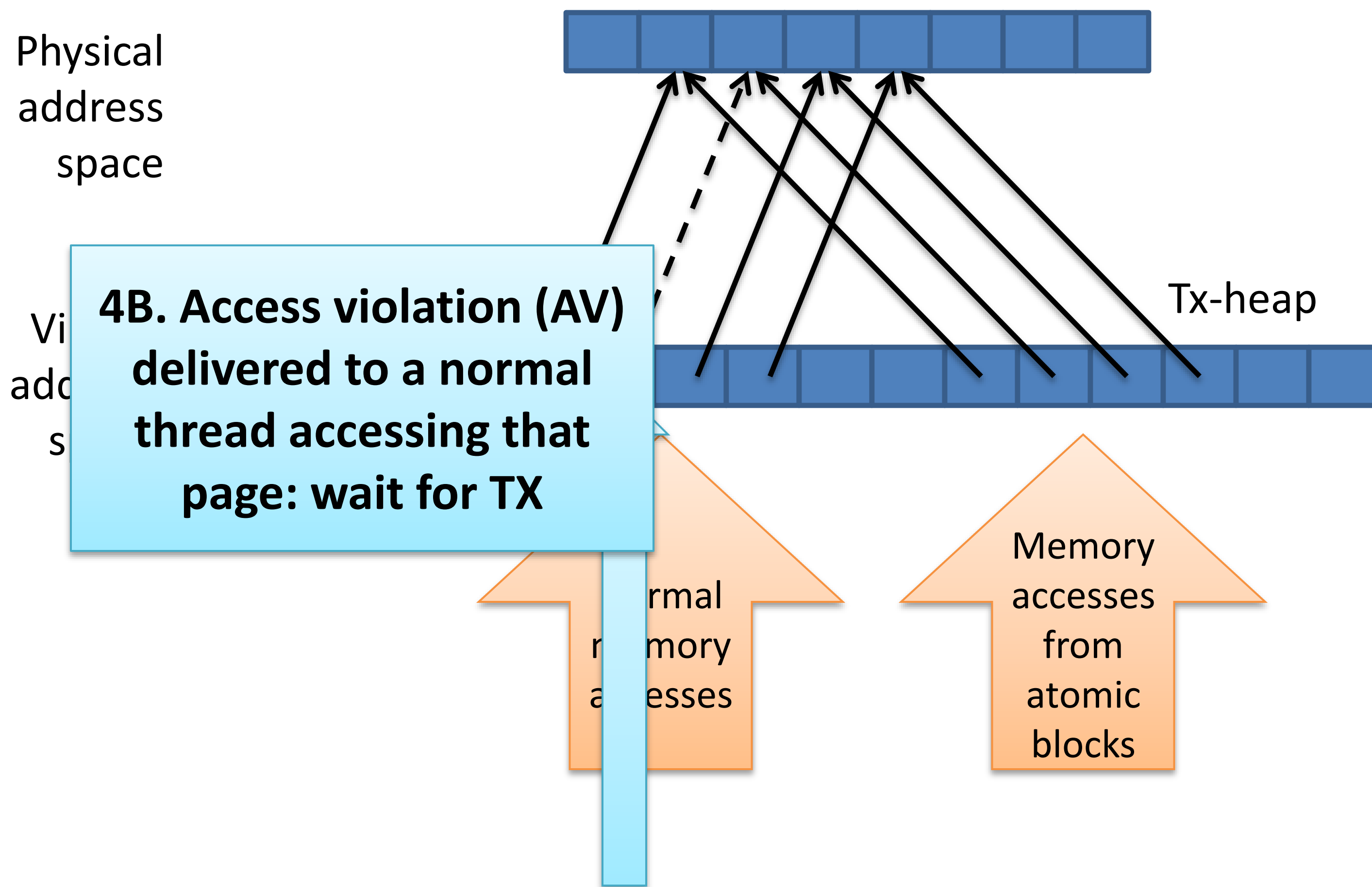
Microsoft Research

# Writes from atomic blocks



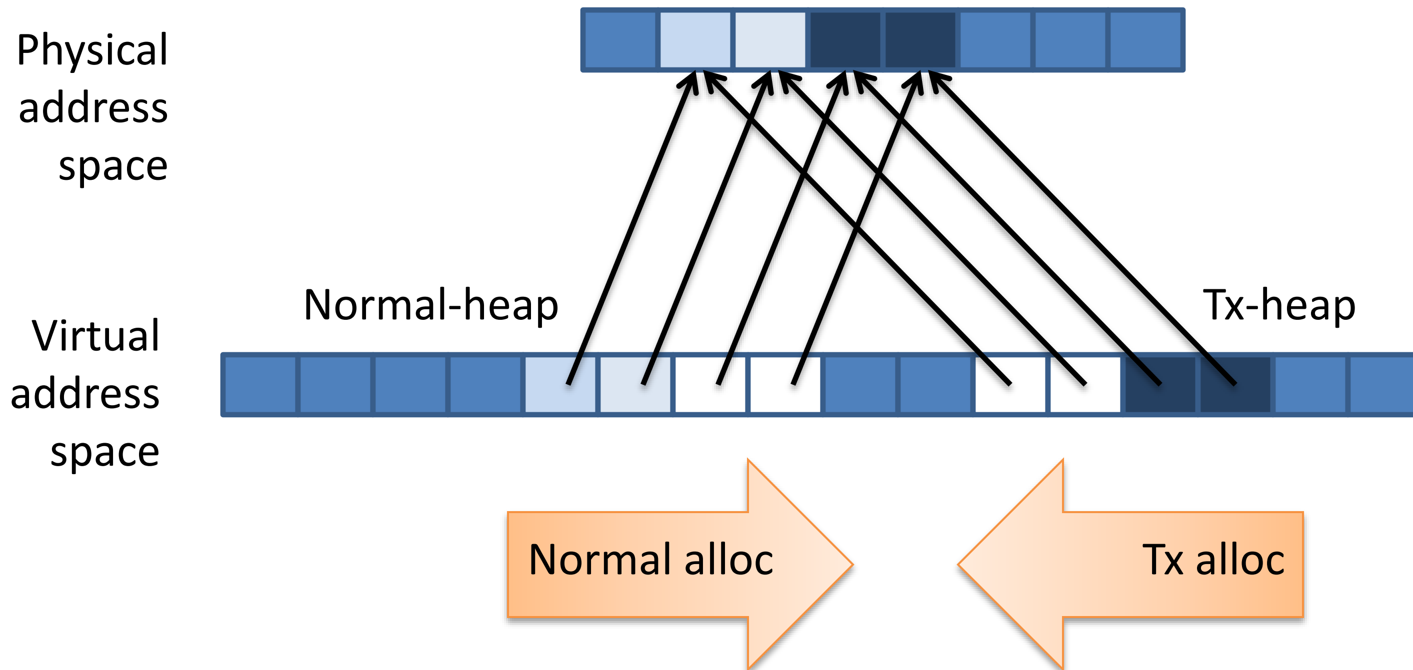
Microsoft Research

# Conflicting normal access

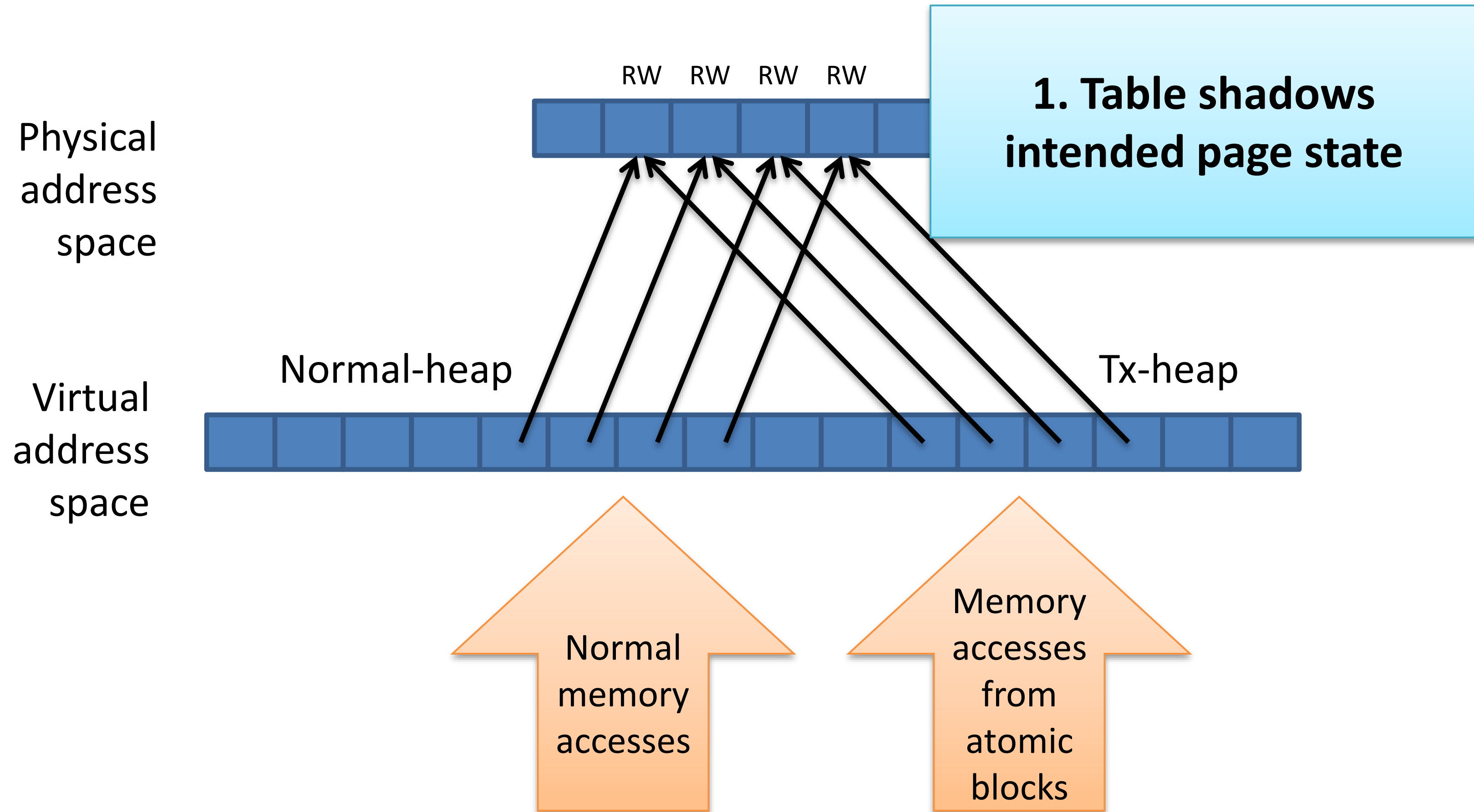


Microsoft Research

# Separate tx / non-tx allocations

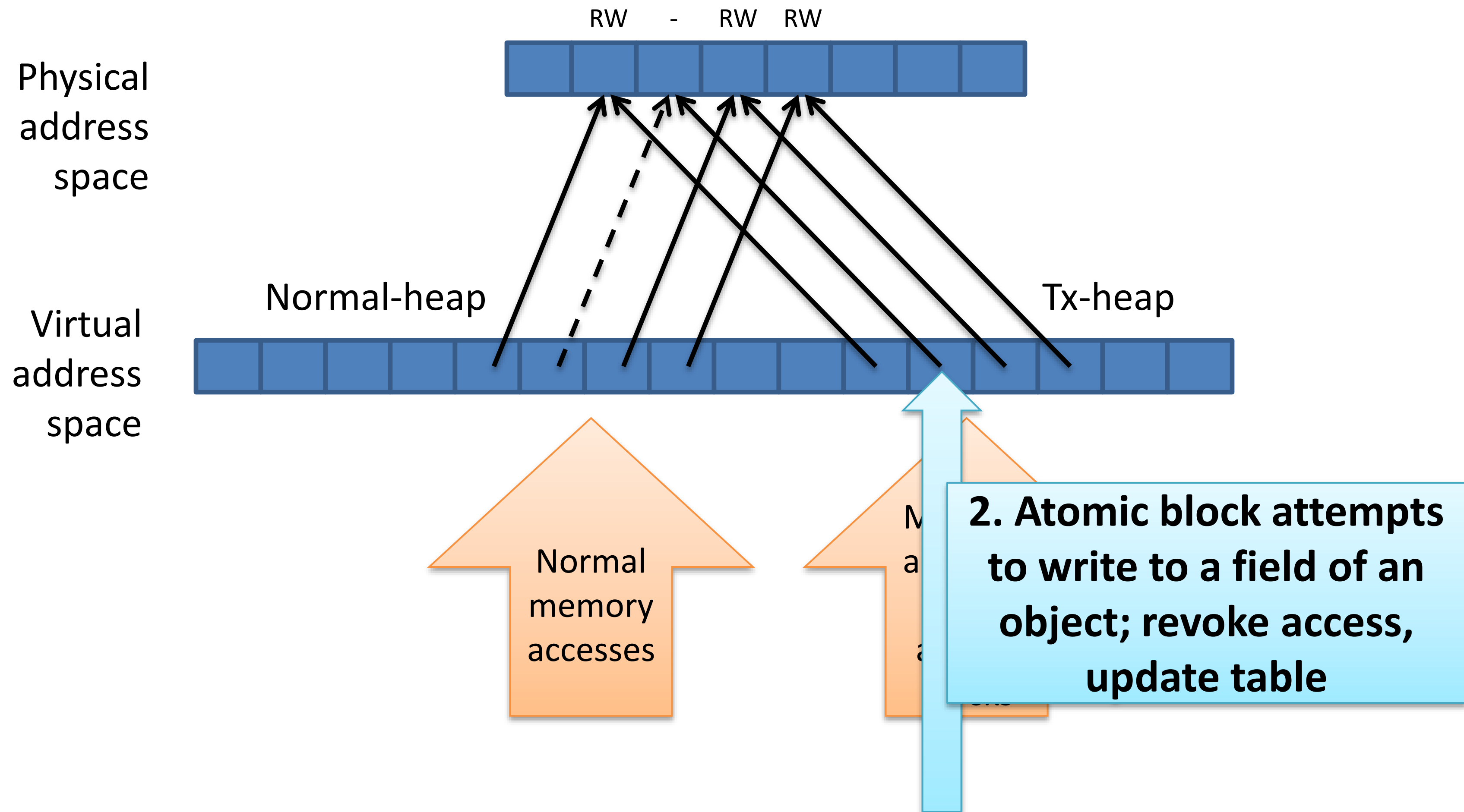


# Make page protections lazily

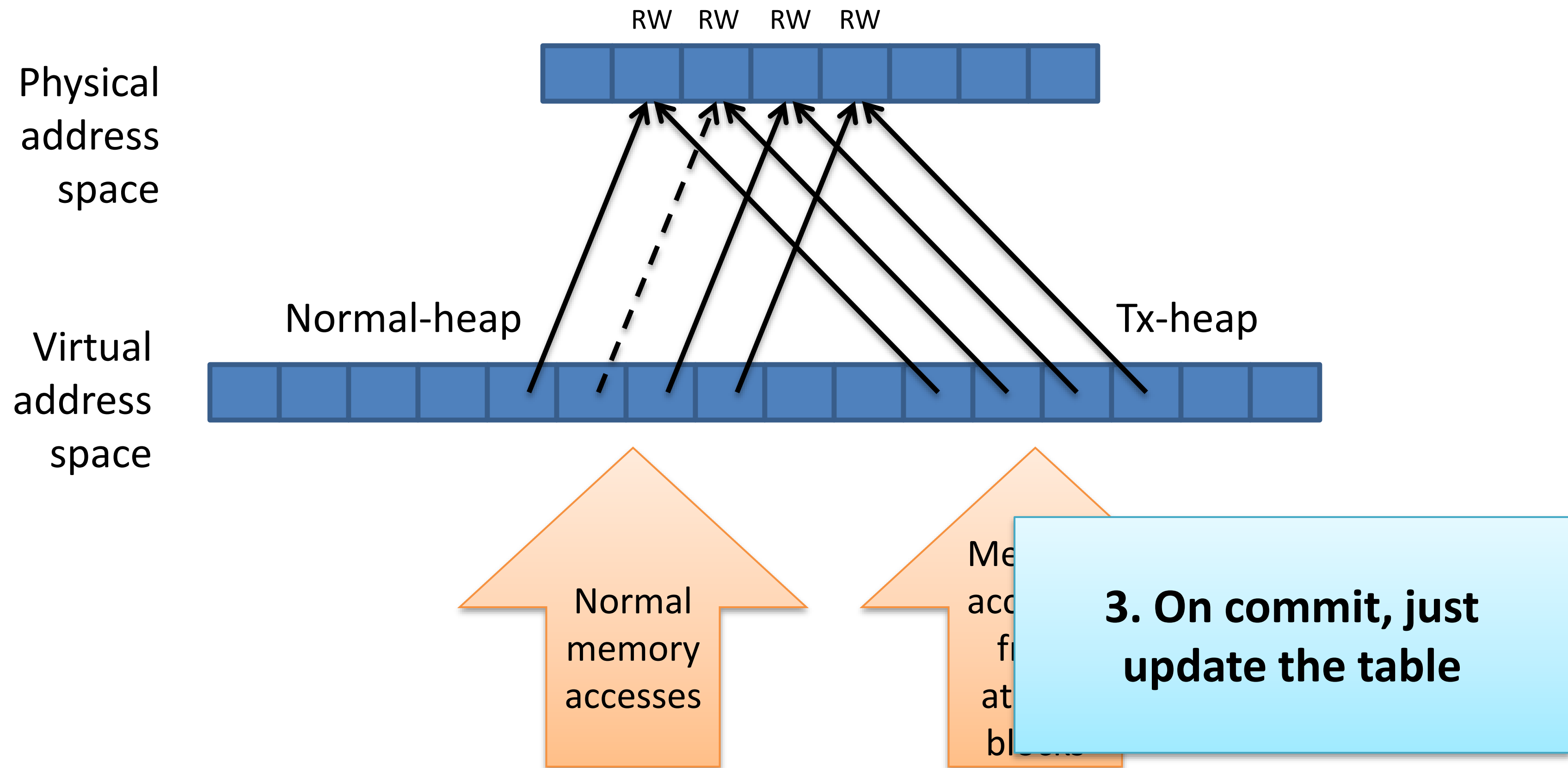



Microsoft Research

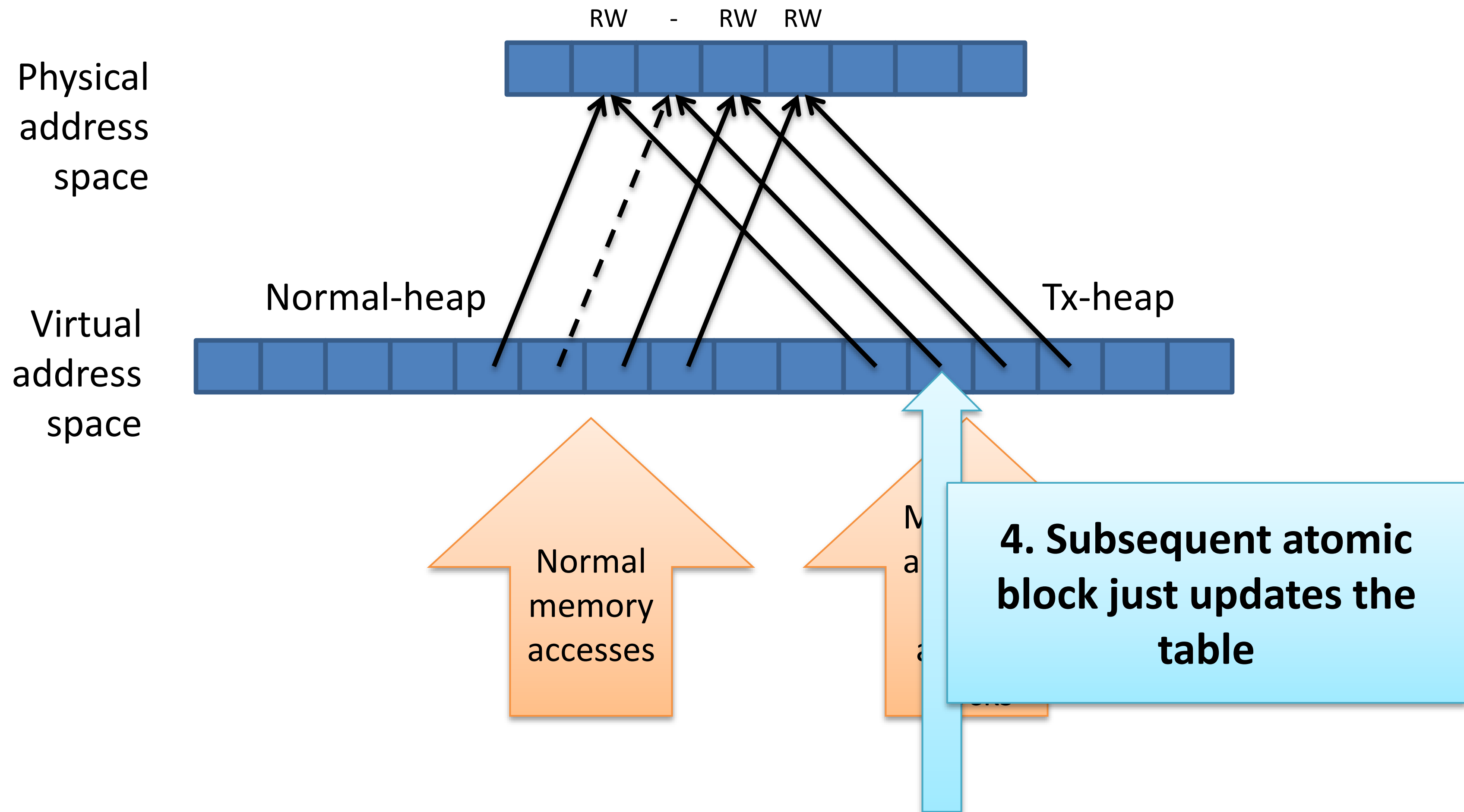
# Make page protections lazily



# Make page protections lazily



# Make page protections lazily



# Additional optimizations

- Use not-accessed-in-transaction (NAIT) analysis
- Patch instructions that trap frequently
- Avoid trapping on “safe” instructions

Atomic blocks

Low-level API & optimizations

Sandboxing zombies

Runtime-system integration

Condition synchronization

# 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 lob

# 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

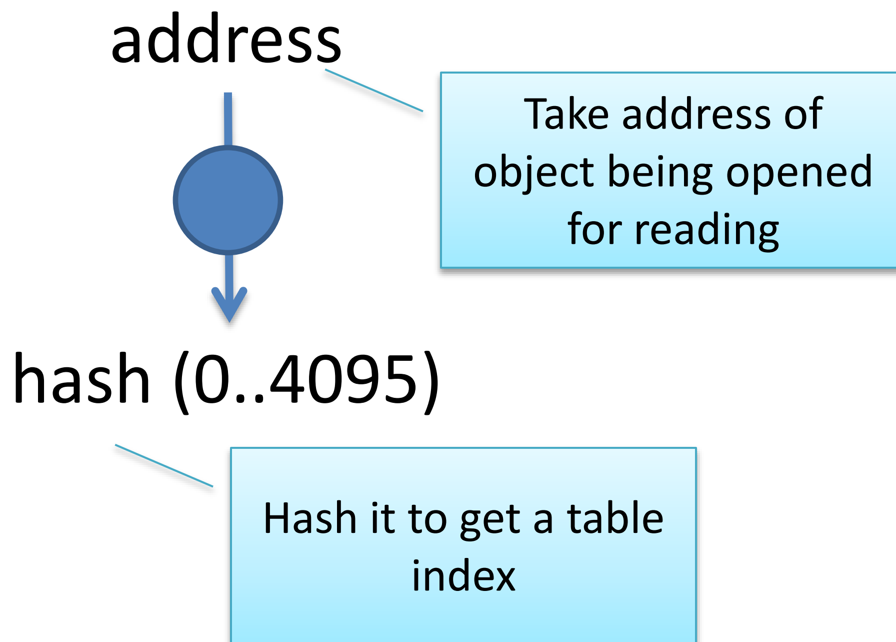
# Removing read-log duplicates

address

Take address of object being opened for reading

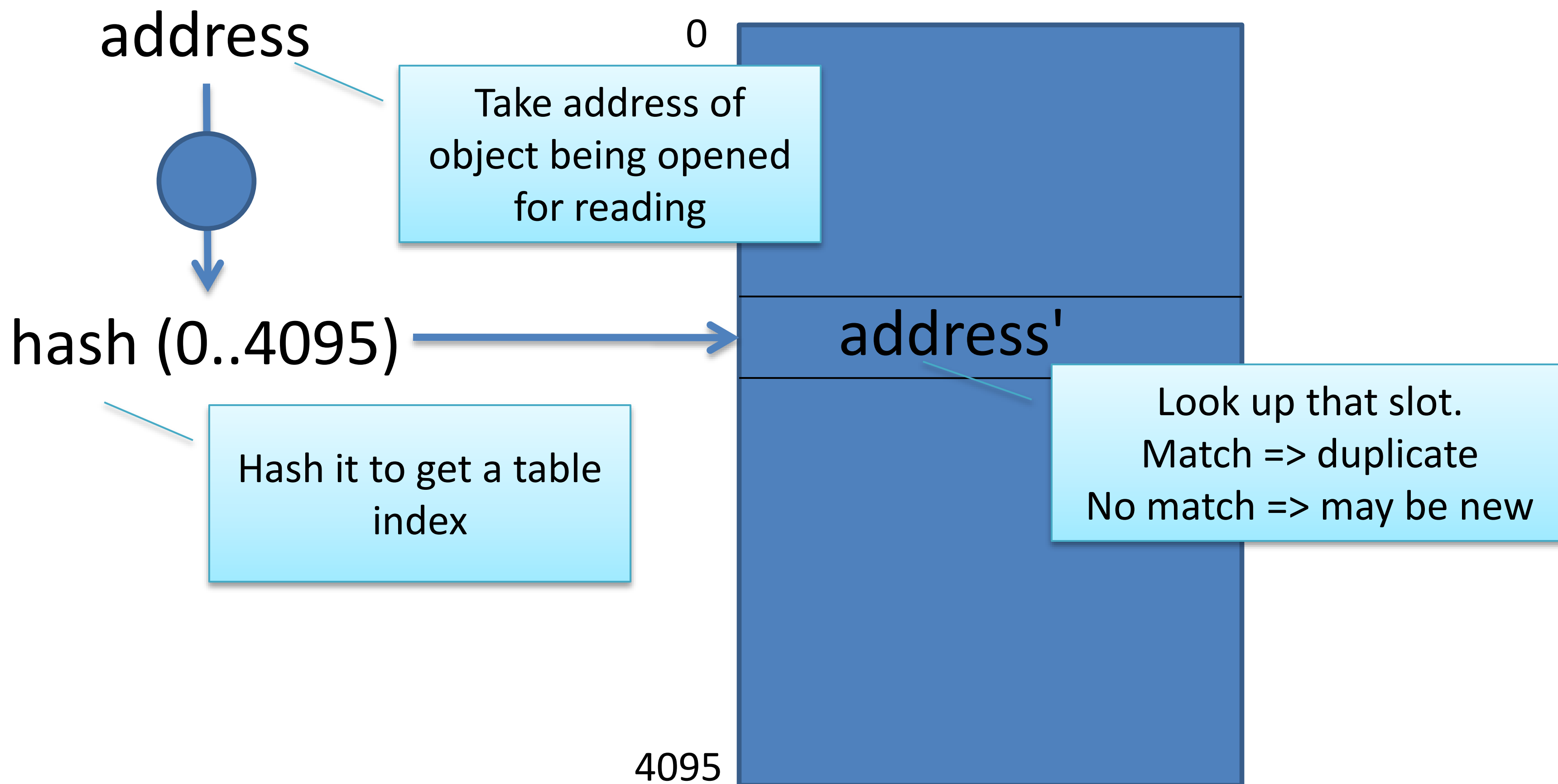
Microsoft Research

# Removing read-log duplicates

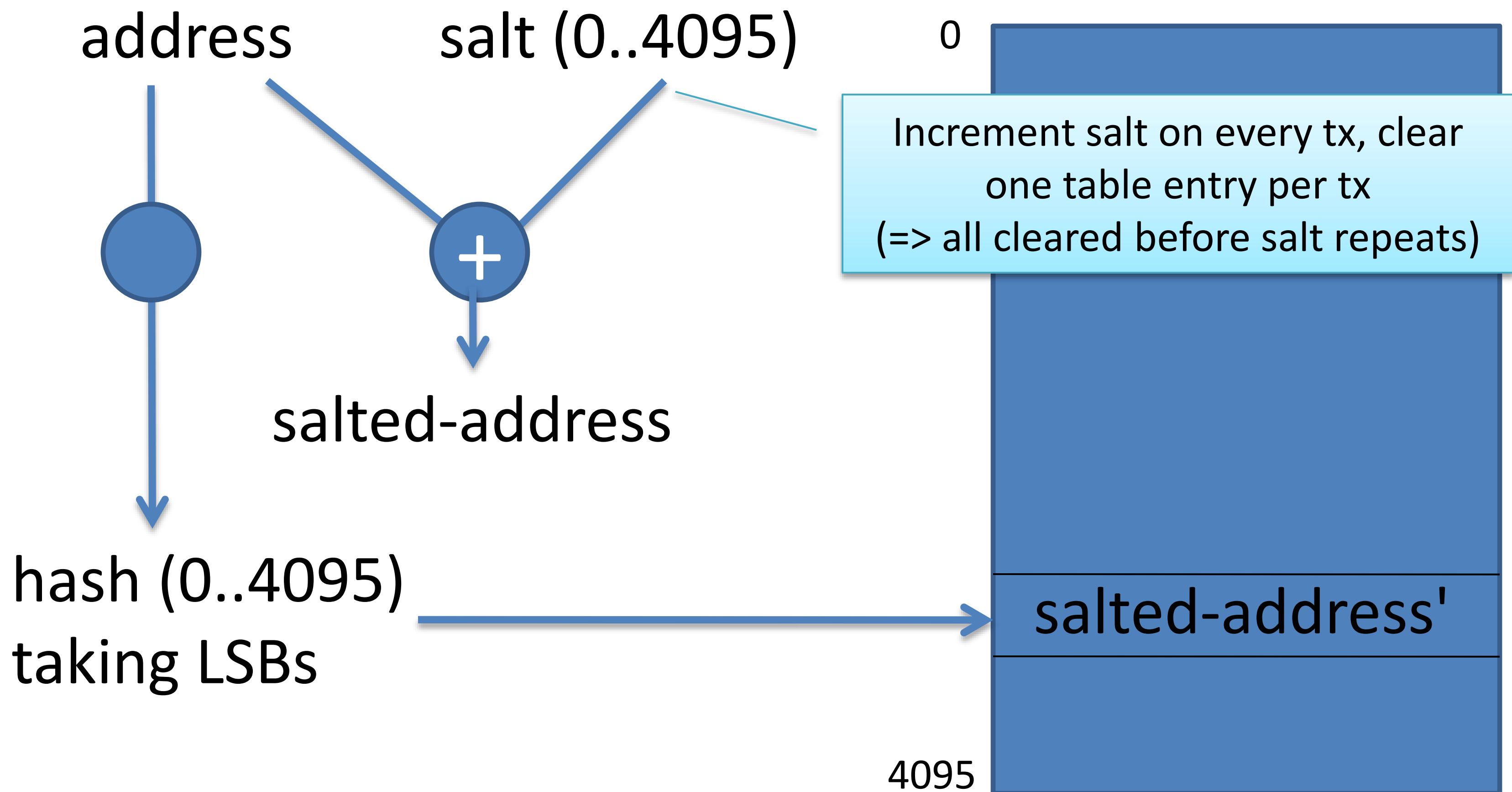


Microsoft Research

# 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

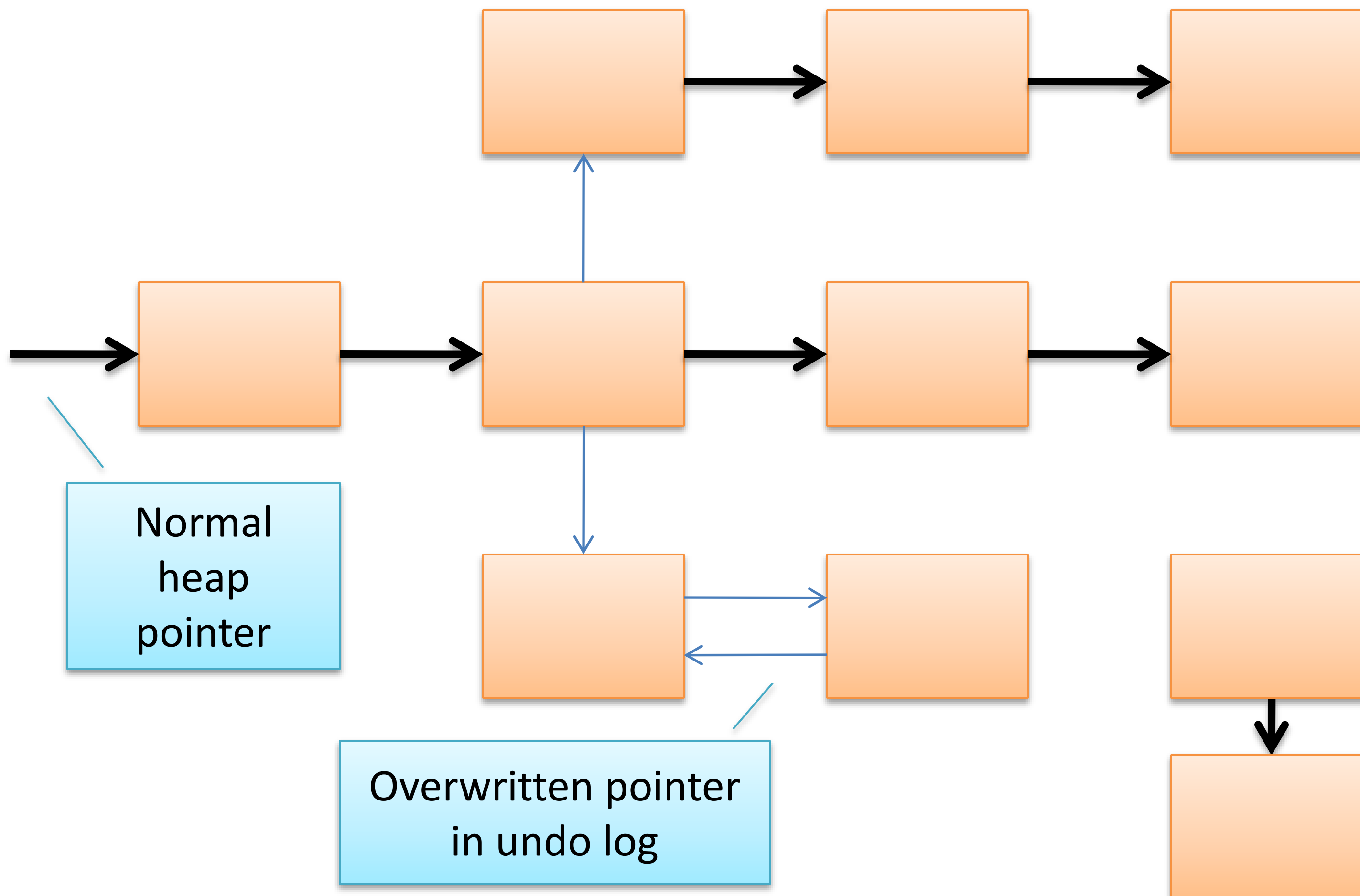
- Abort all running tx on GC?
  - Not ideal: long running tx will not be able to commit
  - Is there a precedent for language features with this kind of perf?
  - (We also rely on GC as a fall-back for duplicate log elimination and to force the validations needed for safe version number overflow)
- Treat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

Microsoft Research

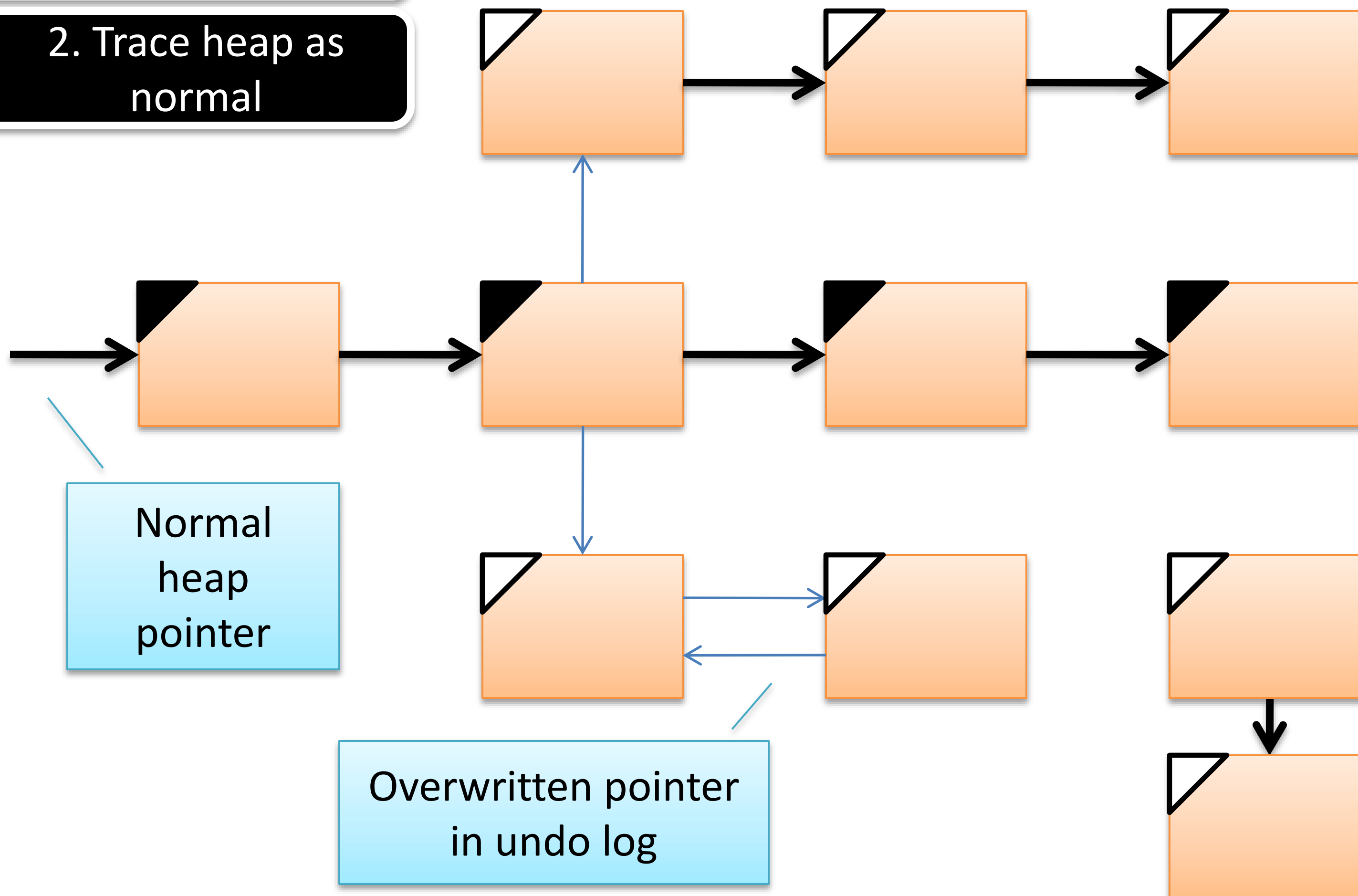
# GC integration

- 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

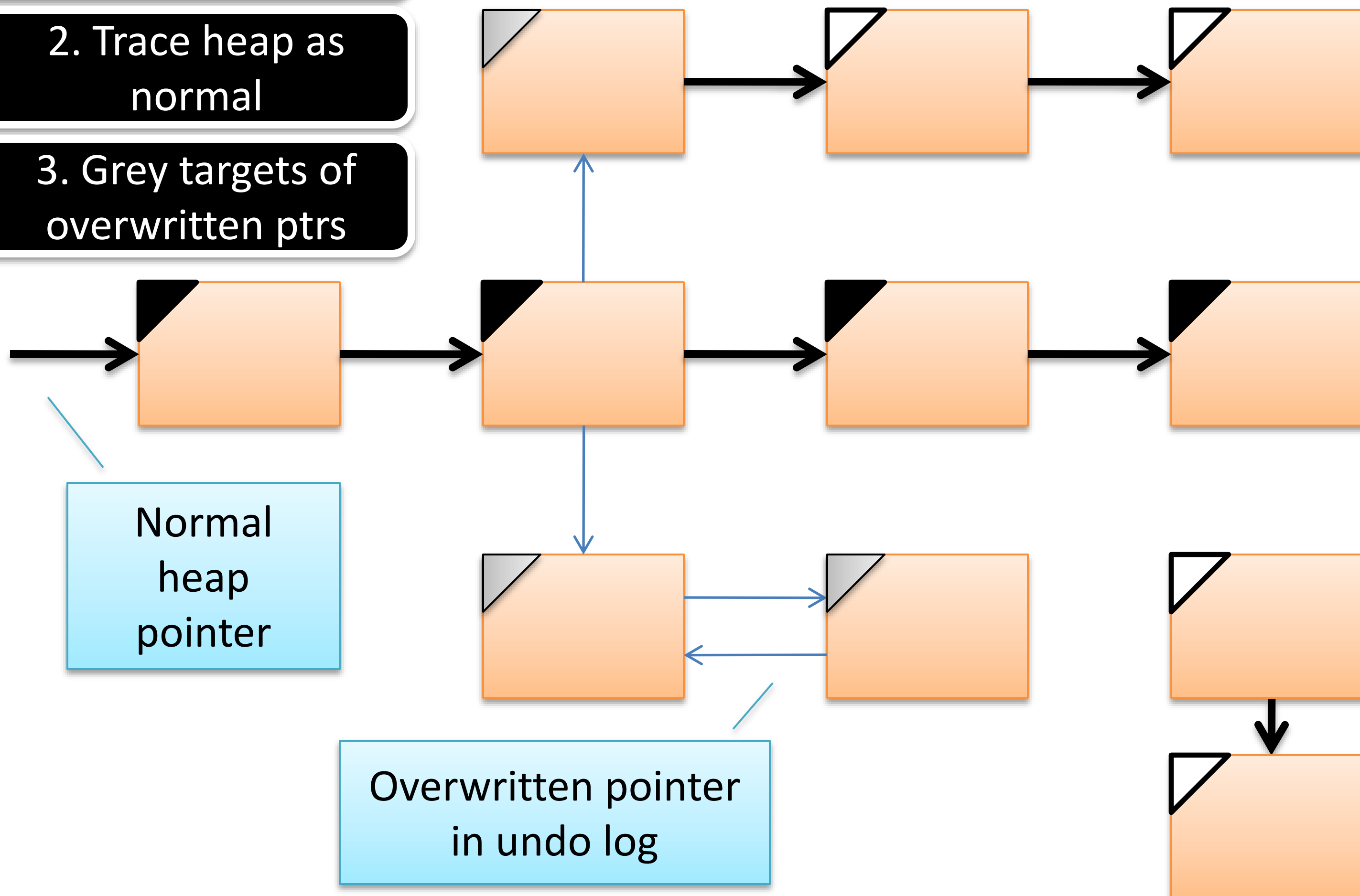
# Example heap



# Conservative algorithm



# Conservative algorithm



# Conservative algorithm

1. Validate tx
2. Trace heap as normal
3. Grey targets of overwritten ptrs
4. Trace from new grey objects

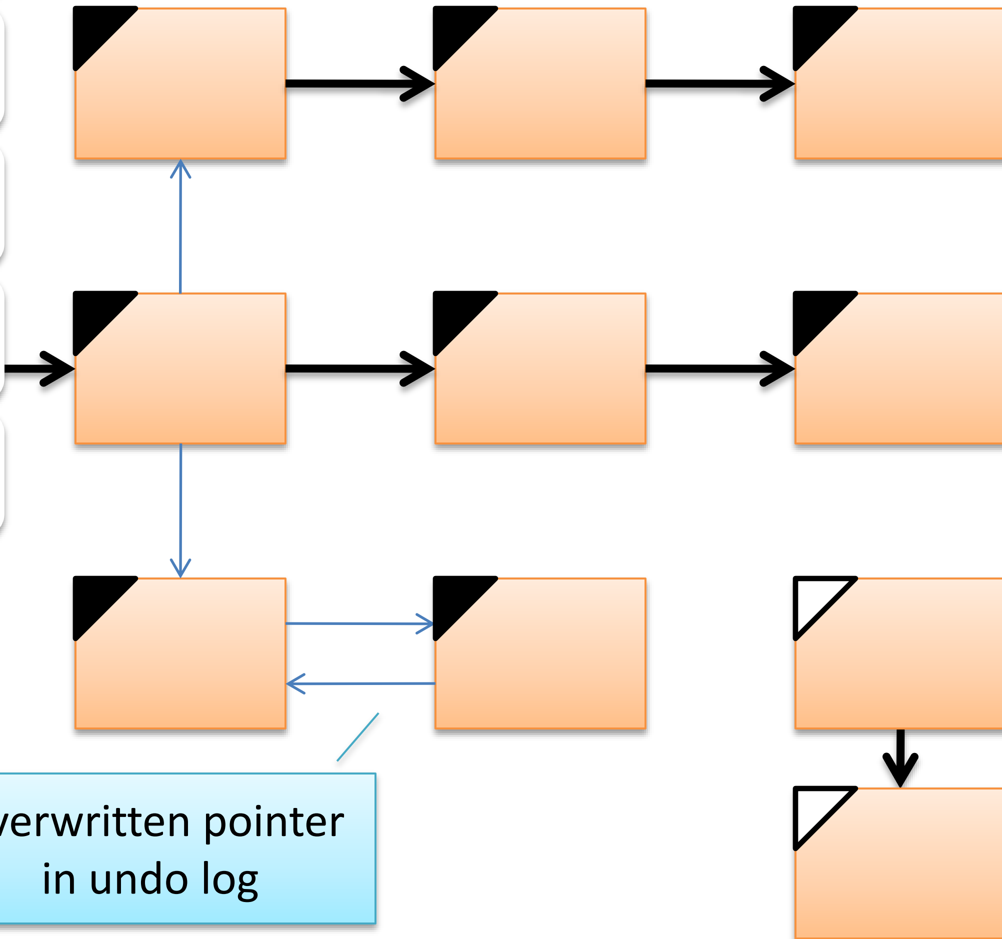
Normal heap pointer

Overwritten pointer in undo log

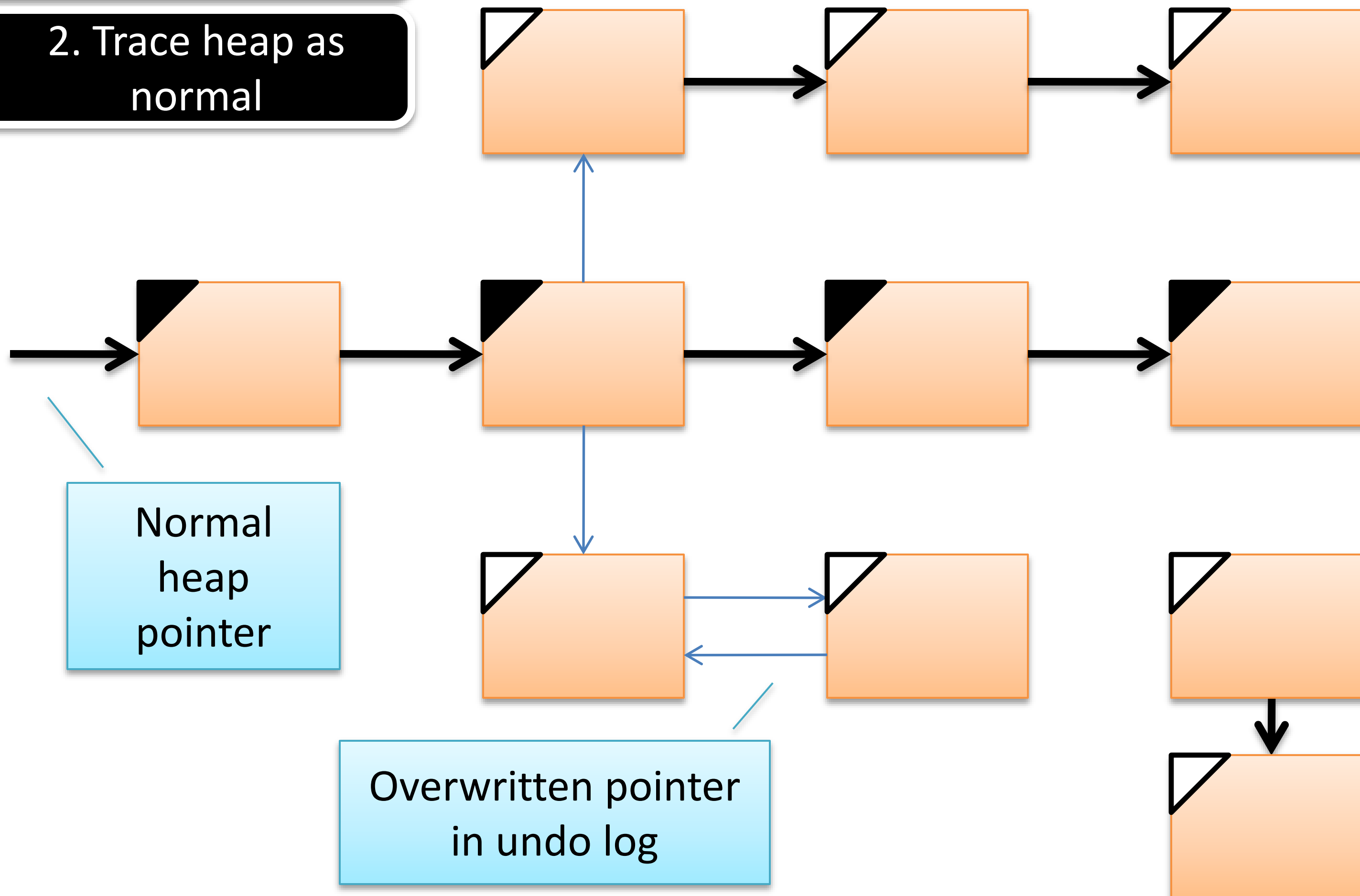
# Conservative algorithm

1. Validate tx
2. Trace heap as normal
3. Grey targets of overwritten ptrs
4. Trace from new grey objects
5. Reclaim white objects

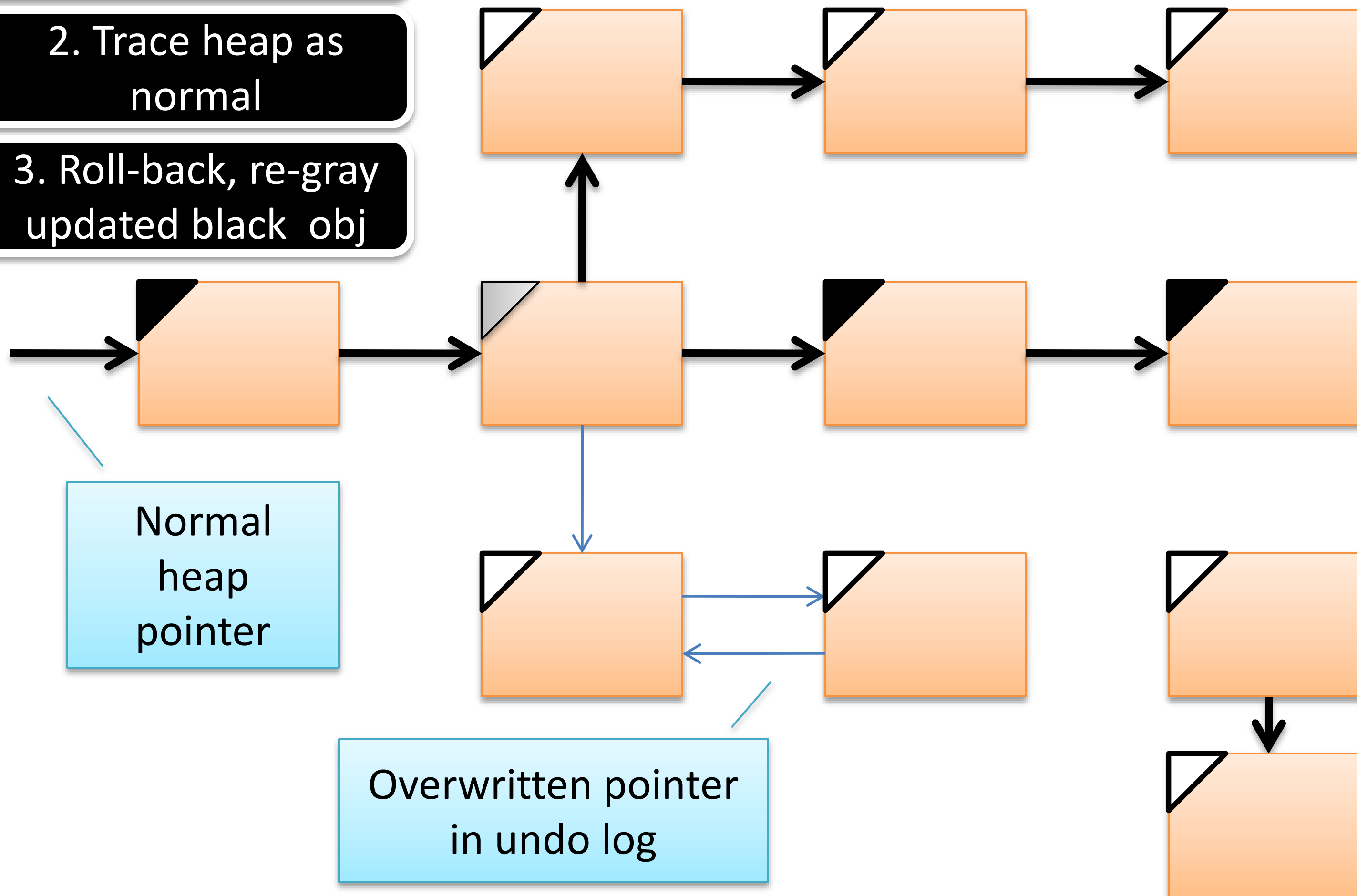
Normal heap pointer



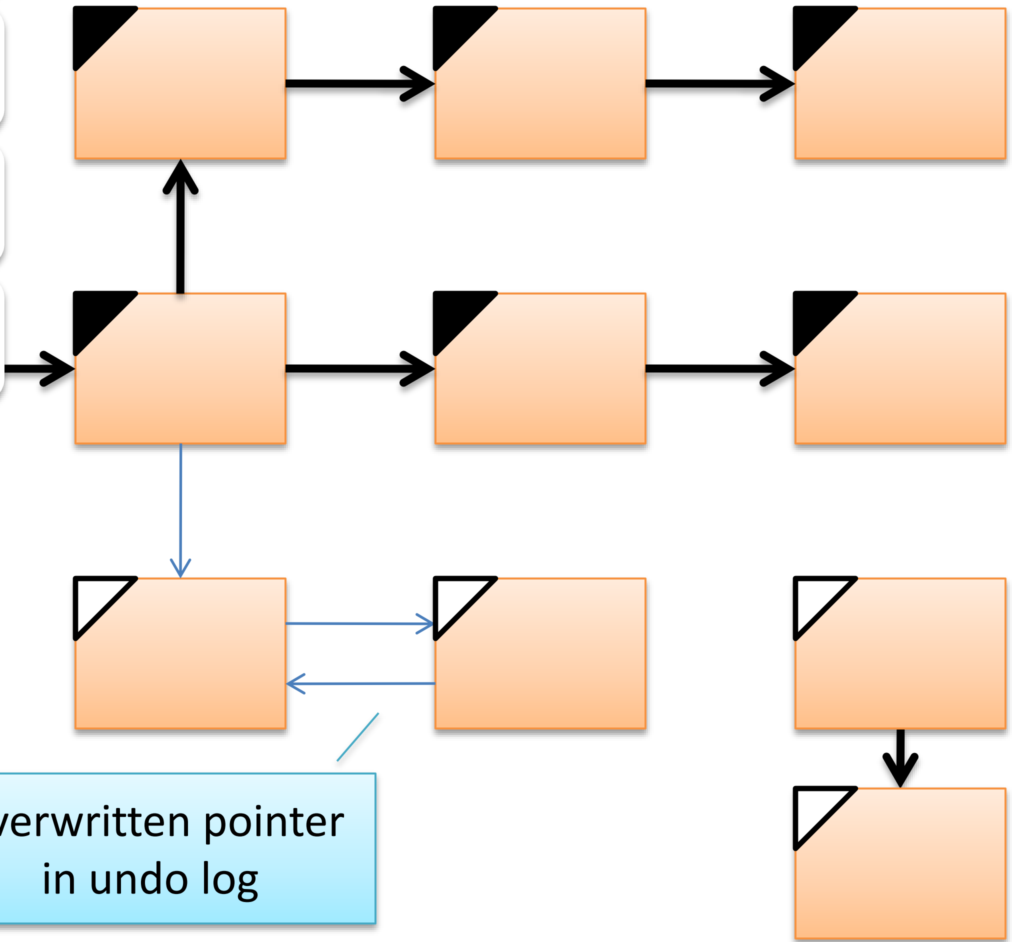
# Precise algorithm



# Precise algorithm



# Precise algorithm



# Precise algorithm

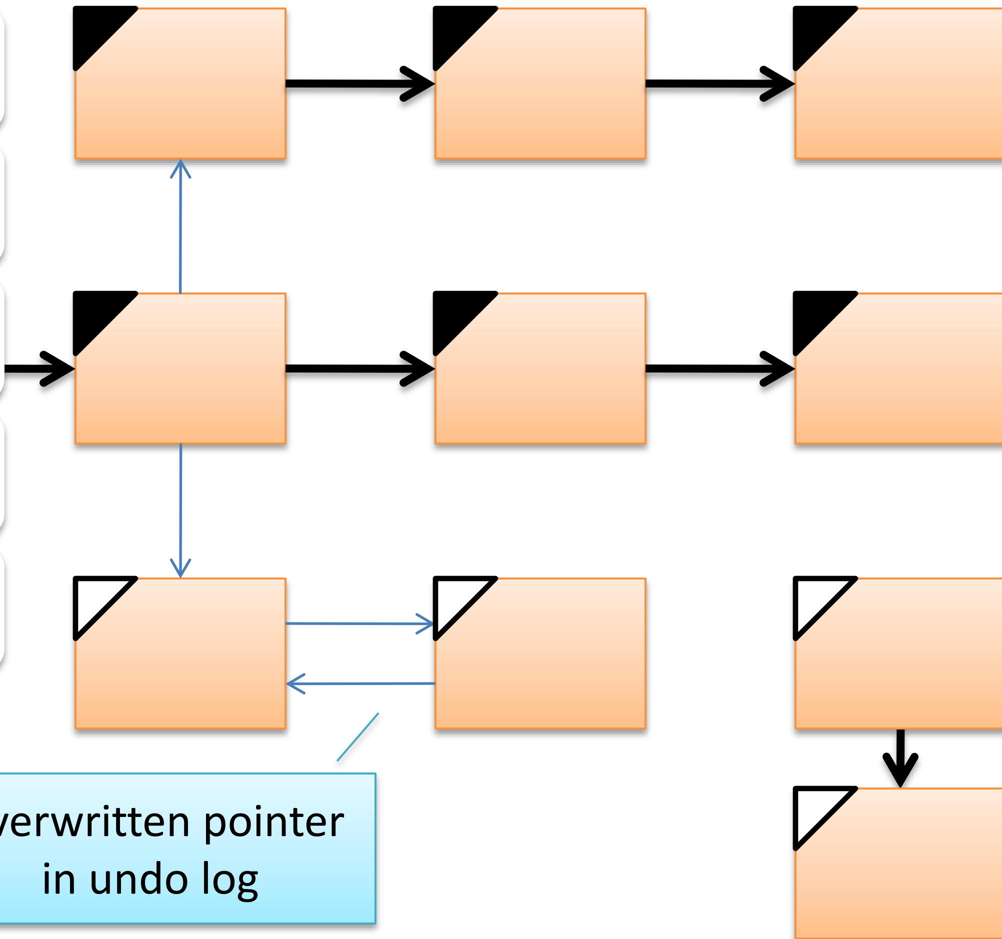
1. Validate tx
2. Trace heap as normal
3. Roll-back, re-gray updated black obj
4. Trace from gray objects
5. Reclaim white objects

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

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

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

# 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

Atomic blocks

Low-level API & optimizations

Sandboxing zombies

Runtime-system integration

**Condition synchronization**

# Condition synchronization

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

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

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

- 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.fu
    retry;
  }
  result = buffe
  buffer.full = f
}
```

```
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

buffer:

| v150 |
| --- |
| Val=0 |
| Full=false |

# Implementing WaitTx

1. Extend object header with list of waiters

buffer:

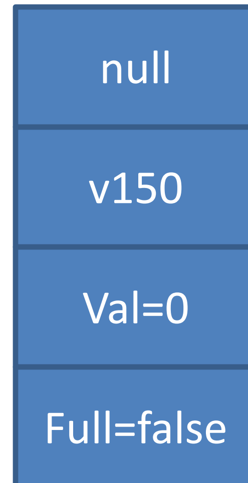
| null |
|---|
| v150 |
| Val=0 |
| Full=false |

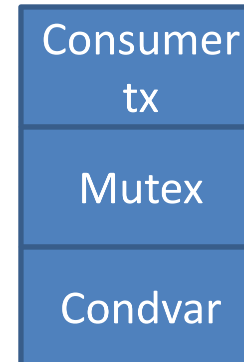
Microsoft Research

# Implementing WaitTx

1. Extend object header with list of waiters

2. Extend tx records with a mutex & condvar pair



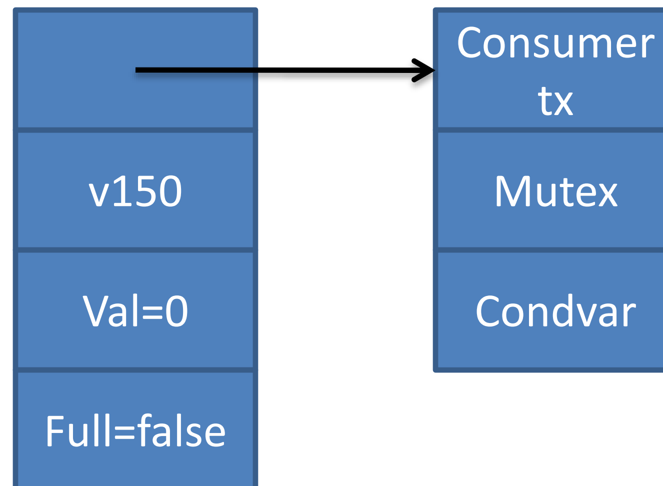



# Implementing WaitTx

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



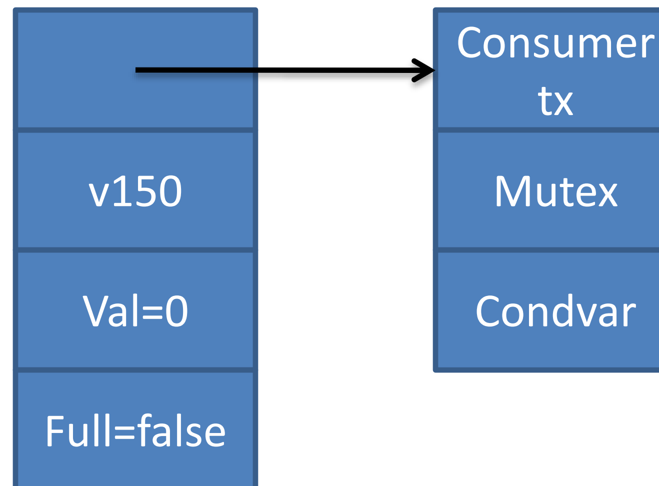
# Implementing WaitTx

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

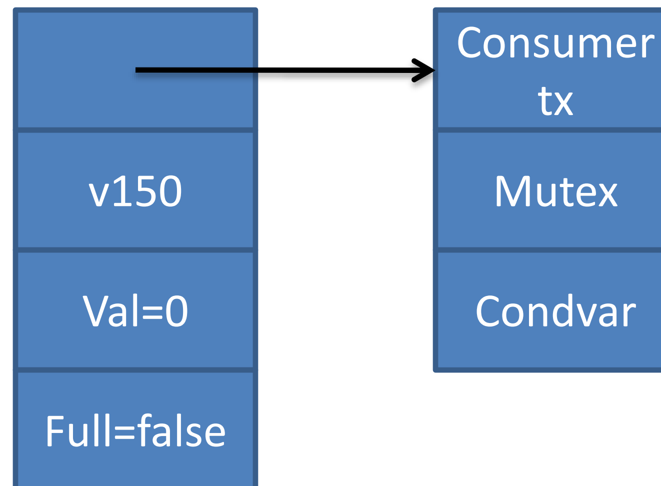
1. Extend object header with list of waiters

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



# Imp… WaitTx

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

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

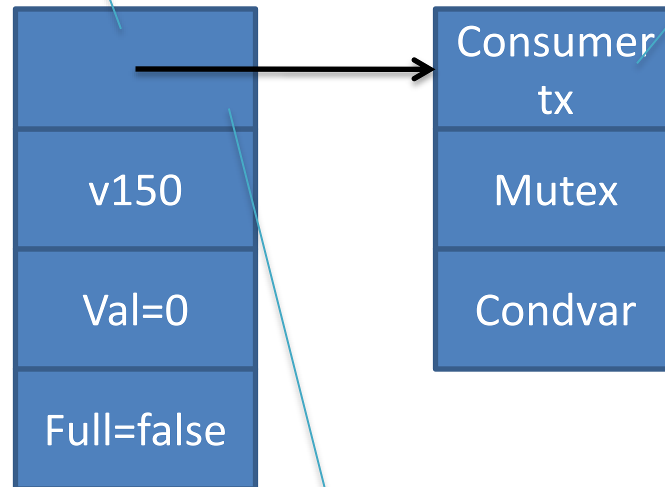
1. Extend object header with list of waiters

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 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.*