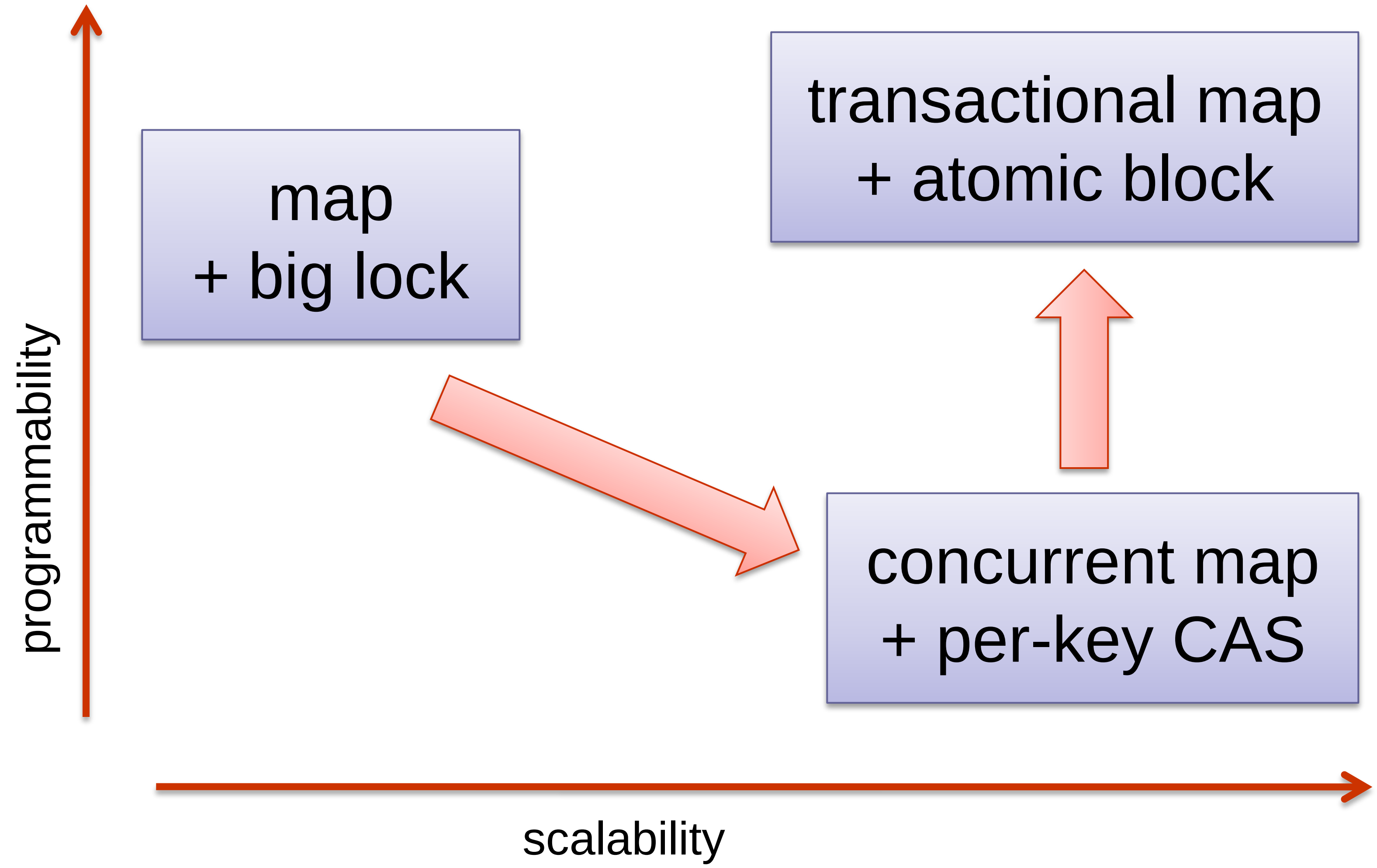
# Transactional Predication: High-Performance Concurrent Sets and Maps for STM

Nathan G. Bronson, Jared Casper,
Hassan Chafi, Kunle Olukotun

*Stanford CS*

PODC - 26 July 2010

# Thread-safe shared maps



map
+ big lock

transactional map
+ atomic block

concurrent map
+ per-key CAS

# What I'd like

```
m = new TransactionalHashMap

v = m.get(key)
m.put(key, pureFunc(key))

atomic {
  prev = m.remove(key1)
  m.put(key2, prev)
}

atomic {
  fwd.put(name, phoneNumber)
  reverse.put(phoneNumber, name)
}

atomic {
  m.get(k).observers += self
}
```

fast access
**outside** a txn

atomic access to
multiple **keys**

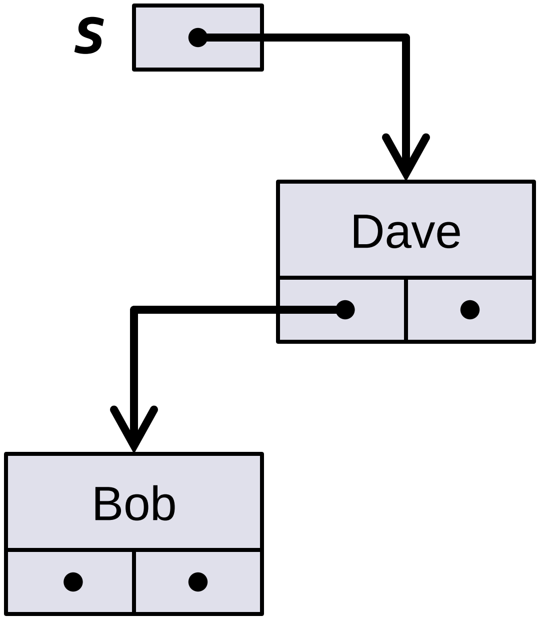
atomic access to
multiple **maps**

**composes** with STM
reads and writes

# Why not just code a map using STM?

- Single-thread overheads
  - Each map op requires multiple STM reads/writes
    - *Reads of shared data must be validated*
    - *Writes to shared data must be logged or buffered*
  - Non-transactional map ops must start a transaction
    - *Even though composition is not required!*
- Scalability limits
  - Not all structural conflicts are semantic conflicts
  - More threads → false conflicts more frequent
  - Bigger txns → false conflicts more wasteful

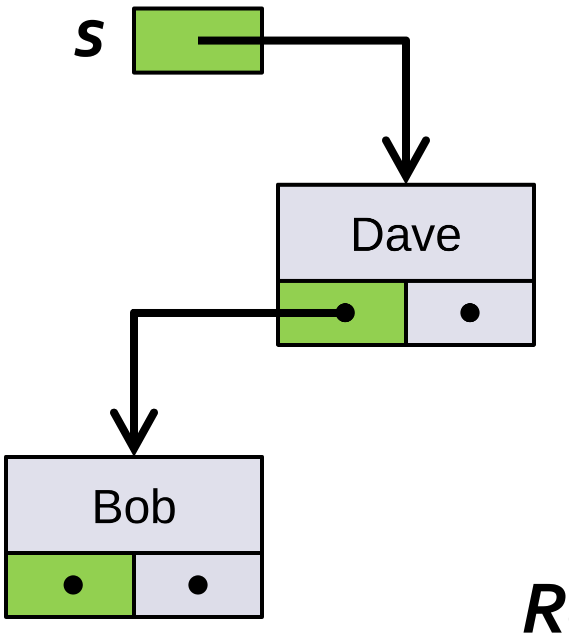
# STM challenges: overheads



```
s = { 'Bob, 'Dave }

atomic {
    s.contains('Alice)
}
```

# STM challenges: overheads



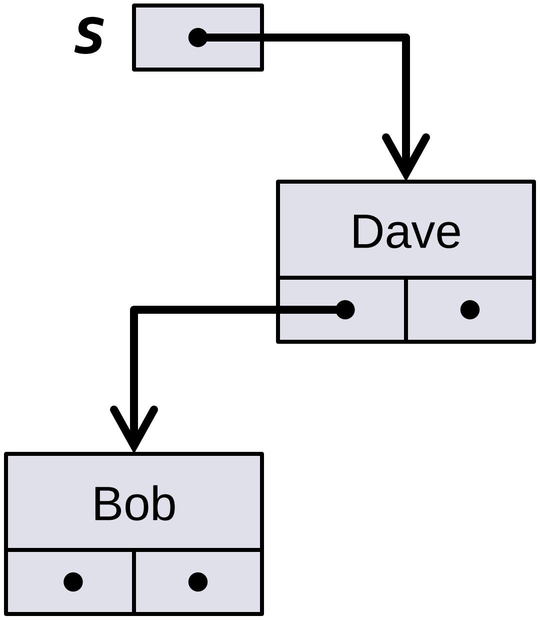
**s** = { ’Bob, ’Dave }

```
atomic {
    s.contains(’Alice)
}
```

*Read set contains 3 entries*

*A transcaction is required for even a solitary non-transactional access*

# STM challenges: false conflicts

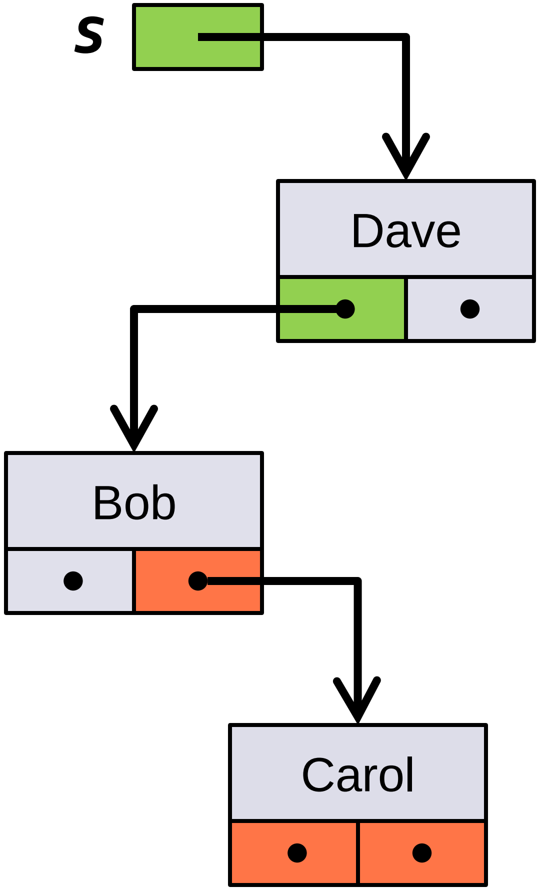


**s** = { ’Bob, ’Dave }

*ThreadA:* **atomic {**
    **s**.contains(’Alice)
**}**

*ThreadB:* **atomic {**
    **s**.add(’Carol)
**}**

# STM challenges: false conflicts

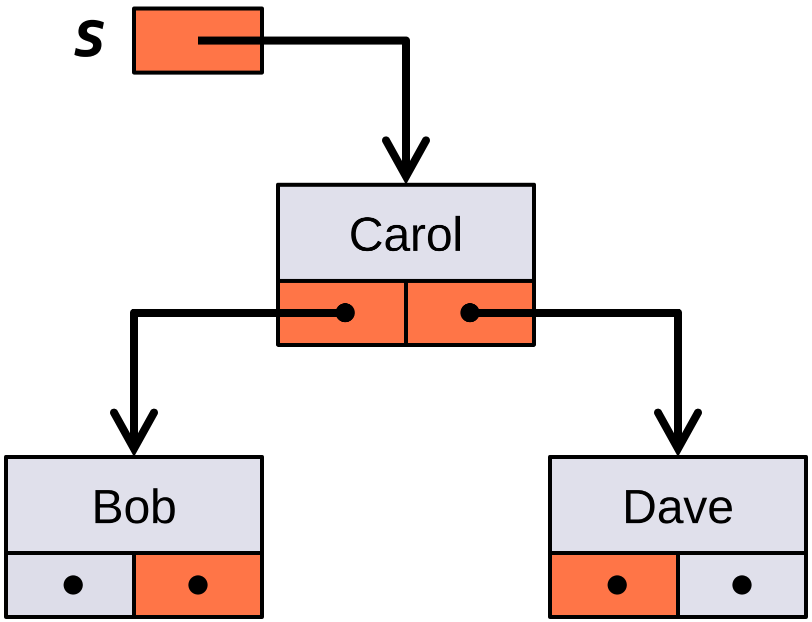


**s** = { 'Bob, 'Dave }

```
ThreadA: atomic {
    s.contains('Alice)
}

ThreadB: atomic {
    s.add('Carol)
}
```

# STM challenges: false conflicts



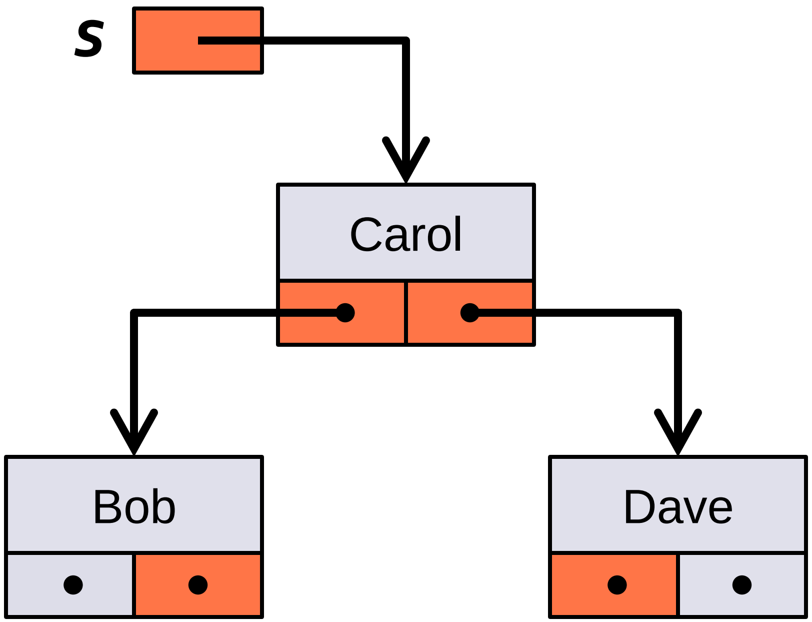
**s** = { ’Bob, ’Dave }

~~*ThreadA:* **atomic {**~~
~~**s**.contains(’Alice)~~
~~**}**~~

*ThreadB:* **atomic {**
**s**.add(’Carol)
**}**

**contains(‘Alice)** ***and*** **add(‘Carol)** ***are semantically disjoint, but have a structural conflict***

# STM challenges: false conflicts



**s** = { ’Bob, ’Dave }

~~*ThreadA:* **atomic {**~~
~~**s**.contains(’Alice)~~
~~lotsOfWork()~~
**}**
*ThreadB:* **atomic {**
**s**.add(’Carol)
**}**

**contains(’Alice)** ***and*** **add(’Carol)** ***are semantically disjoint, but have a structural conflict***

# Are all the STM accesses required?

- The read or write of a single memory location corresponds to accessing the set’s abstract state
  - contains(’Alice) → `bob.left.stmRead()`
  - add(’Carol) → `bob.right.stmWrite(...)`
- Additional reads and writes are required to navigate to that location and maintain the data structure
- Overheads and false conflicts come mainly from the **navigating** and **maintenance** accesses

*We should navigate and maintain the structure outside the transaction, access the abstract state inside the transaction*

# Factoring the set data structure

1. Don’t store the transactional set $\mathbf{S}$ directly
2. Store the elements of a superset $\mathbf{U} \supseteq \mathbf{S}$
3. Store a predicate $f\colon \mathbf{U} \to \{0,1\}$ that tests membership, $f(e) = 1$ iff $e \in \mathbf{S}$

*The trick*

- Adding $e$ to $\mathbf{U}$ doesn’t change $\mathbf{S}$ if $f(e) = 0$
- $\mathbf{U}$ and $f$ can be grown in an escape action
- The STM only needs to manage 1 bit per $e$

# Storing U and *f*

1. Don’t store the transactional set $\mathbf{S}$ directly
2. Store the elements of a superset $\mathbf{U} \supseteq \mathbf{S}$
3. Store a predicate $f\colon \mathbf{U} \rightarrow \{0,1\}$ that tests membership, $f(e) = 1 \text{ iff } e \in \mathbf{S}$

*A thread-safe representation*

```
univ = ConcurrentMap[A,TVar[Boolean]]
U = univ.keySet()
f(e) = univ.get(e).stmRead()
```

# A minimal* implementation

```
class THashSet[A] {
  def contains(e: A) = bitForElem(e).stmRead()
  def add(e: A)      { bitForElem(e).stmWrite(true) }
  def remove(e: A)   { bitForElem(e).stmWrite(false) }

  private val univ = new ConcurrentHashMap[A,TVar[Boolean]]

  private def bitForElem(e: A): TVar[Boolean] = {
    var bit = univ.get(e)
    if (bit == null) {
      val fresh = new TVar(false)
      bit = univ.putIfAbsent(e, fresh)
      if (bit == null)
        bit = fresh
    }
    return bit
  }
}
```

* - We'll add GC of TVars later

# What does the factoring buy us?

- Lower STM overheads
  - Read- and write-set entries are minimized
    - *Set read is **one** txn read*
    - *Set insert or removal is **one** txn write*
  - Non-composed accesses don't need a transaction
    - *STMs can heavily optimize isolation barriers*
- Better scalability
  - No structural false conflicts
  - Transactional accesses to the set conflict if and **only if** they perform a conflicting operation on the same key
- Atomicity and isolation still managed by the STM
  - Optimistic concurrency and invisible readers
  - Modular blocking with `retry`/`orElse` works

# Predicating a map

```
TSet[A] →
  ConcurrentMap[A,TVar[Boolean]]
```

```
TMap[K,V] →
  ConcurrentMap[K,TVar[Option[V]]]
```

```
univ.get(k).stmRead() == Some(v)
```

if the current txn context observes $k \mapsto v$

```
univ.get(k).stmRead() == None
```

if the current txn context observes $k$ to be absent

# Trimming the universe

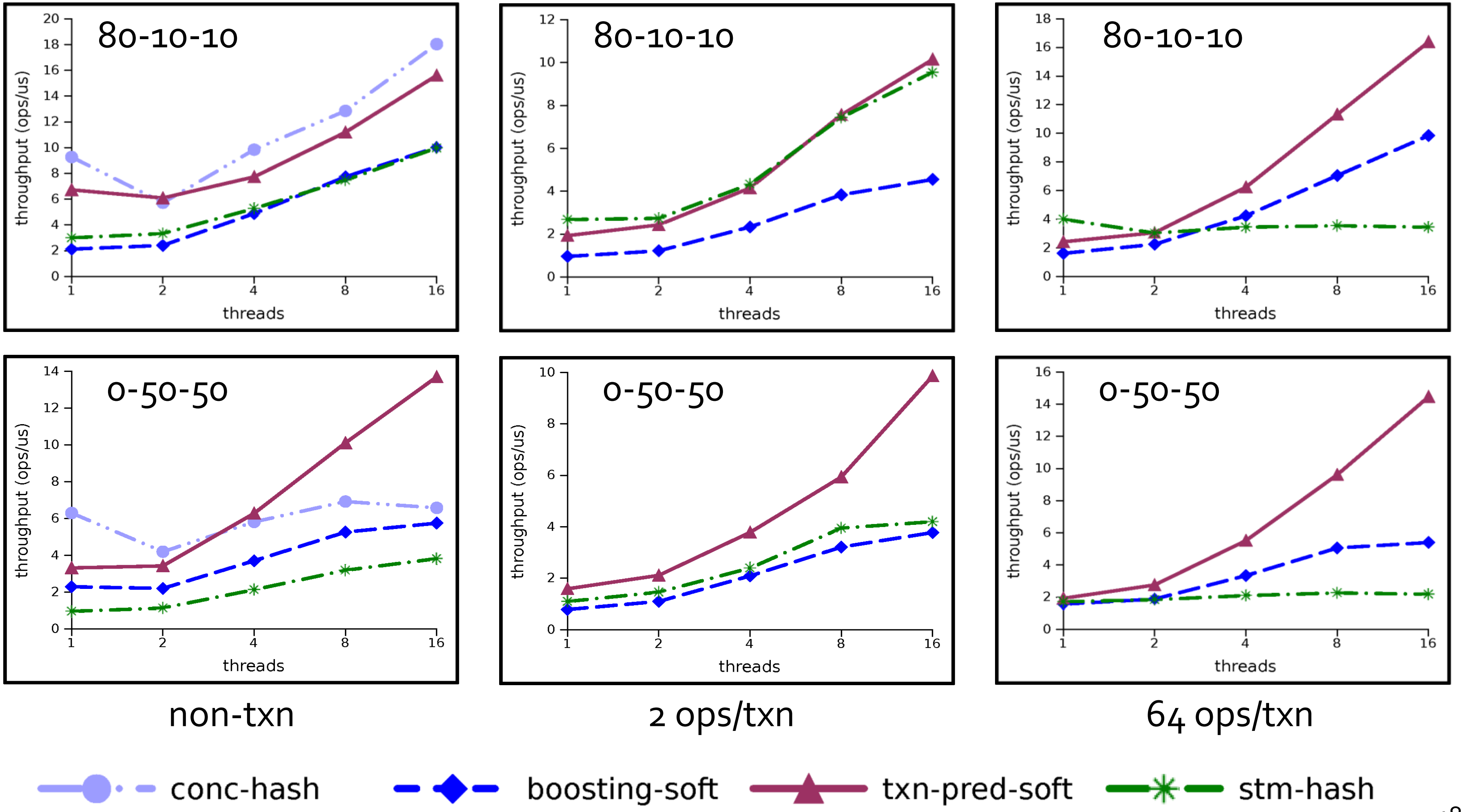
$e$ can be removed when $f(e) = 0$ and no txns are using $e$ (reading, writing, or blocked on `retry` for $e$'s `TVar`)

1. Reference counting
   - Enter before use, exit on txn completion
   - Add bonus when committing $f(e) = 1$
   - Speculatively read $f(e)$, skip entry/exit when bonus is present
2. Soft reference to a throw-away token
   - When $f(e) = 1$, *TVar* holds a strong reference to the token
   - When $f(e) = 0$, *TVar* has only a soft reference
   - Txn using $e$ keeps a strong reference
   - GC of token means all participants agree on absence

# Performance: low contention

key range of 200K

get% - put% - remove%



non-txn

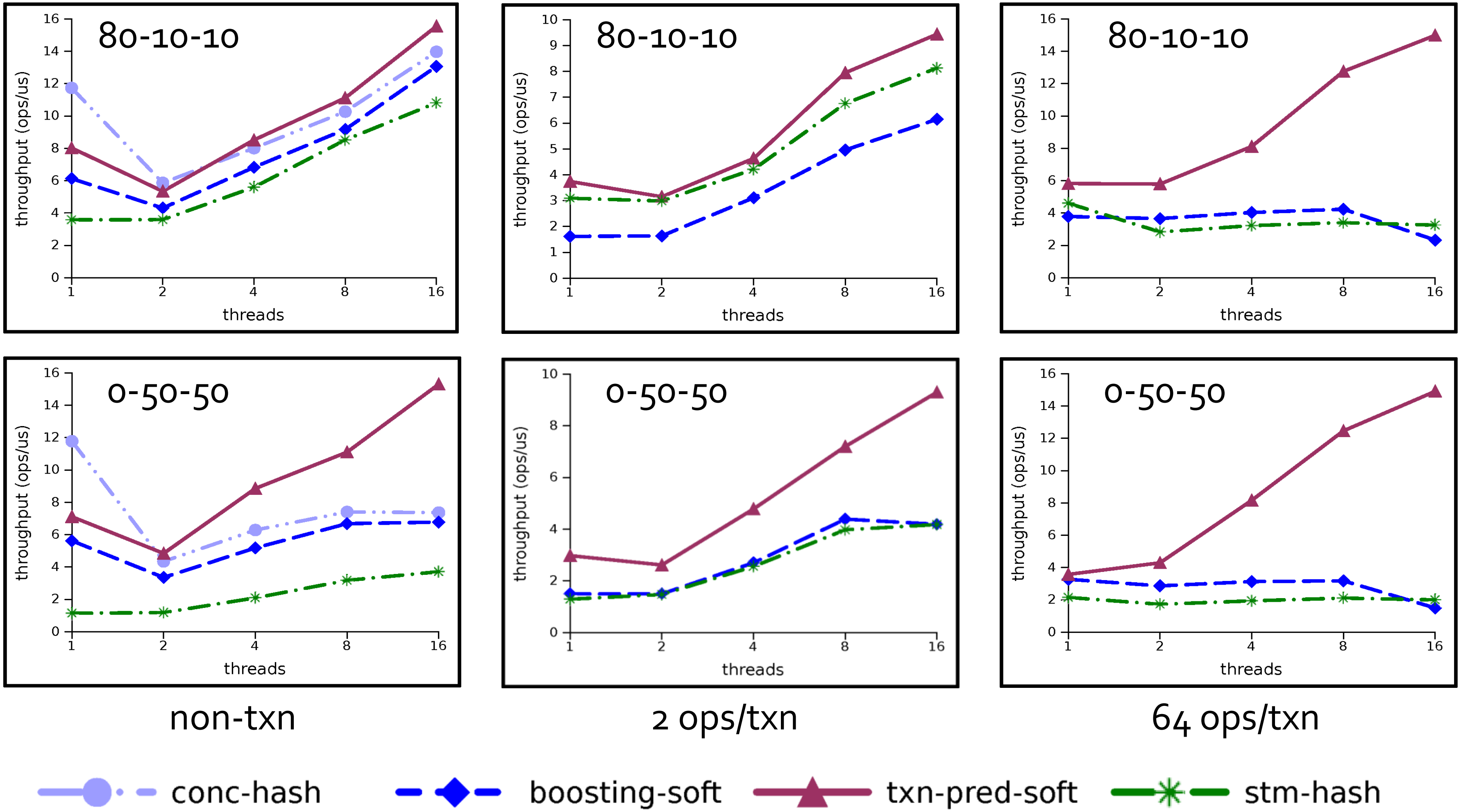
2 ops/txn

64 ops/txn

# Performance: high contention

key range of 2K

get% - put% - remove%



non-txn

2 ops/txn

64 ops/txn

# Conclusion

Transactionally-predicated sets and maps

- Fast when used outside an atomic block
- Full STM integration
- Lower overhead and better scalability than existing approaches
- Retains the features of the underlying STM
    - *Optimistic concurrency and invisible reads*
    - *Opacity*
    - *Modular blocking*

*Thank you*

# Previous methods for semantic conflict detection

- Open nesting
  - Carlstrom et al., and Ni et al., both PPoPP'07
  - Reduces false conflicts
  - Worsens STM overheads
- Transactional boosting
  - Herlihy et al., PPoPP'08
  - Reduces false conflicts and TM overheads
  - Adds non-transactional work to locate associated locks
  - Pessimistic visible readers limit concurrency and scalability
  - Boosting voids the forward progress, opacity, and modular blocking properties of the underlying STM

# Boosting (Herlihy et al.)

- Start with a thread-safe object
  - Implemented without STM
- Associate a lock with each set of non-commutative operations
  - set.op(k1) and set.op(k2) only affect each other if k1 = k2
  - So, associate one lock per key

-
  ```
  Set[A] => { s: ConcurrentSet[A];
    locks: ConcurrentMap[A,Lock] }
  ```
- Transactional access
  - Acquire locks(key), then call s.op(key)
    - *Even if key is not in the set*
  - Hold lock until the end of the transaction
  - Record result of op, apply compensating action on rollback

# Problems with Txn Boosting

- Scalability + performance
  - Pessimistic concurrency means readers cannot overlap writers
  - Adds an extra concurrent map lookup to each operation
- Correctness
  - Deadlock must be detected and avoided separately
- Functionality
  - Not compatible with conditional retry (retry + orElse)

*Basically, this is a pessimistic visible-reader STM implemented using callbacks. It ignores most of the research into how to build an efficient and scalable STM!*

# THashSet: An Example

```
begin T1
  S.contains(10)
  |  bitForElem(10)
  |  |  univ.get(10) -> null
  |  |  f = new TVar(false)
  |  |  univ.putIfAbsent(10,f)
  |  |      -> null
  |  -> f
  |  f.stmRead() -> false
  -> false

  // other work in txn

  CONFLICT on f
```

```
begin T2
  S.add(10)
    bitForElem(10)
    |  f = univ.get(10)
    -> f
    f.stmWrite(true)
commit
```

# Transactional Predication: Enumeration + Search

- Basic strategy
  - Enumerate or search in the underlying map
  - Skip entries that are conceptually absent
  - Add transactional state that is modified by any structural insertion that conflicts with the search
- Examples
  - Unordered collection: maintain a striped size
    - *Insertions and removals update their stripe*
    - *Iteration counts entries, checks against the sum of the stripes*
  - Ordered collection: maintain per-node predecessor and successor insertion counts
    - *Insertion counts are incremented non-transactionally when updating the structure, with recursive helping to avoid races*
    - *Search and enumeration read the insertion counts*