DREADLOCKS: Efficient Deadlock Detection

BROWN

Maurice Herlihy

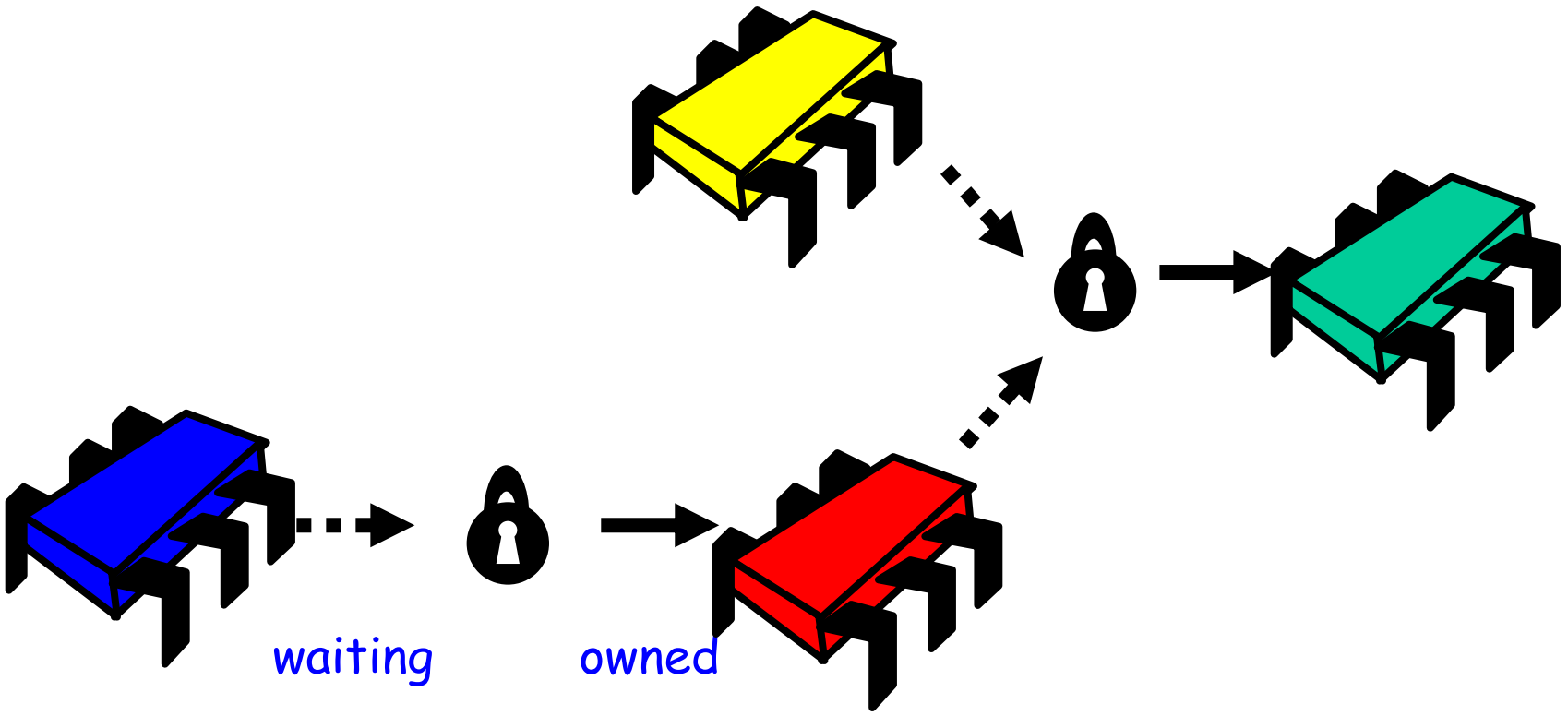
Joint work with Eric Koskinen

Spin-Locks

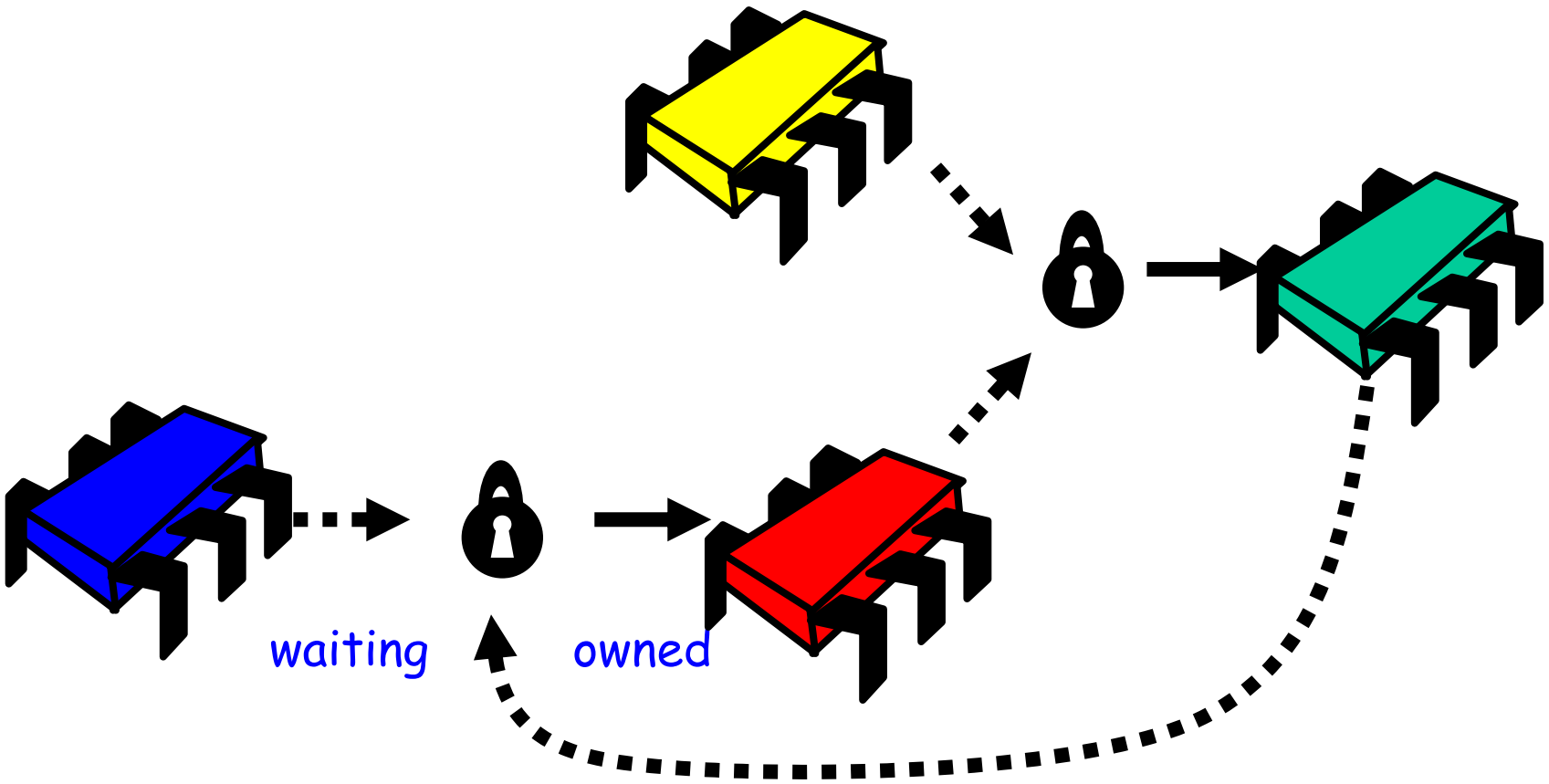
- Good for short pauses
- Multiprocessors
- Can be made cache-friendly

Deadlock

Waits-For Graph



Deadlock



Dealing with Deadlock

- Avoidance
  - Not practical if demands not known in advance
- Detection
  - Existing algs require complex probing and poking

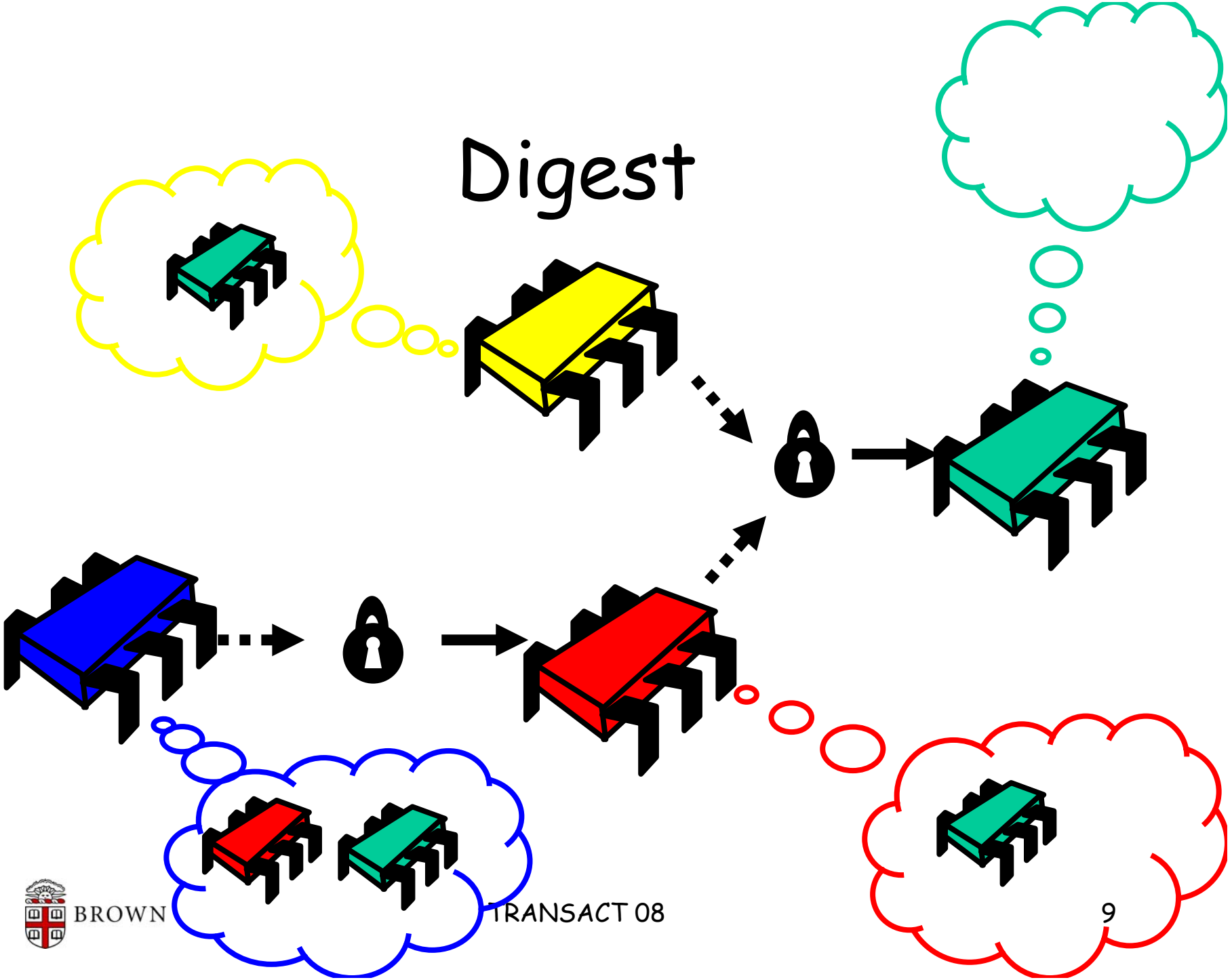
Can Use Timeouts

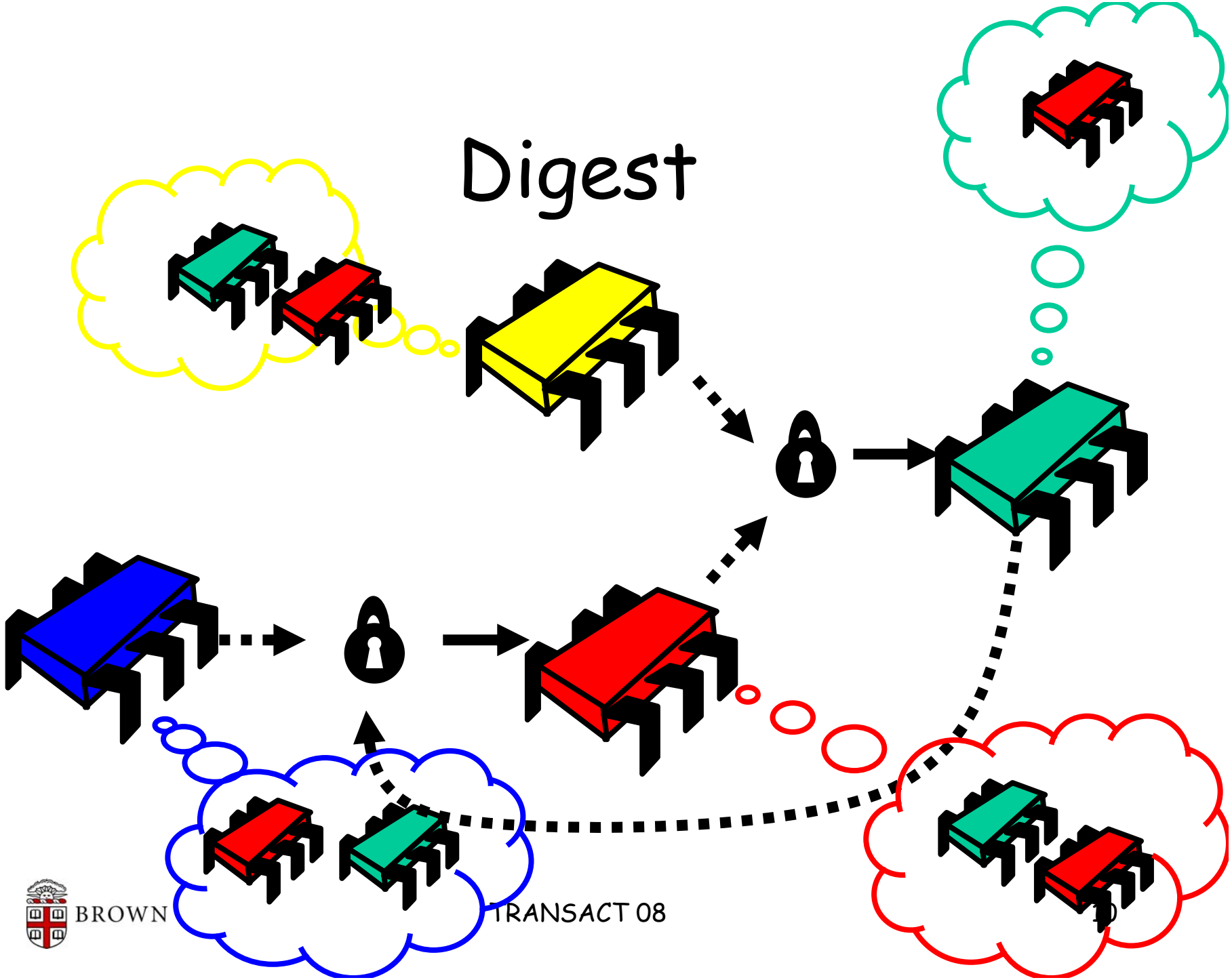
- Abort if you don't get lock in time
- Plus
  - Easy to implement
- Minus
  - How do you choose timeout?
  - Is choice robust?
    - Different platforms/apps?

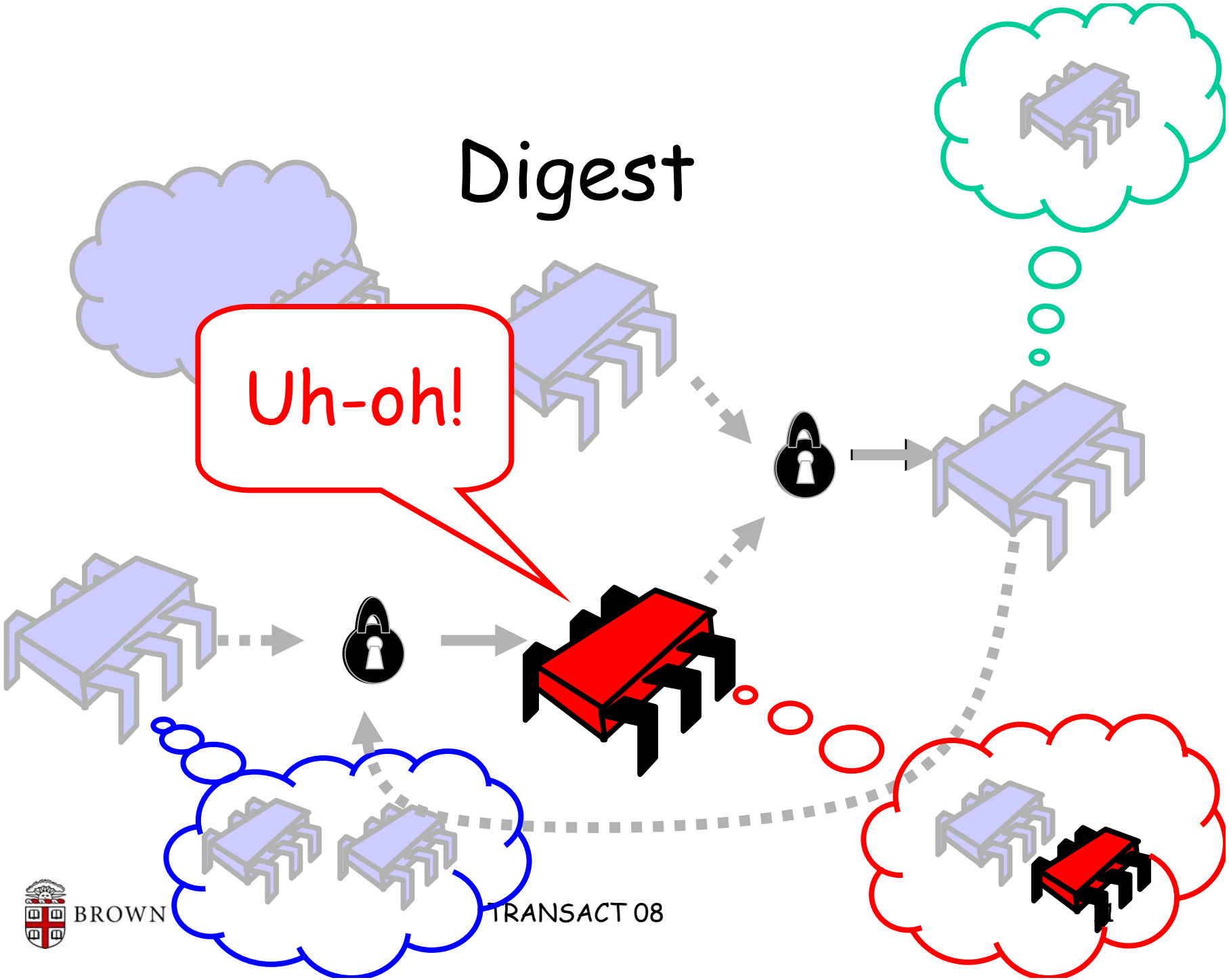
Dreadlocks

- Fast, incremental deadlock detection
- Low overhead
- Cache-friendly

- Algorithm …


Digest


Digest


Digest
Uh-oh!

Algorithm

- Each thread publishes digest
  - Initially contains only itself
  - Always contains itself
- Spins on lock owner's digest
  - Propagates changes (union with own)
  - Aborts if finds itself in owner's digest

T&T&Set Spin Lock

```
public void lock() {
  while (true) {
   while ((owner = state.get()) != null) {
     if (owner.contains(me)) {
       throw new AbortedException();
     } else if (owner.changed()) {
       myDigest.setUnion(owner, me);
     }
   }
   if (state.compareAndSet(null, myDigest)) {
     myDigest.setSingle(me);
     return;
   }
 }
}
```

T&T&Set Spin Lock

```
public void lock() {
  while (true) {
    while ((owner = state.get()) != null) {
      if (owner.contains(me)) {
        throw new AbortedException();
      } else if (owner.changed()) {
        myDigest.setUnion(owner, me);
      }
    }
    if (state.compareAndSet(null, myDigest)) {
      myDigest.setSingle(me);
      return;
    }
  }
}
```

Busy lock points to owner digest

T&T&Set Spin Lock

```
public void lock() {
  while (true) {
   while ((owner = state.get()) != null) {
     if (owner.contains(me)) {
       throw new AbortedException();
     } else if (owner.changed()) {
       myDigest.setUnion(owner, me);
     }
   }
   if (state.compareAndSet(null, myDigest)) {
     myDigest.setSingle(me);
     return;
   }
 }
}
```

Abort if I'm in owner's digest

T&T&Set Spin Lock

```
public void lock() {
  while (true) {
   while ((owner = state.get()) != null) {
     if (owner.contains(me)) {
       throw new AbortedException();
     } else if (owner.changed()) {
       myDigest.setUnion(owner, me);
     }
   }
   if (state.compareAndSet(null, myDigest)) {
     myDigest.setSingle(me);
     return;
   }
  }
}
```

Back-propagate changes from owner's digest

T&T&Set Spin Lock

```
public void lock() {
  while (true) {
    while ((owner = state.get()) != null) {
      if (owner.contains(me)) {
        throw new AbortedException();
      } else if (owner.changed()) {
        myDigest.setUnion(owner, me);
      }
    }
    if (state.compareAndSet(null, myDigest)) {
      myDigest.setSingle(me);
      return;
    }
  }
}
```

Clear own digest when lock acquired

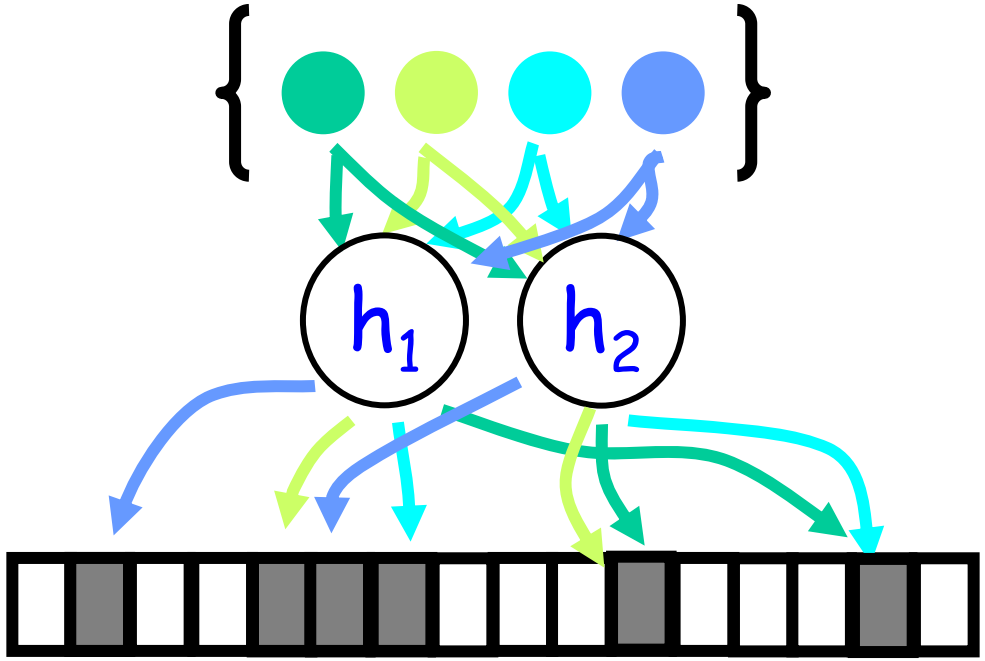
Concurrency

- Digest methods don't have to be atomic
  - False positives OK if rare
  - False negatives OK if transient
- Means we can use multi-word bitmaps

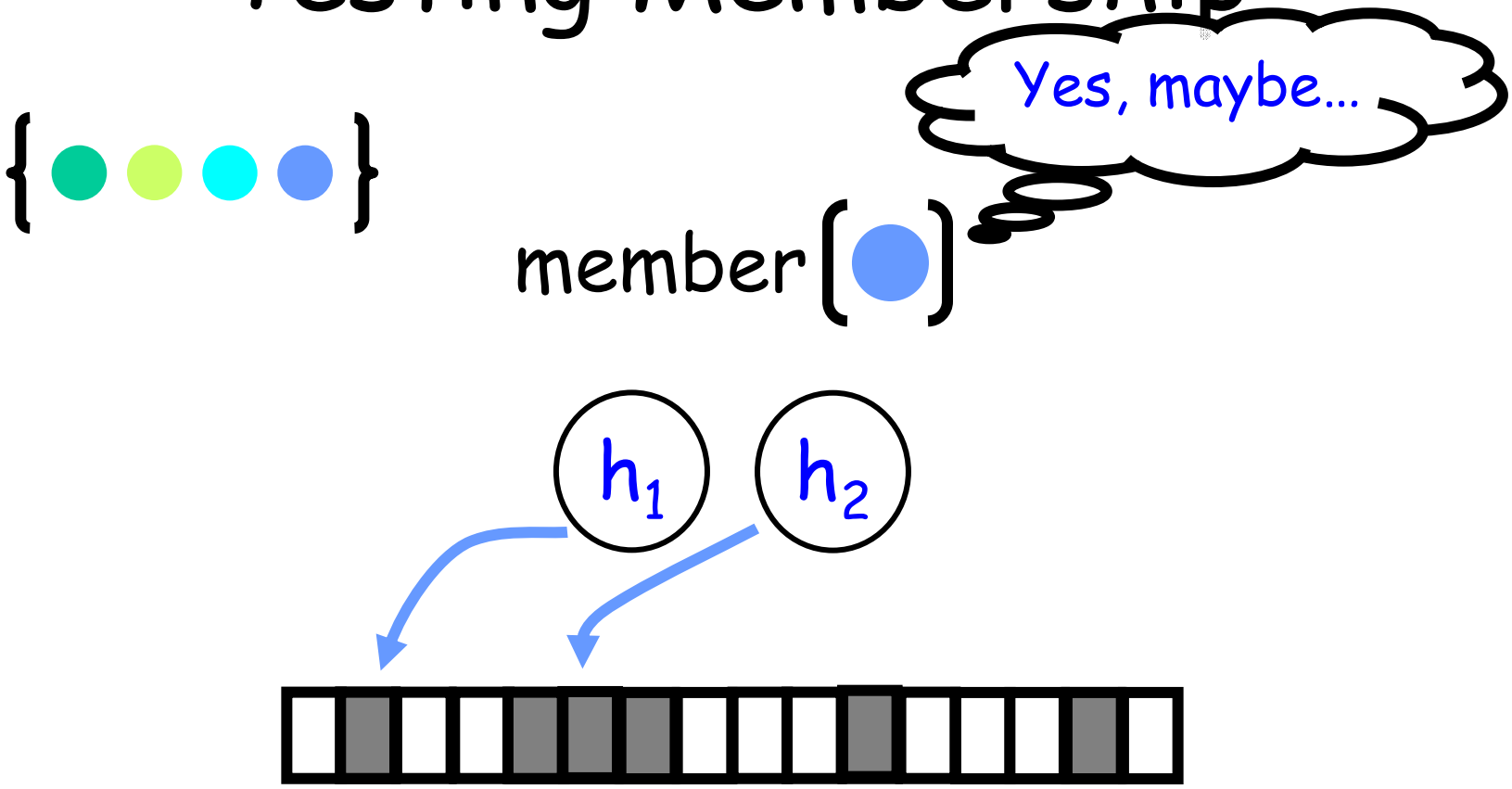
Bit Maps

- 32 (or 64) bit array
- Good for small sets
- Exact membership tests
- Manipulated by shifting & masking

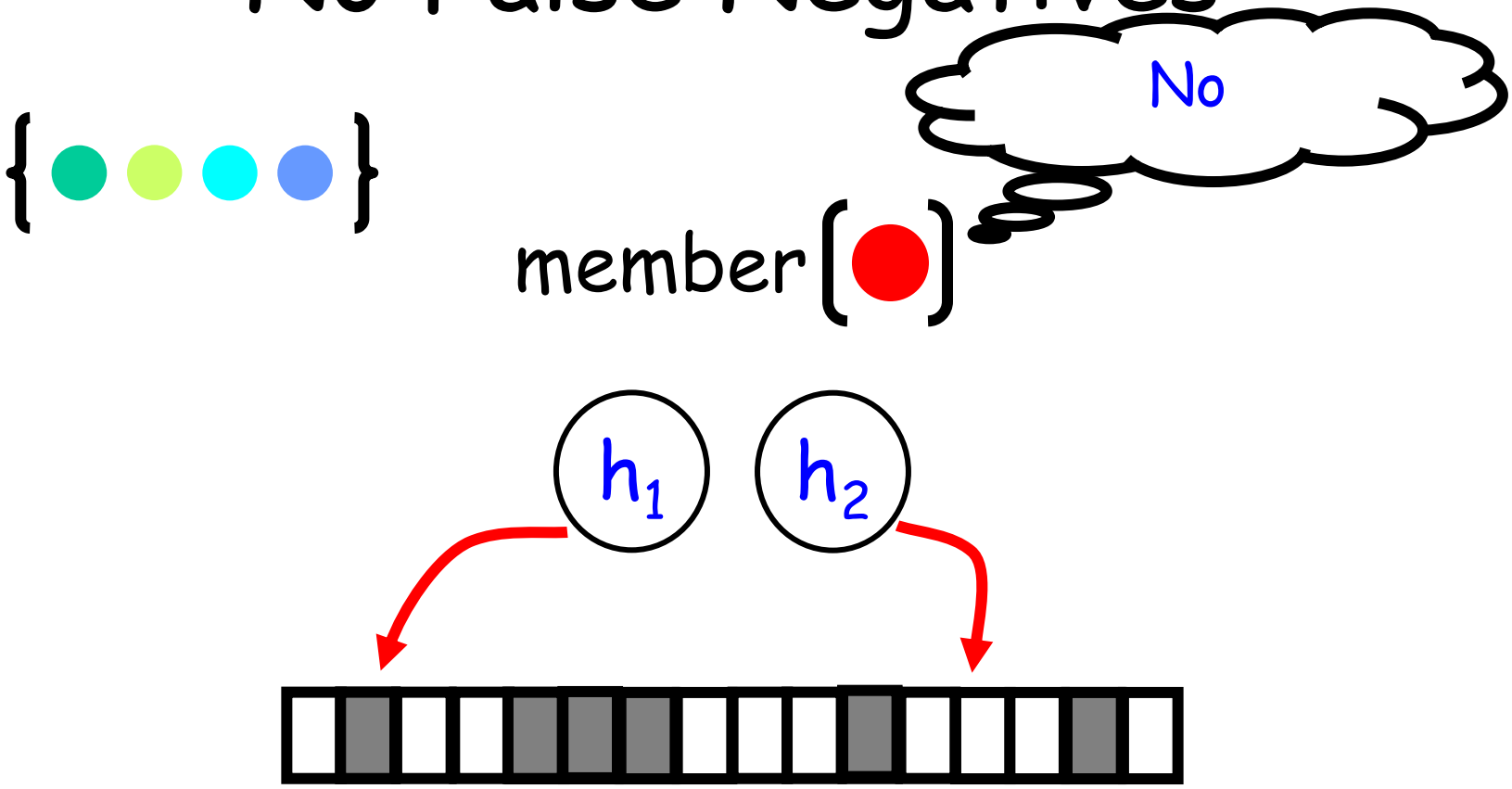
Bloom Filter



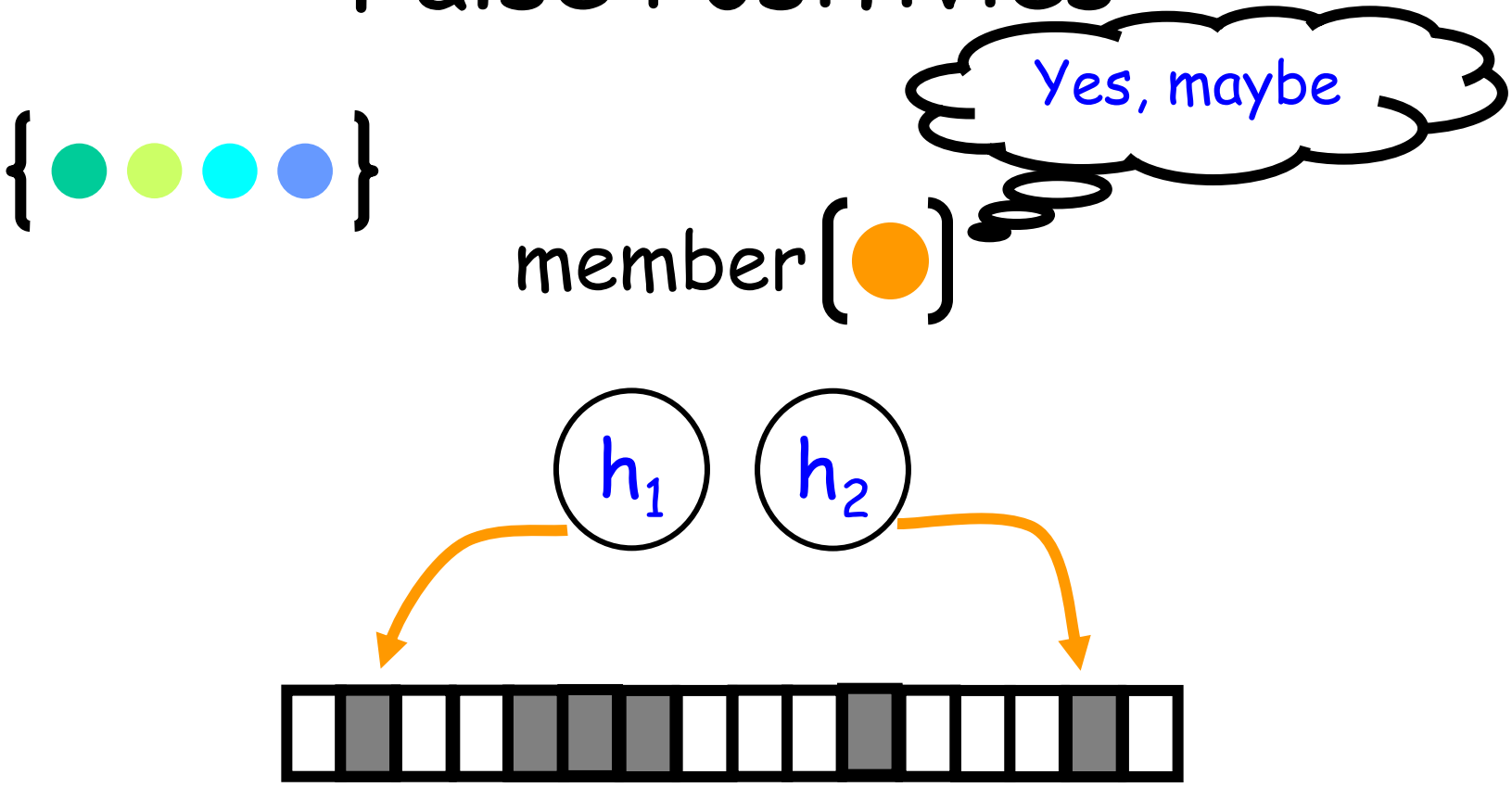
Testing Membership



No False Negatives



False Positivies

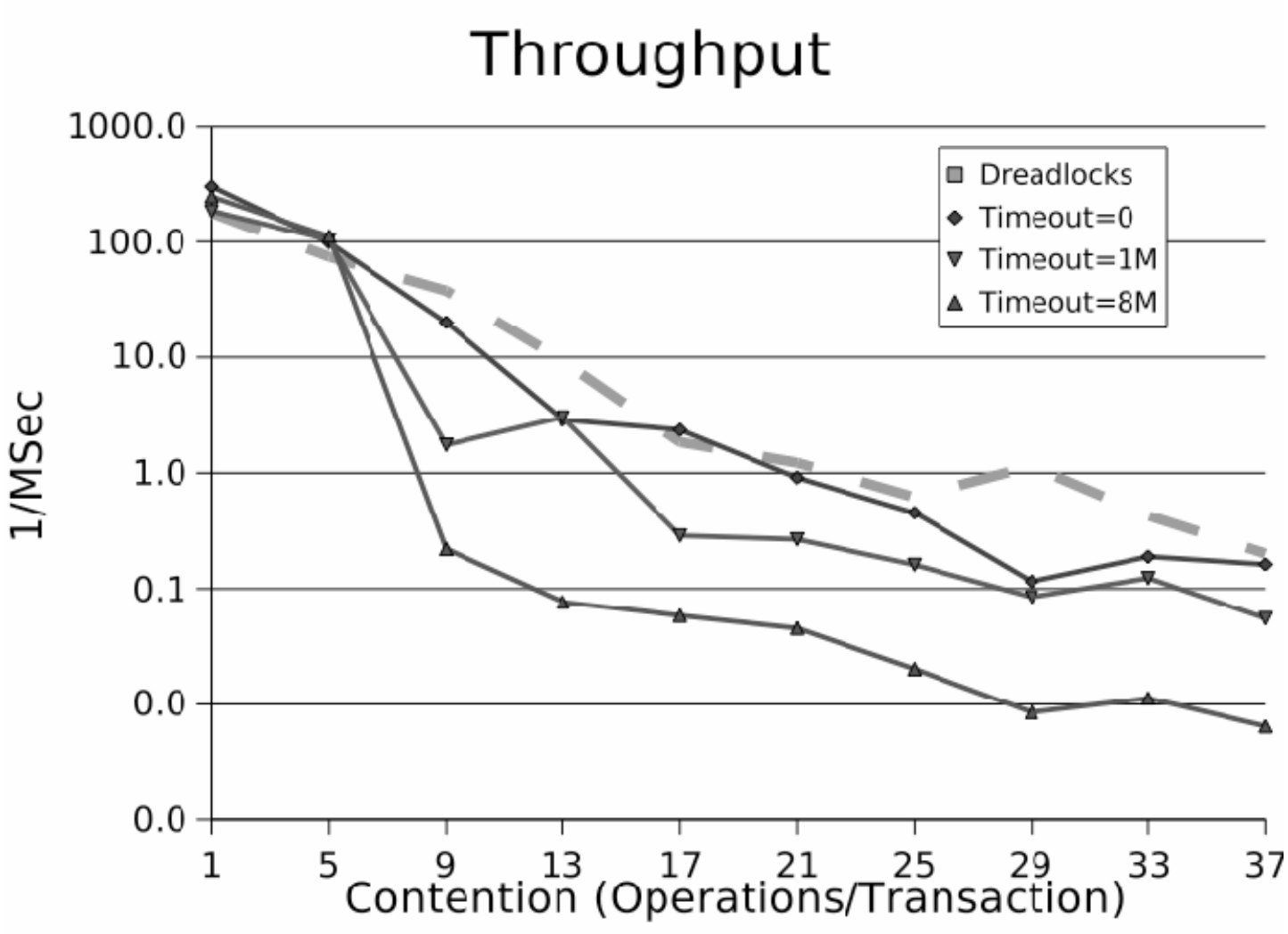


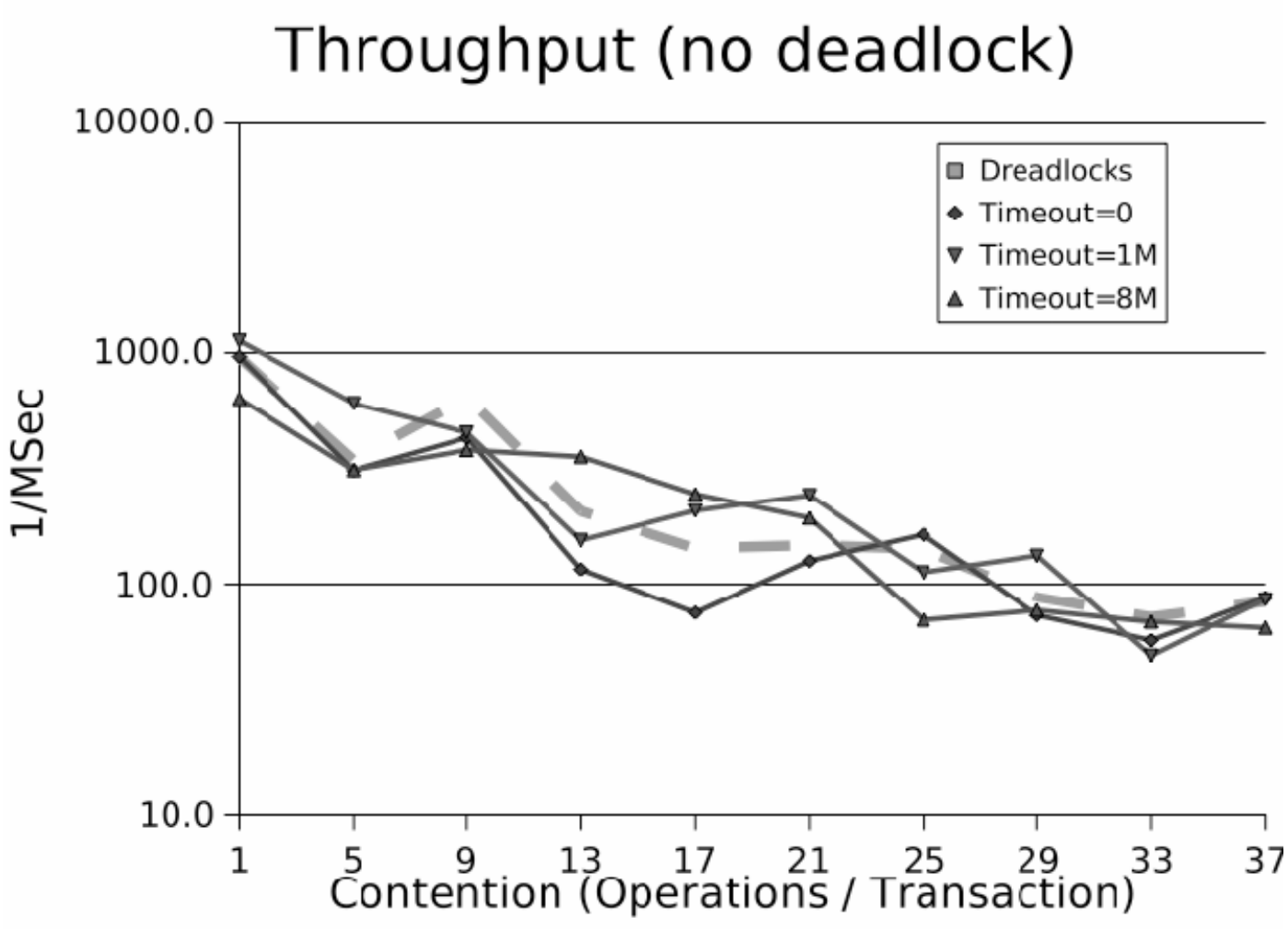
Experiments

- Dreadlock TTAS lock
- Used for Abstract Locking in Boosting

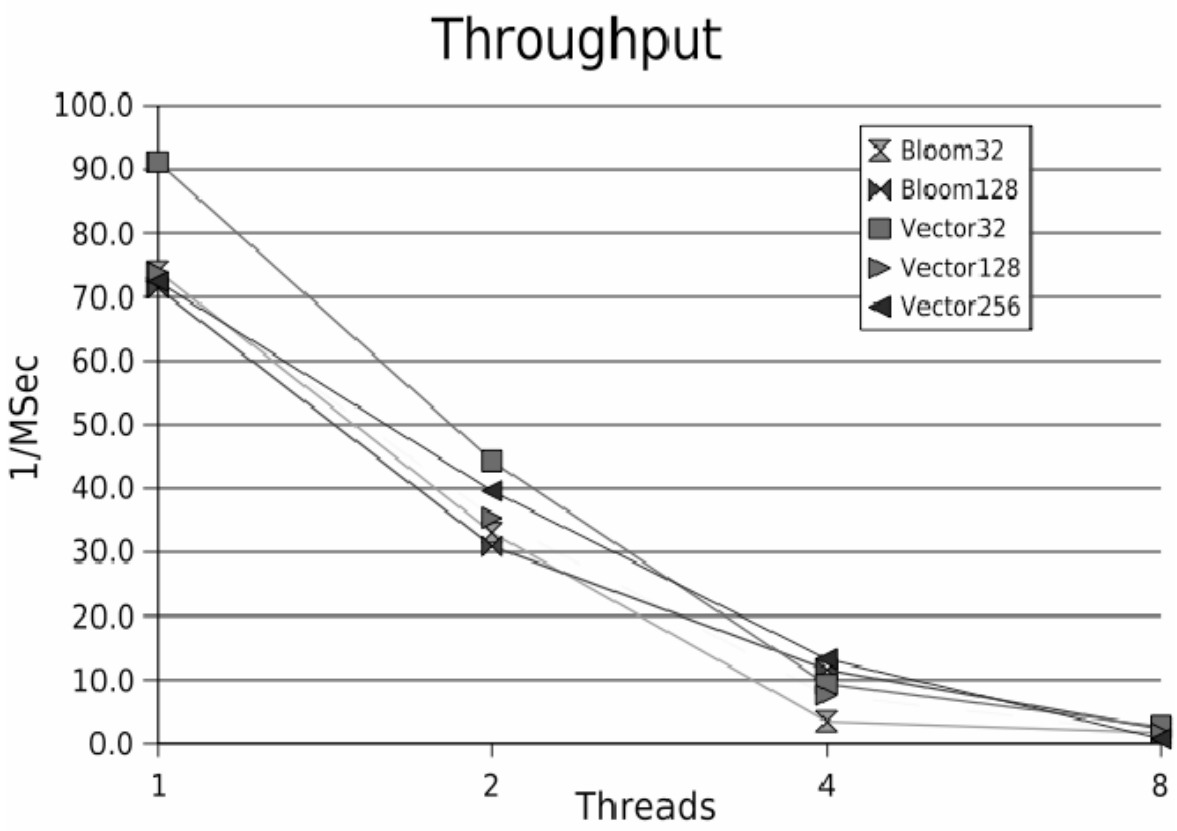
- Implemented TTAS lock in C
- Built upon Boosting / TL2

- Synthetic benchmark: boosted array

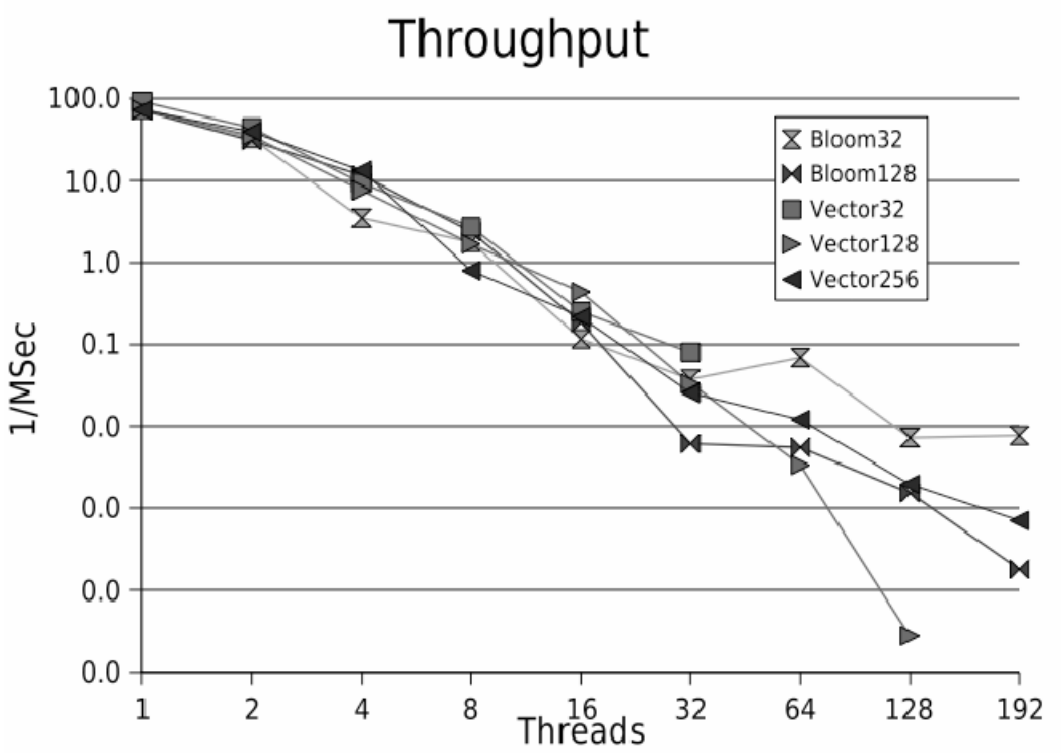

Throughput
1000.0
100.0
10.0
1.0
0.1
0.0
0.0
1/MSec
Dreadlocks
Timeout=0
Timeout=1M
Timeout=8M
1
5
9
13
17
21
25
29
33
37
Contention (Operations/Transaction)


Throughput (no deadlock)
10000.0
1000.0
100.0
10.0
1/MSec
Dreadlocks
Timeout=0
Timeout=1M
Timeout=8M
1
5
9
13
17
21
25
29
33
37
Contention (Operations / Transaction)

Throughput Small # Threads



Throughput Large # Threads



No Deadlock No Cry

- Can use locks with reckless abandon
- Dreadlocks will detect deadlocks

- So far
  - Slightly higher overhead