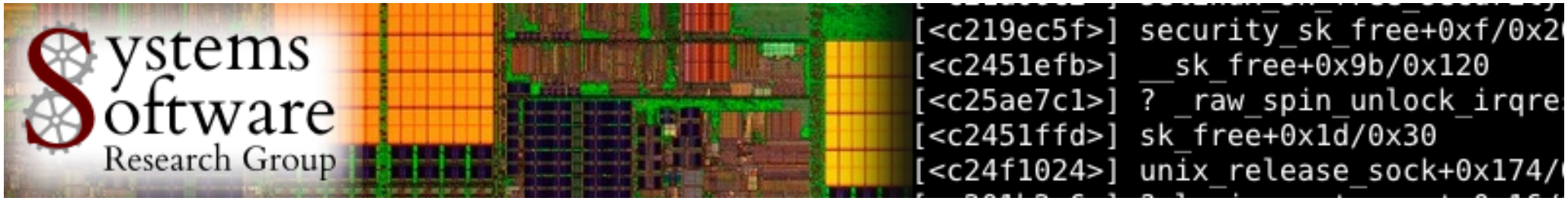


On Developing Optimistic Transactional Lazy Set

Ahmed Hassan, Roberto Palmieri, Binoy Ravindran

Systems Software Research Group

Virginia Tech

OPODIS 2014

Virginia Tech

Motivation

- Concurrent data structures are well optimized for high performance
  - E.g., Lazy linked-list, Lazy skip-list

What about Transactional data structures?

Why Transactional data structures?

```
Shared data: concurrentList


atomicFoo()
{
        concurrentList.add(x);
}
```

Why Transactional data structures?

```
Shared data: concurrentList

atomicFoo()
{
        concurrentList.add(x);
        concurrentList.add(y);
}
```

- Composability

Why Transactional data structures?

```
Shared data: concurrentList1
Shared data: concurrentList2

atomicFoo()
{
        concurrentList1.remove(x);
        concurrentList2.add(x);
}
```

- Composability

Why Transactional data structures?

```
Shared data: concurrentList

atomicFoo()
{
        If(concurrentList.add(x))
                n1++;
        Else
                n2++;
}
```

- Composability
- Integration

Solutions?

- Software Transactional Memory (STM)?
  - Yes, but will lose performance

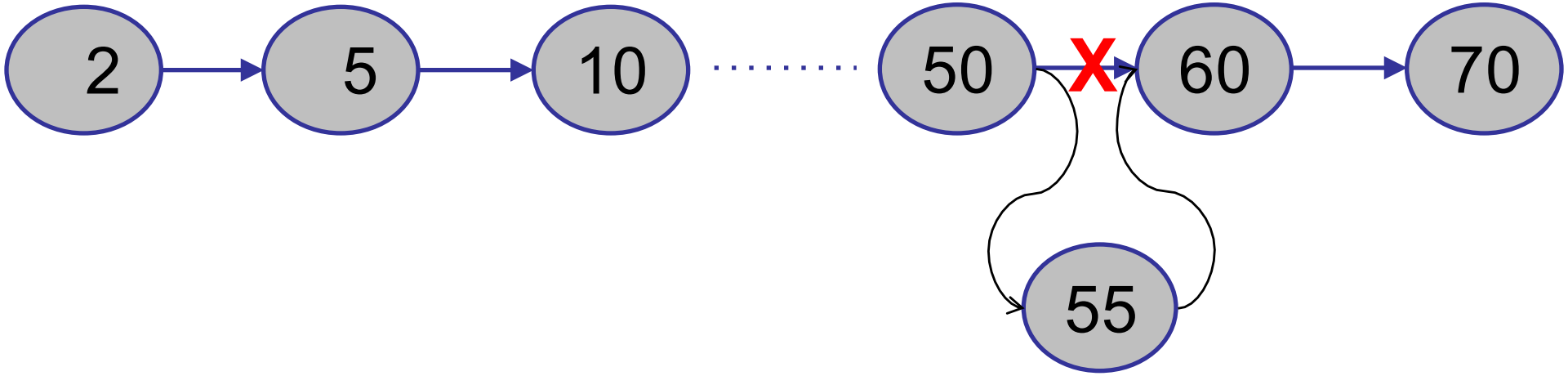
```
Shared data: sequentialList

@Atomic
atomicFoo()
{
        sequentialList.add(x);
        sequentialList.add(y);
}
```

- Why?
  - For STM to be a general framework, data structures will suffer from false conflicts

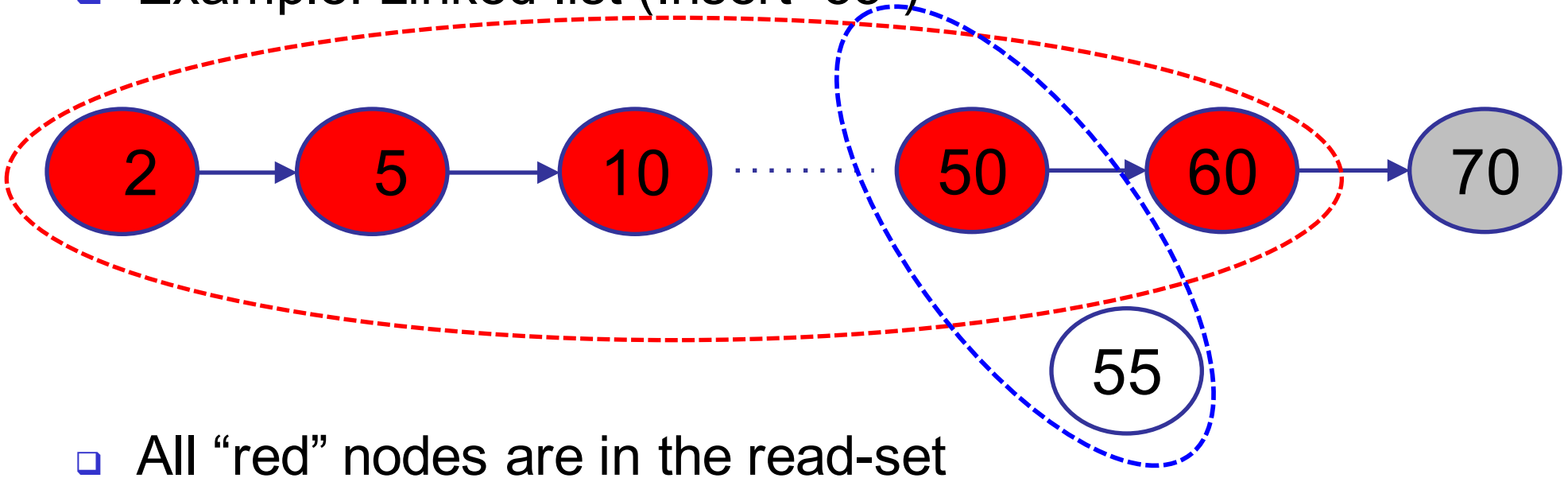
False Conflict

- Example: Linked list (Insert “55”)



False Conflict

- Example: Linked list (Insert “55”)



- All “red” nodes are in the read-set
- “50” and “55” are in the write-set
- What if a concurrent transaction deletes “5”??

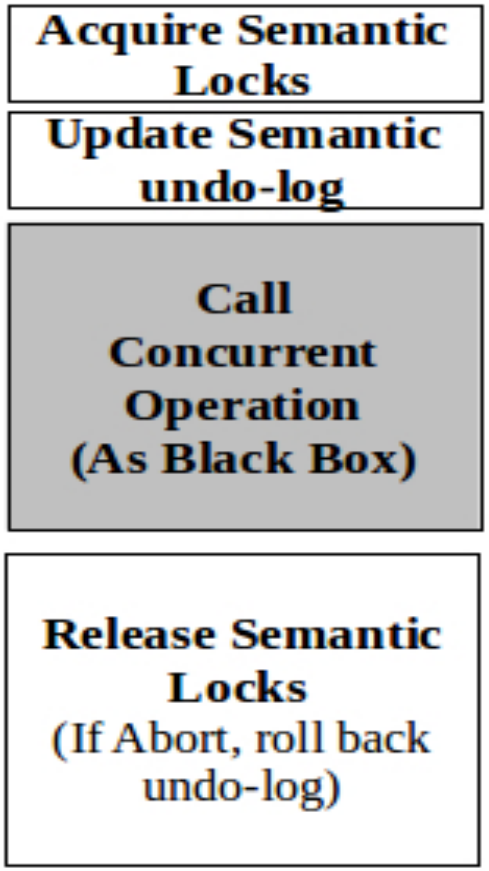
False Conflict

Earlier Solution: Transactional Boosting

- Convert highly concurrent data structures to be transactional.

- Composable (like STM)
- And efficient (like lazy/lock-free)

- Issues:
  - Eager locking.
  - Inverse operations.
  - Black-box concurrent data structure.



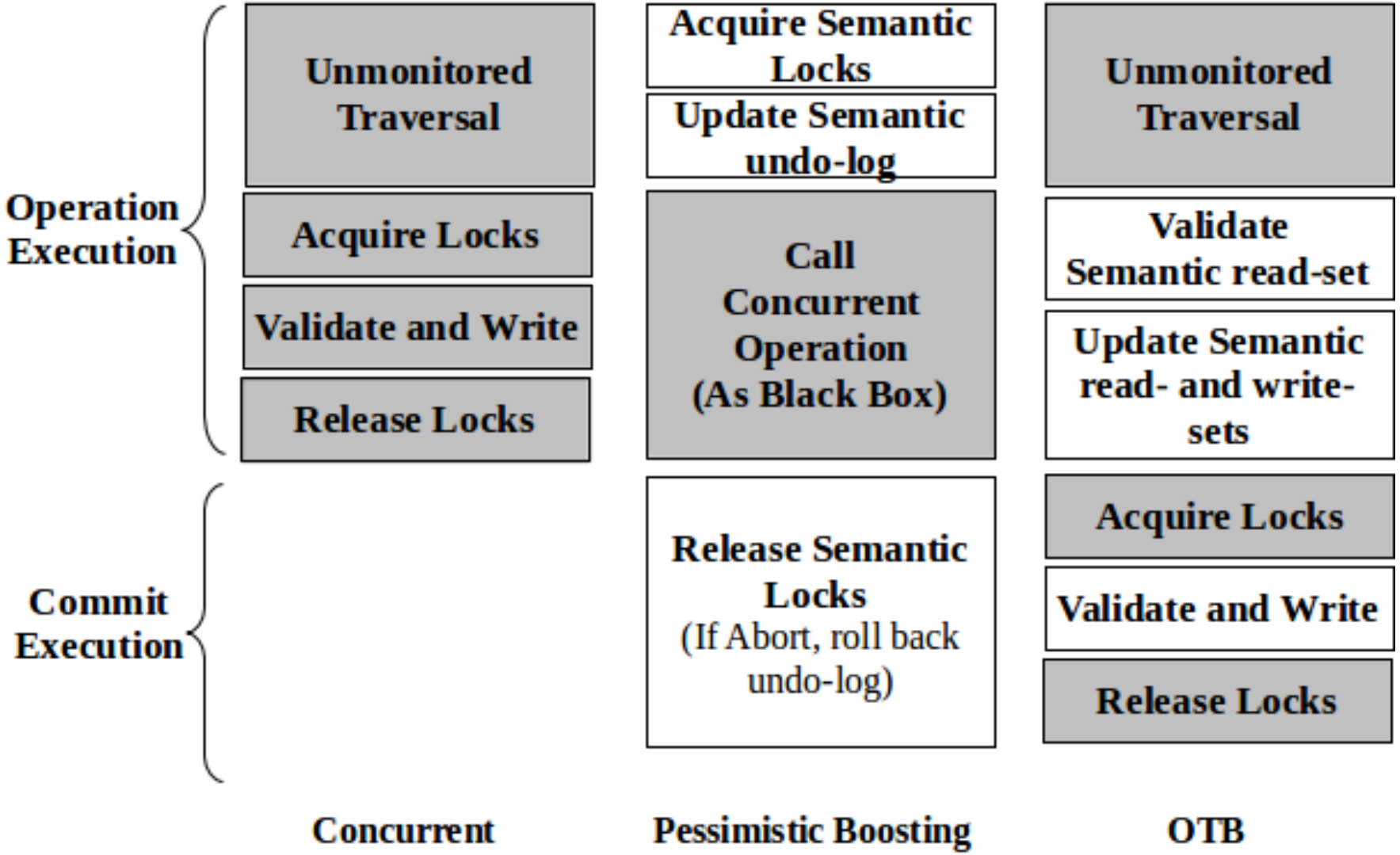
Our Solution: Optimistic Transactional Boosting (OTB)

- Convert highly concurrent data structures to be transactional.

AND

- Lazy updates.
- White-box data structures.
- No need for inverse operations.
- Easy integration with STM frameworks.

Lazy Vs Boosting Vs Optimistic Boosting



OTB Guidelines

G1: Split the (semantic) data structure operations.

G2: Validate/Commit to guarantee Opacity.

} Non optimized

G3: Optimize the data structure.

} Optimized

OTB Guidelines

- Split the data structure operations.
- Validate/Commit to guarantee Opacity.
- Optimize the data structure.

G1: Split Operation

- Example: Linked list (Insert “55”)

G1: Split Operation

- Example: Linked list (Insert “55”)

- Traversal (unmonitored)

G1: Split Operation

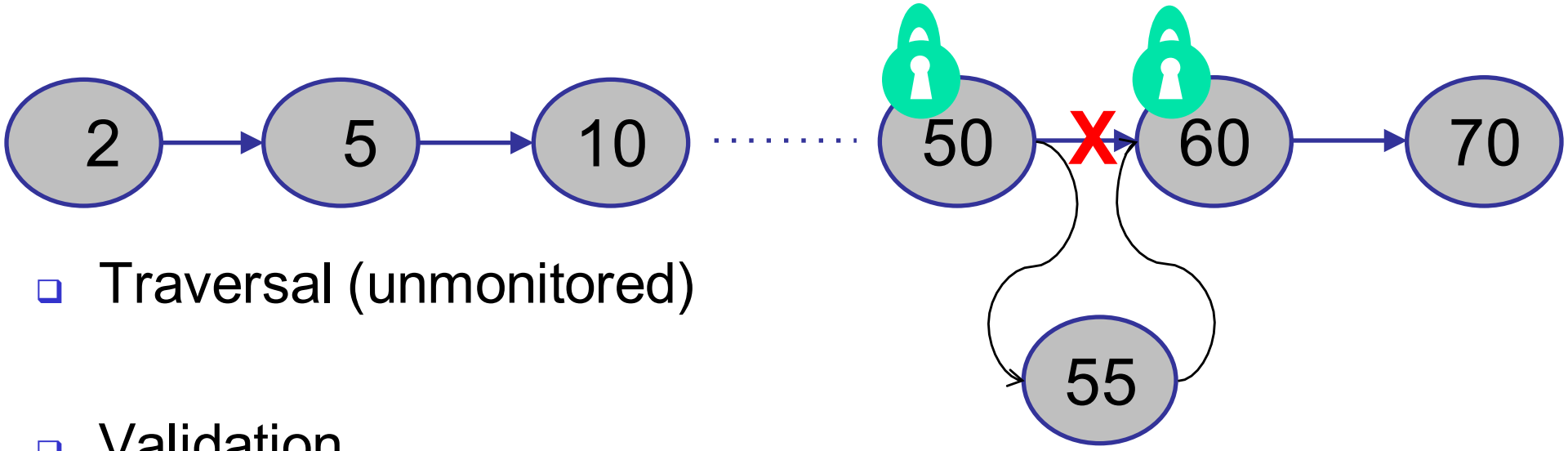
- Example: Linked list (Insert “55”)

- Traversal (unmonitored)

- Validation

G1: Split Operation

- Example: Linked list (Insert “55”)



- Traversal (unmonitored)
- Validation
- Commit

G1: Split Operation

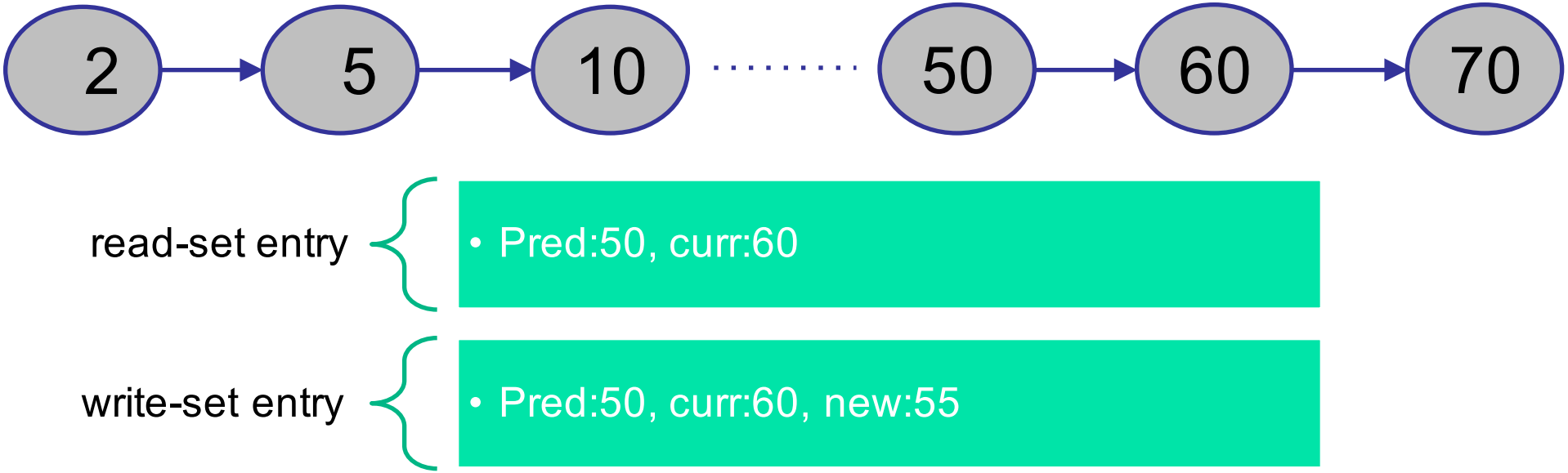
- Results of traversal are saved in local objects:
  - Semantic read-set: to be validated.
  - Semantic write-set: to be published at commit.

G1: Split Operation

- Example: Linked list (Insert “55”)

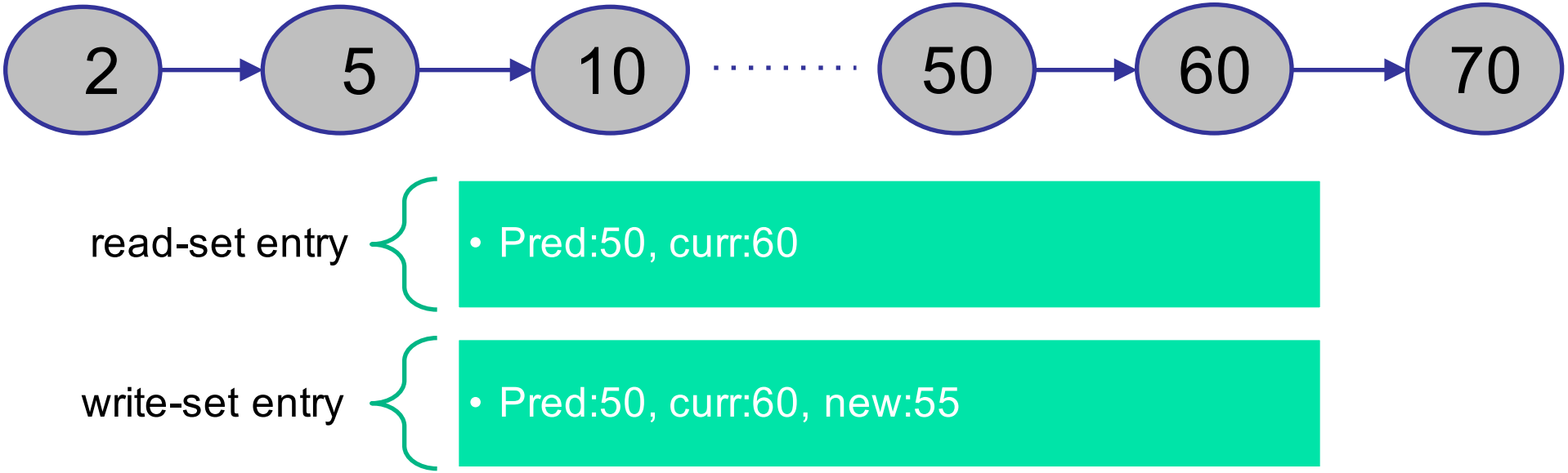
G1: Split Operation

- Example: Linked list (Insert “55”)



G1: Split Operation

- Example: Linked list (Insert “55”)



- Validation:
  - Pred.deleted == false
  - Curr.deleted == false
  - Pred.next == Curr

G1: Split Operation

- Performance:
  - Traversal without instrumentation: No false conflicts.
- Functionality:
  - Validation guarantees that unmonitored traversal does not harm.
  - Defer Commit to the end of the transaction: Composability & TM Integration.

OTB Guidelines

- Split the data structure operations.
- Validate/Commit to guarantee Opacity.
- Optimize the Data structure.

G2: Validation & Commit

- How OTB guarantee opacity:

1. Each operation, scan the local write-set first.

2. Re-validation of semantic read-set after each operation and during commit.

3. Two Phase Locking during commit.

G2: Validation & Commit

- How OTB guarantee opacity:
  4. During commit, publish operations according to the order they are invoked in the transaction, and propagate their effect.

G2: Validation & Commit

- How OTB guarantee opacity:
  4. During commit, publish operations according to the order they are invoked in the transaction, and propagate their effect.
- Example: Linked list (insert “55” and “56”)

write-set entry1

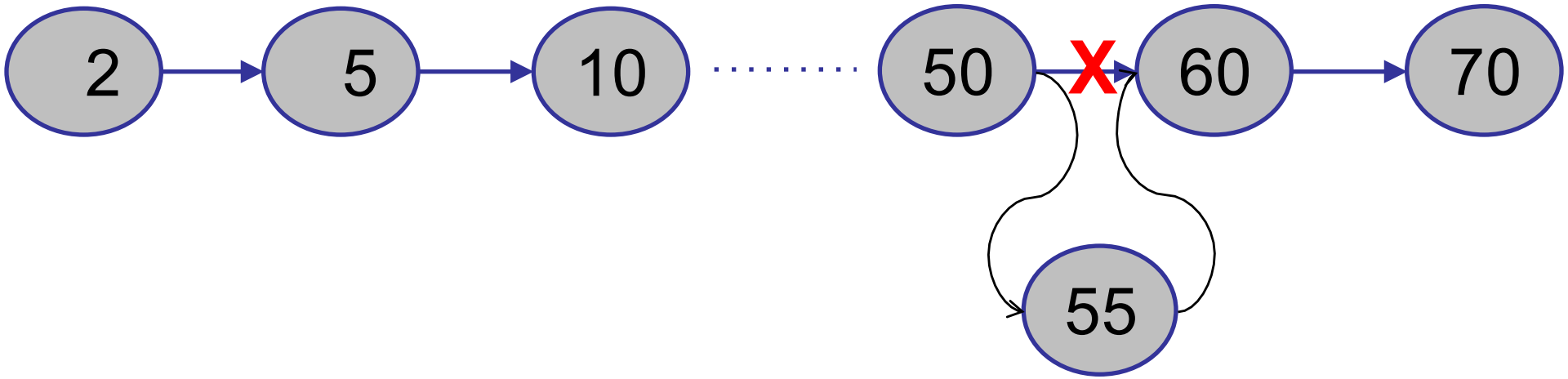
- Pred:50, curr:60, new:55

write-set entry2

- Pred:50, curr:60, new:56

G2: Validation & Commit

- How OTB guarantee opacity:
  4. During commit, publish operations according to the order they are invoked in the transaction, and propagate their effect.
- Example: Linked list (insert “55” and “56”)



write-set entry1

- Pred:50, curr:60, new:55 (Published)

write-set entry2

- Pred:55, curr:60, new:56

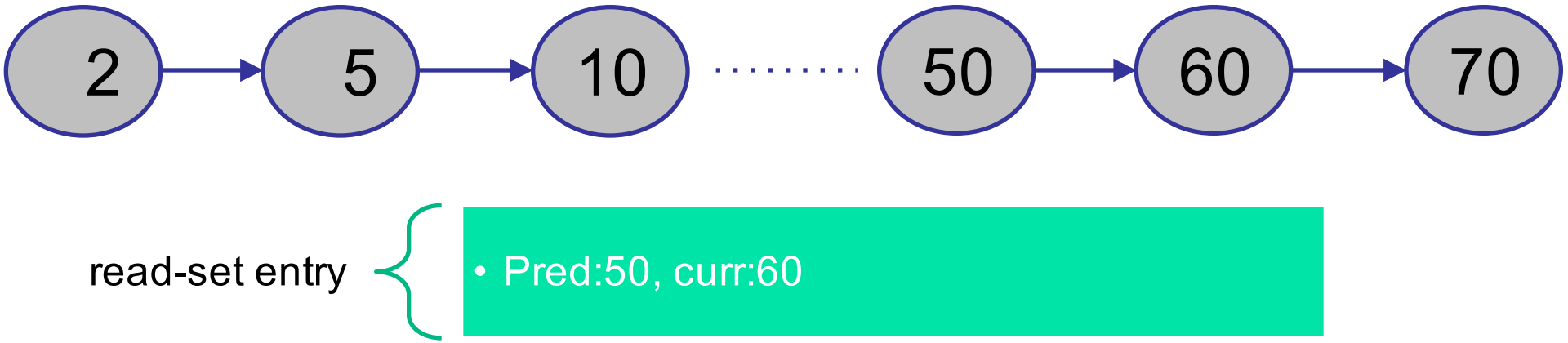
G2: Validation & Commit

- How OTB guarantee opacity:
  5. All operations has to be validated even if they are not validated in the concurrent version (e.g., contains).

G2: Validation & Commit

- How OTB guarantee opacity:
  5. All operations has to be validated even if they are not validated in the concurrent version (e.g., contains).

- Example: Linked list (search for “60”)



G2: Validation & Commit

- How OTB guarantee opacity:
  5. All operations has to be validated even if they are not validated in the concurrent version (e.g., contains).

- Example: Linked list (search for “60”)

read-set entry

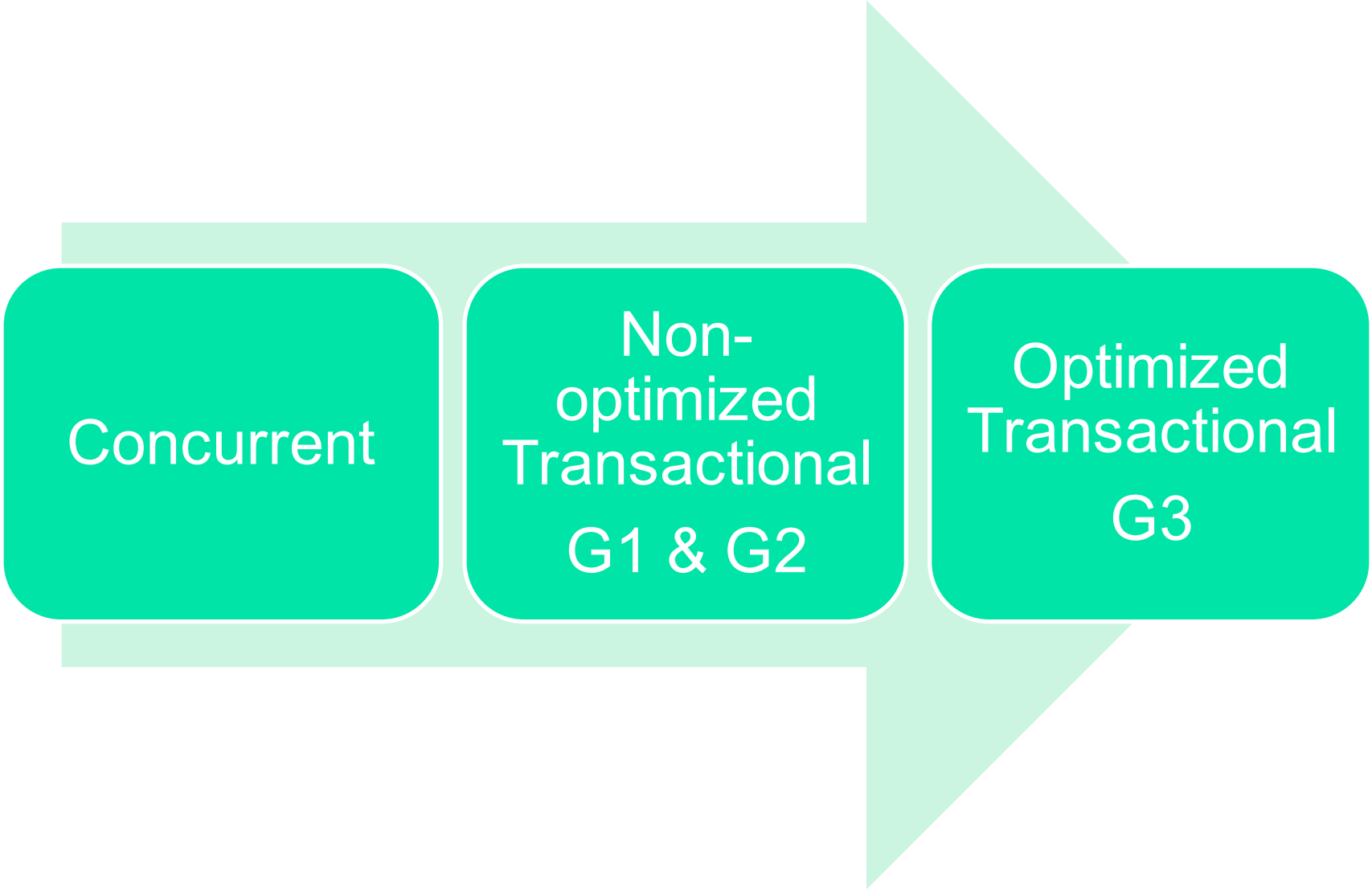
- Pred:50, curr:60

  - During commit: this entry has to be validated to ensure that 60 is still in the list and not deleted.
  - In the concurrent version, this validation is not needed.

OTB Guidelines

- Split the data structure Operation.
- Validate/Commit to guarantee Opacity.
- Optimize the Data structure.

G3: Specific Optimizations



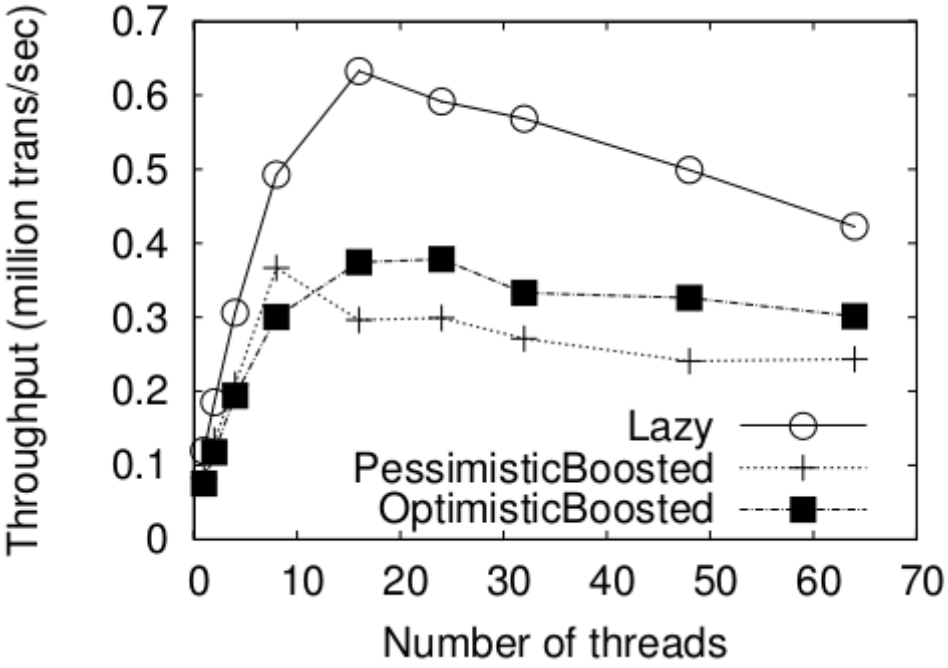
G3: Specific Optimizations

- Example optimizations on Linked-List and Skip-List
  - Elimination:
    - Ex. Add(x) then Remove(x).
    - No need to access the shared data structure.

G3: Specific Optimizations

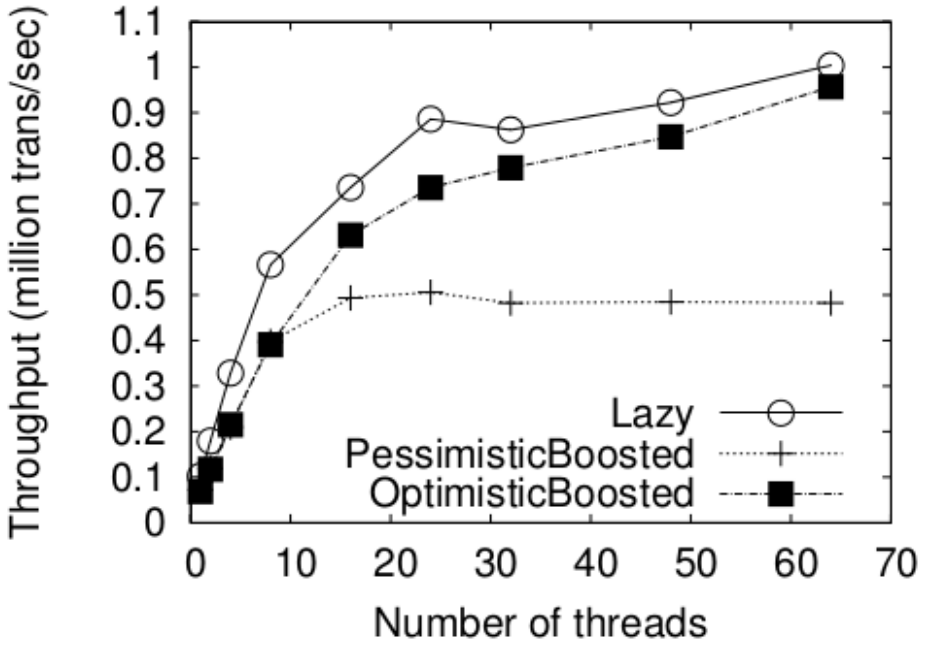
- Example optimizations on Linked-List and Skip-List
  - Optimizing Unsuccessful add/remove operations
    - Consider them as successful/unsuccessful contains.
    - No need for having write-set entries.
    - Possible because at commit time we know everything about the operation.

Results



Skip-list 512 Nodes

5 ops/transaction



Skip-list 64K Nodes

5 ops/transaction

Thanks!

Questions?

Conclusions

- Moving from “concurrent” to “transactional” data structures is important to support composability and integration
- Previous solutions (e.g. STM, pessimistic boosting) are inefficient and/or non-programmable.
- OTB solves this issue by boosting concurrent lazy data structures to be transactional.
- OTB provide guidelines for designing
  - General (non-optimized) version
  - Data structure specific (optimized) version.