Integrating Transactionally Boosted Data Structures with STM Frameworks: A Case Study on Set

Ahmed Hassan, Roberto Palmieri, Binoy Ravindran

Virginia Tech

TRANSACT 2014

State of art

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

What about Transactional data structures?

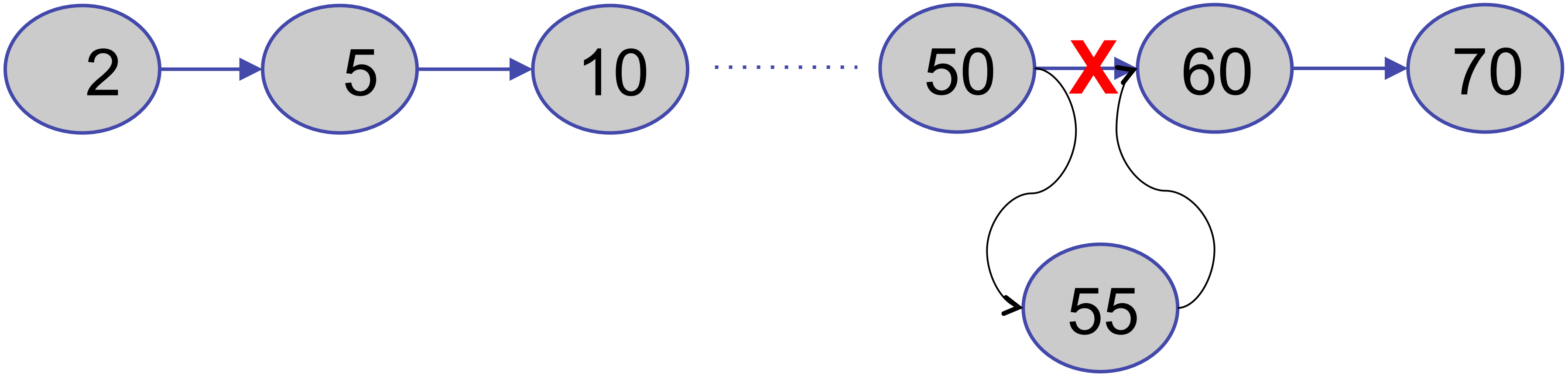
What about Transactional data structures?

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

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

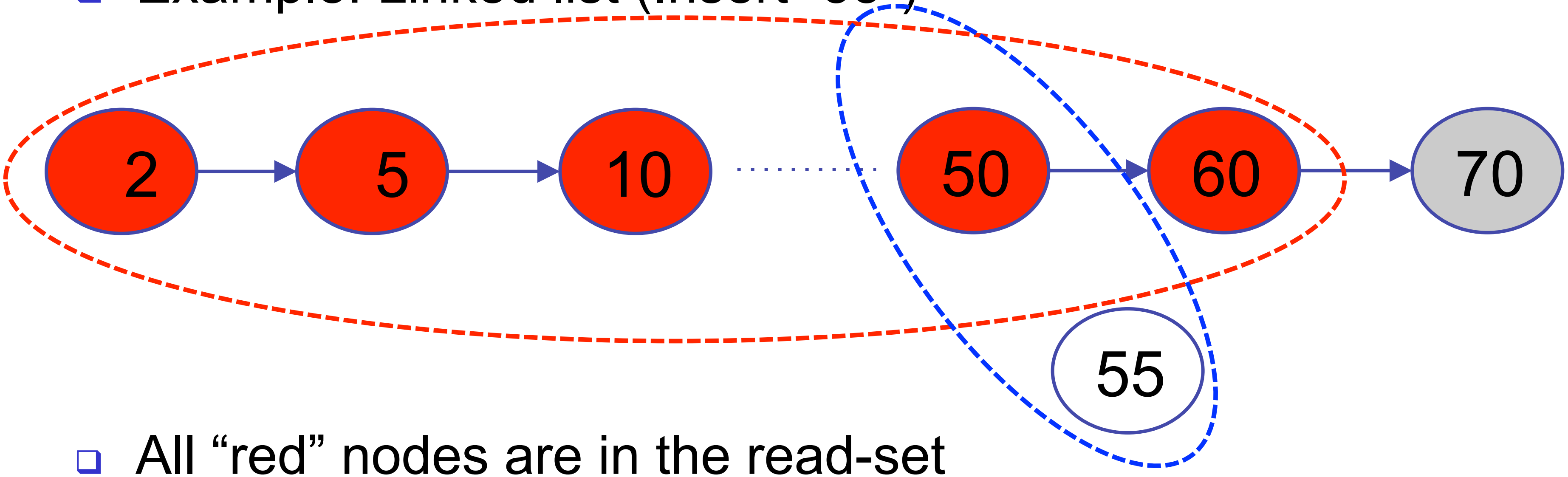
False Conflict

- 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

Solution for transactional data structures

- Solution: Transactional Boosting [Herlihy PPoPP08]
  - Convert highly concurrent data structures to transactional ones

- Other trials:
  - Early release, Elastic transactions, …
  - …but programmability is hampered

Motivation by examples

- Example of pure memory accesses to shared objects:

```
Shared data: n1, n2

@Atomic
foo()
{
    n1++;
    n2++;
}
```

- Good:
  - Easy to program
  - Strong correctness and progress guarantees

Motivation by examples

- Example of pure memory accesses to shared data structure:

```
Shared data: boostedSet

foo(x)
{
        boostedSet.add(x);
}
```

- Good:
  - Transactional support
  - Optimized for:
    - ensuring high performance
    - minimum false conflicts

Motivation by examples

Having pure memory accesses and data structure accesses merged in the same transaction

```
Shared data: boostedSet, n1, n2

@Atomic
foo()
{
   if(boostedSet.add(x))
         n1++;
   else
         n2++;
}
```

Without an integrated support for allowing the coexistence of memory accesses and data structure accesses, boostedSet has to be a pure-STM set

What we propose

- An integrated framework enabling:
  - Application programmers to exploit in the same transaction both STM accesses, as well as data structure accesses, without paying the cost of monitoring in the STM all memory accesses due to data structure operations (thus solving the problem of false conflicts)
  - Protocol designers to leverage the proposed software architecture for embedding new optimized data structures and STM protocols, in a way they can coexist in the same transaction

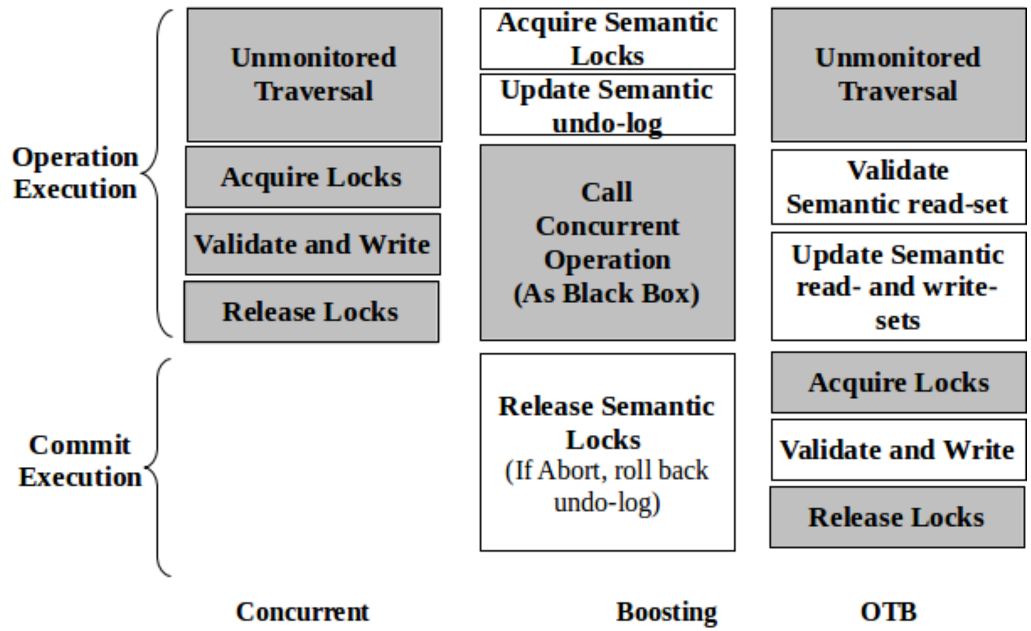
Design Choices

- As a guideline for implementing optimized transactional data structure, we adopt:
  - Optimistic Transactional Boosting (OTB) [PPoPP14]

- Why OTB?
  - OTB is an optimistic methodology for converting concurrent data structures into transactional, and it is designed to support integration with STM
  - OTB uses the concepts of Validation, Commit, and Abort in the same way as several (optimistic) STM algorithms
  - OTB allows data structure-specific optimizations

Lazy Vs Boosting Vs Optimistic Boosting

- Comparison among:
  - Concurrent Lazy data structures
  - Transactional data structures based on Original Boosting
  - Optimistic Transaction Boosting



Design Choices

- As a basic framework for the integration, we use DEUCE

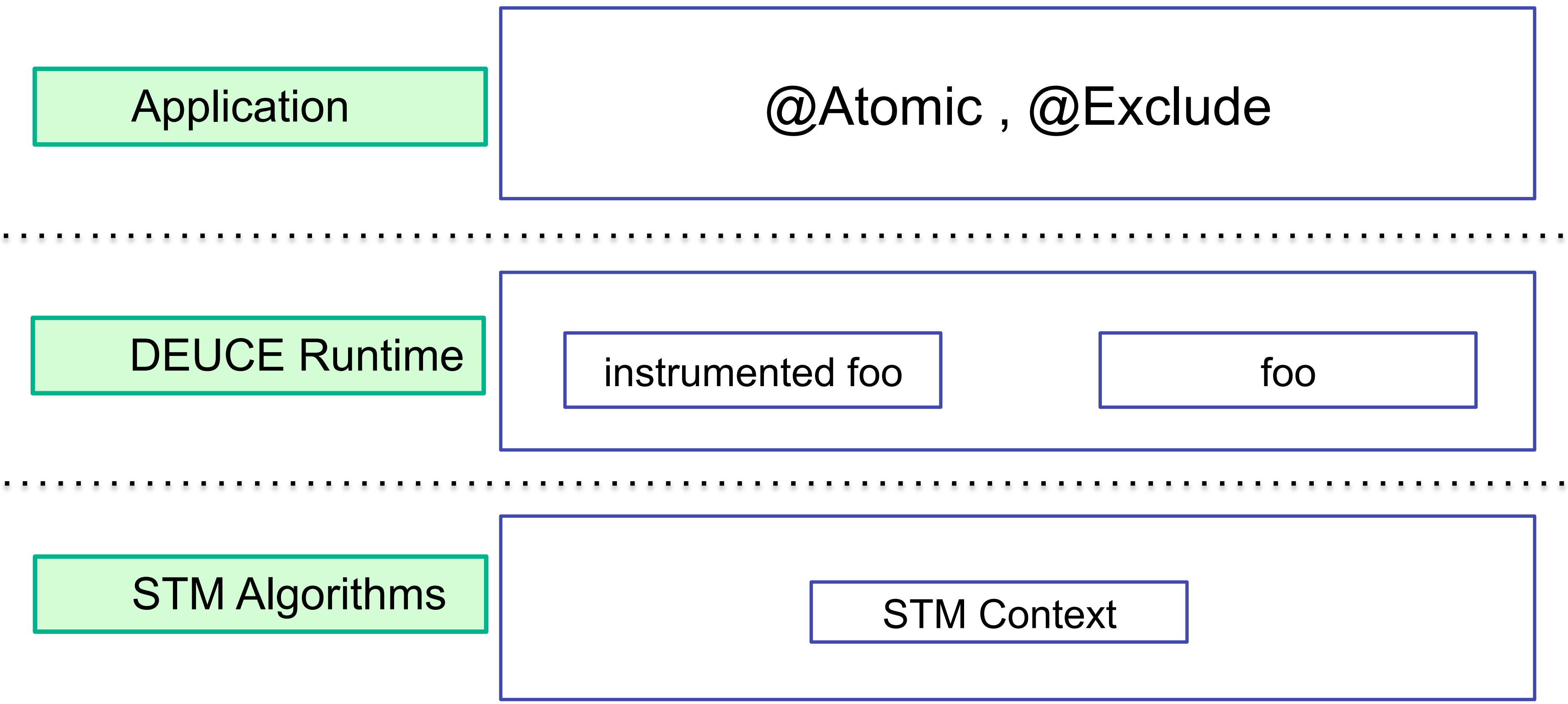
- Why DEUCE?
  - It is a Java STM framework with a simple interface
  - It already provides several STM algorithms

Our goals

- The design of our integrated solution has three main goals:
  - Keeping the simple programming interface of DEUCE
  - Allowing the integration between OTB data structures’ operations and memory reads/writes
  - Giving developers a simple API to plug-in their own OTB data structures and/or OTB-STM algorithms

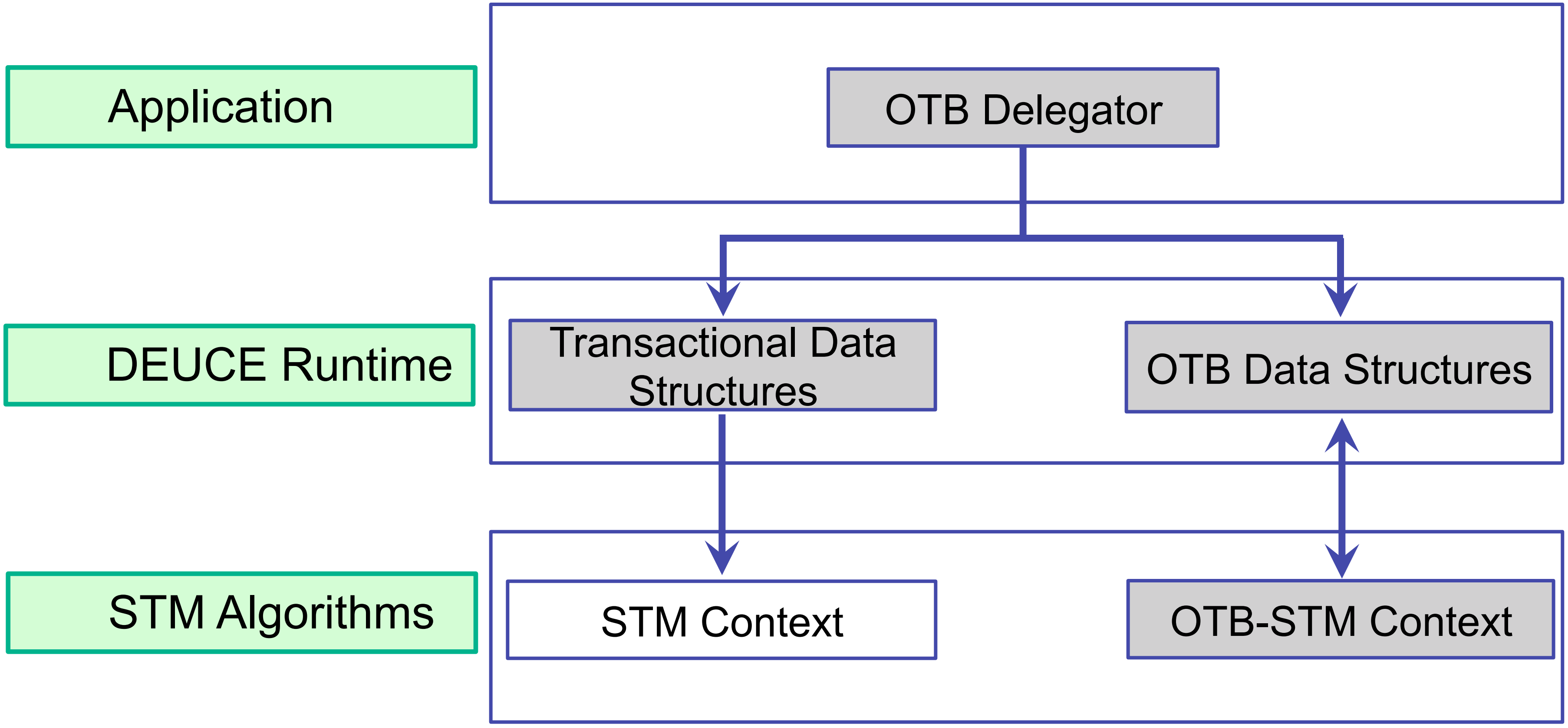
Framework Design

- The original DEUCE Framework:



Framework Design

- Our Additional Building Blocks:



Our case study

- In this paper we provide the integration of:
  - OTB Set, with
  - TL2, and
  - NOrec
- Other OTB data structures are presented in the technical report: ”Optimistic Transactional Boosting”, available at http://www.hyflow.org/pubs/ppopp_14_TR.pdf

OTB Set

- Design:
  - Semantic read-set: *pred*, *curr*, *operation*
  - Semantic write-set: *pred*, *curr*, *operation*, *newValue*
- Correctness:
  - Lazy (linearization): *pred* and *curr* are not deleted, and *pred* points to *curr*
  - STM (serialization): post-operation validation and commit validation
- Integration:
  - First Operation: *attachSet*
  - Validation: *validate-data*, *validate-data&locks*
  - Commit: *preCommit*, *onCommit*, *postCommit*

Integration with NOrec

- Integration with NOrec is simple:
  - both OTB set and NOrec validate the read-set after each operation and perform a value-based validation at commit
- NOrec uses a coarse-grain lock, thus acquiring fine-grain semantic locks is not needed
- Validation:
  - onReadAccess: call OTB set's *validate-data*
  - onOperationValidate: call NOrec's validation
- Commit:
  - Do not call set's *preCommit* and *postCommit* during transaction commit
  - Do not call set's *onAbort* during transaction abort

Integration with TL2

- Integration with TL2 requires the acquisition of fine-grain semantic locks
- Validation for OTB set is not value-based thus semantic locks are implemented as sequence locks.
- Validation
  - onReadAccess: call OTB set's *validate-data&locks*
  - onOperationValidate: Do nothing with TL2
- Commit:
  - Call set's *preCommit* and *postCommit* during transaction commit
  - Call set's *onAbort* during transaction abort

Generalization

- All the differences between the integration of NOrec and TL2 are due to optimizations

- We can generalize validation and commit for any STM algorithm (losing STM-specific optimizations, e.g. *validate-data* without checking locks)

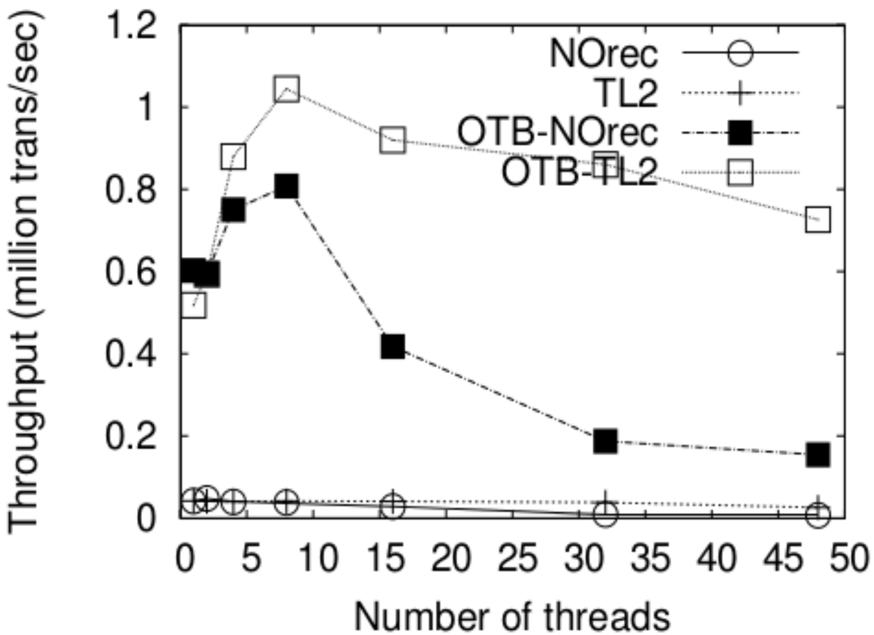
- Further investigation on the generalization is considered as future work

Performance Evaluation

- 48-core AMD Opteron machine
- 1400 MHz, 32 GB of memory, and 16KB L1 data cache.
- Average of 5 runs
- Warm-up phase of 2 seconds
- Execution phase of 5 seconds

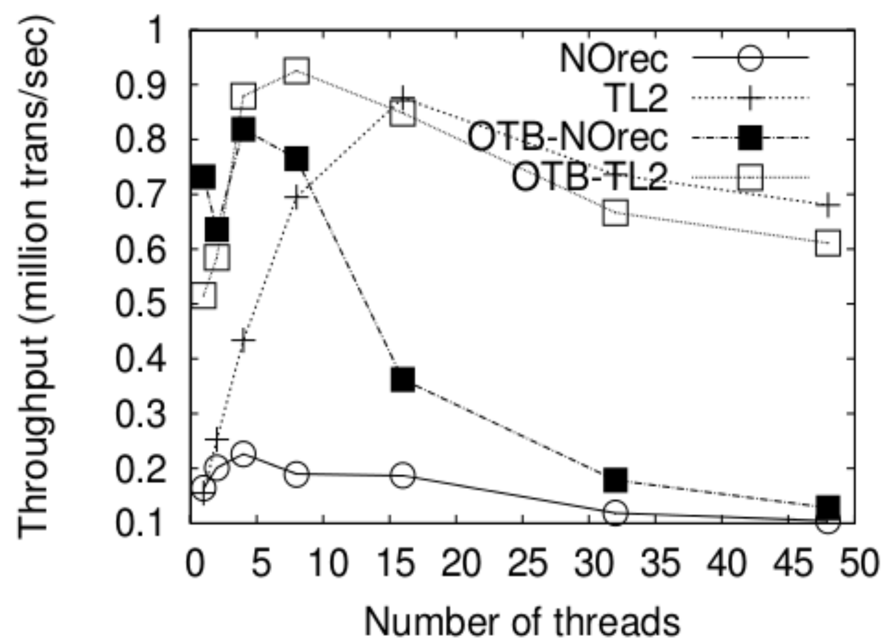
Performance Evaluation

- Micro-Benchmarks – without pure memory reads/writes



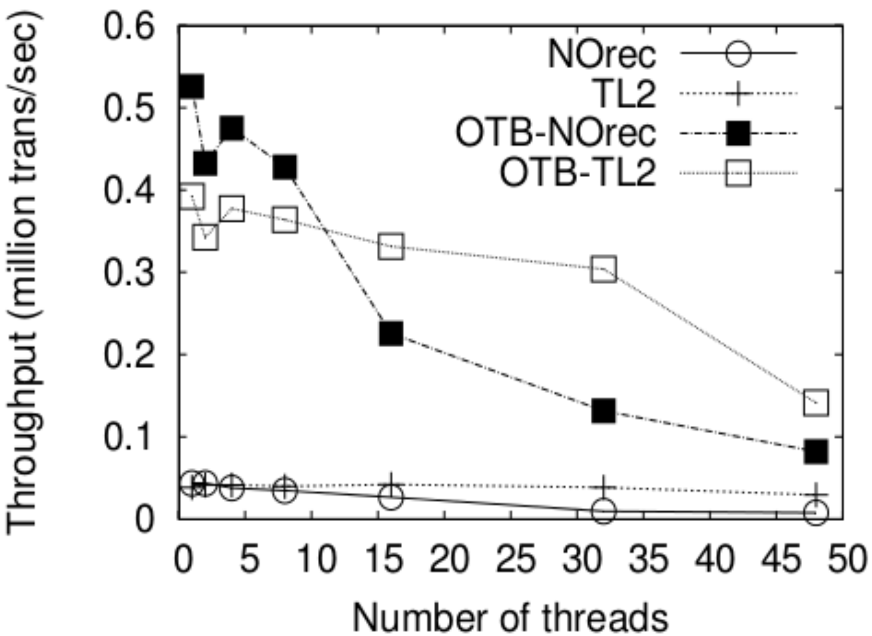
Linked List - 512 nodes - 50% reads

Skip List - 4K nodes - 50% reads



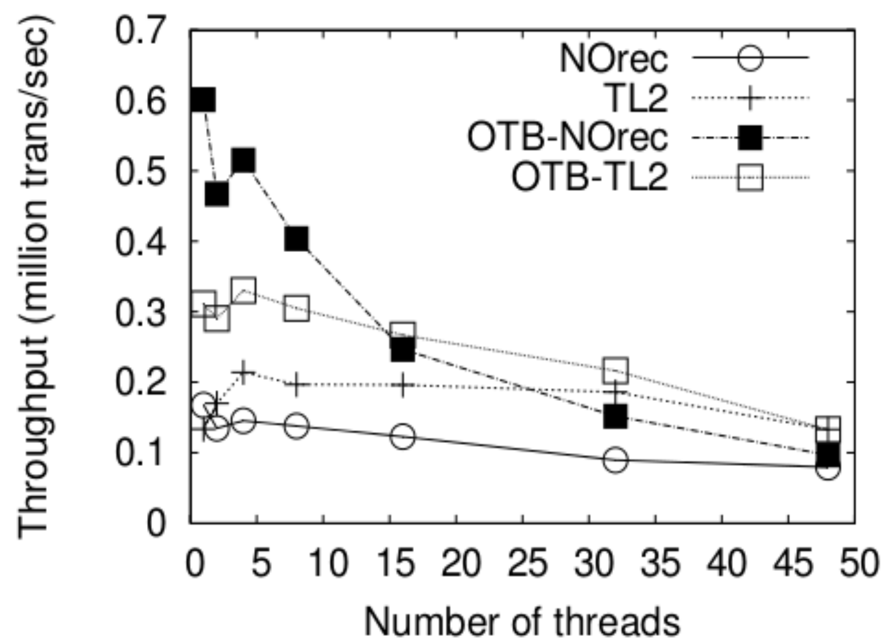
Performance Evaluation

- Micro-Benchmarks – with pure memory reads/writes



Linked List - 512 nodes - 50% reads

Skip List - 4K nodes - 50% reads



Thanks!

Questions?