Reducing STM Overhead with Access Permissions

Nels Beckman, Yoon Phil Kim, Sven Stork and Jonathan Aldrich

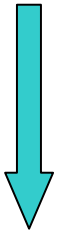
This Talk…

- Transactional memory is great…
  - But it has a high overhead
- Access permissions
  - Tell us about aliasing (unique, immutable, etc.)

- So we'll use them to
  - Remove logging, remove synchronization
  - Permissions tell us that some objects don't need protection or can be protected by other objects

Transactional Memory is Great

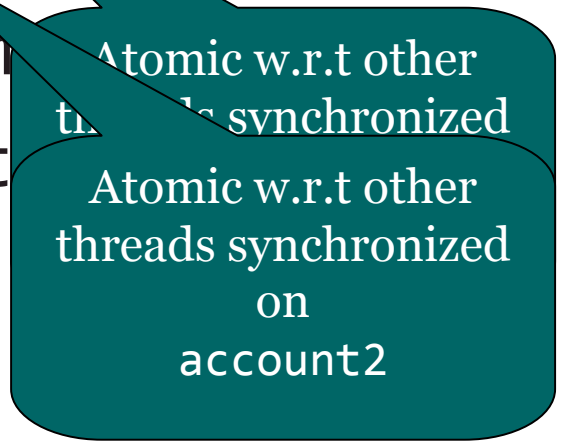
```
atomic {
  account1.withdraw(amt);
  account2.deposit(amt);
}
```



Executes as if no other threads

Synchronized Blocks are Tougher

```
synchronized(account1) {
  synchronized(account2) {
    account1.withdraw(am
    account2.deposit(amt
  }
}
```



But it Has High Overhead

- Typical implementation:
  - Transactional Memory
    1. Optimistically run thread
    2. Record memory reads and writes
    3. Roll back if inconsistent memory view
- Our control implementation*:
  - Optimistic reads, pessimistic writes
  - Object granularity
  - In-place updates
  - Weak atomicity

* A.-R. Adl-Tabatabai, et al. PLDI 2006.

Control Implementation: Details

- Thread in Transaction must:
  - *Own* an object to modify it
    - Before every write, calls into runtime system
      - "Open object for writing"
      - (Sets 'owned' flag, adds to write set, makes a copy)
  - Can read an object if owner or unowned
    - Before every read, calls into runtime system
      - "Open object for reading"
      - (Checks 'owned' flag, adds to read set)

Overhead Legend:
Memory barrier operation
Logging operation

Control Implementation: Commits and Conflict Detection

- At TXN commit-time:
  - If thread saw consistent view of memory
    - For each object in write set
      - Increase version number
      - Reset owner field
    - Clear read, write sets
  - Else perform *managed* back-off
    - Reinstall initial value for each modified object
- Thread detects inconsistent reads:
  - When version # of object in read set < current version in memory.

This Talk…

- Transactional memory is great…
  - But it has a high overhead
- **Access permissions**
  - **Tell us about aliasing (unique, immutable, etc.)**
- So we'll use them to
  - Remove logging, remove synchronization
  - Permissions tell us that some objects don't need protection or can be protected by other objects

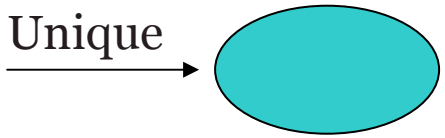
Access Permissions* Tell Us About Aliasing

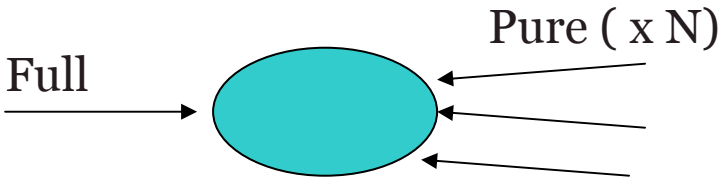
- Type Annotations on references
  - The type of a reference encodes:
    - Can this object be modified?
    - Is this object aliased?
  - Provided by developer
- It's a type, so it's checked for consistency
- (We use them to modularly verify typestate in concurrent programs*.)

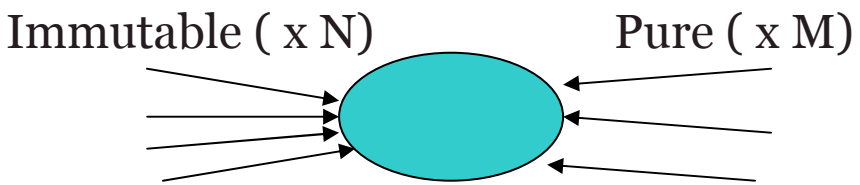
*Bierhoff, Aldrich. OOPSLA 2007.
*Beckman, Bierhoff, Aldrich. OOPSLA 2008.

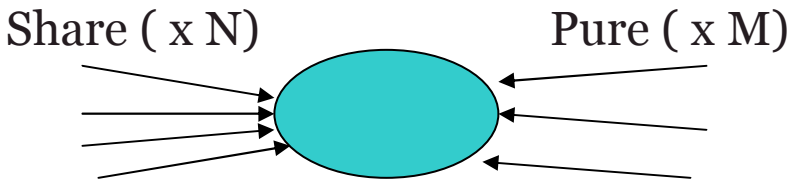
Access Permission Kinds

- Unique, Full, Immutable, Pure, Share









Approach: Optimizing Implementation

- Modify our control implementation of STM
  - (A source-to-source translation)
  - Remove calls to run-time (open for read/open for write) when unnecessary
  - Sometimes insert new calls for soundness

Approach

- 4 Rules

Rule 1: Never open **Immutable** refs for reading.

  - Removes owner check.
  - Item not in read set.

Approach

- 4 Rules

Rule 2: Don't test-and-set owner when writing to **Unique** objects. (But record value in case of undo.)

Approach

- 4 Rules

Rule 3: Never open **Unique** or **Full** refs for reading.

  - Removes owner check.
  - Item not in read set.

Approach

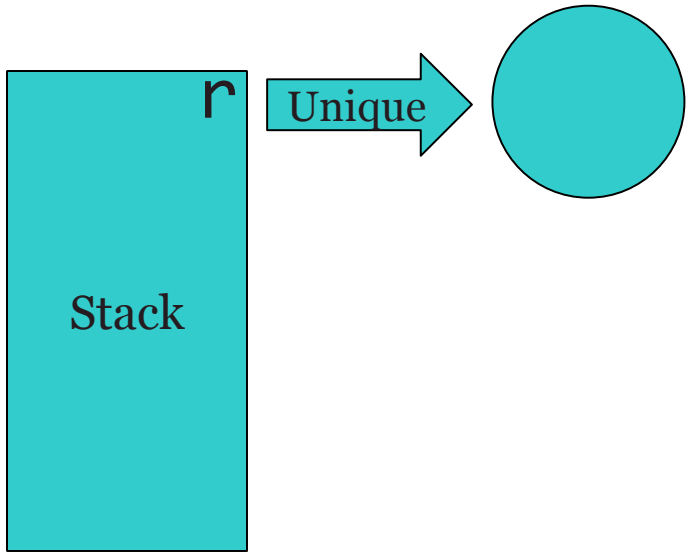
- 4 Rules

Rule 4: In order to make the above rules sound:

- Open **Share**, **Pure** or **Full** ref for writing…
- When using permission to a **Unique** or **Full** field of that object.
- Performs test-and-set and adds to write set.

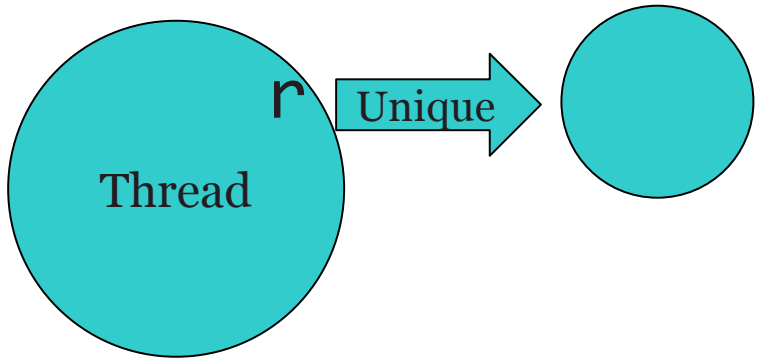
Discussion

- Opening outer object for writing creates 'zone of protection'



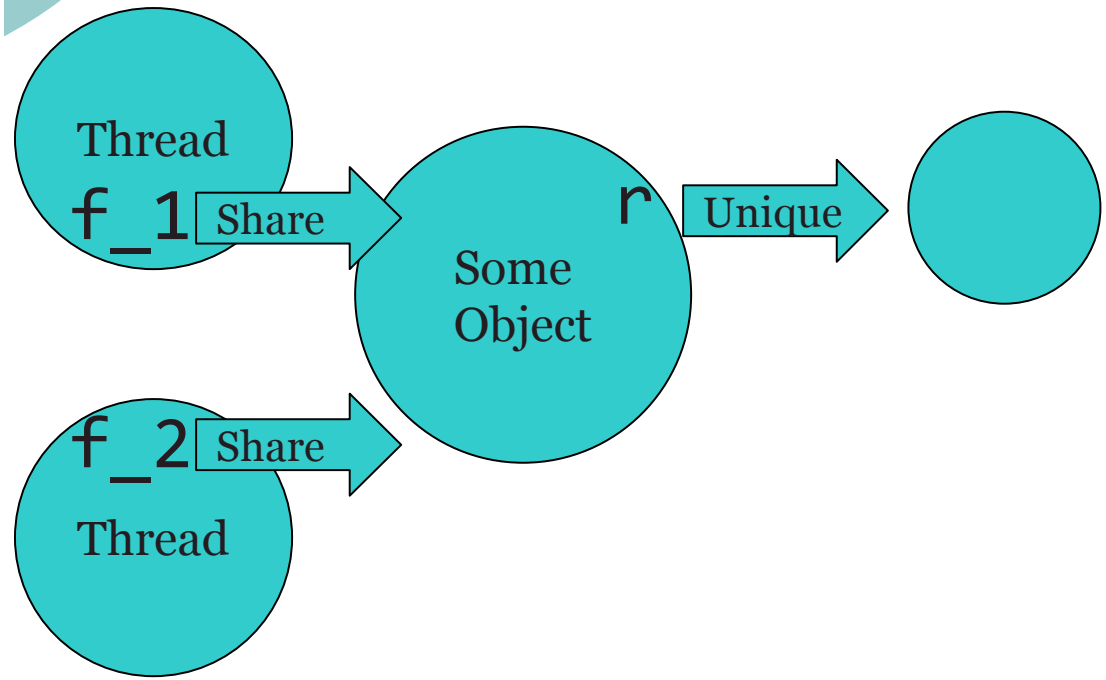
Discussion

- Opening outer object for writing creates 'zone of protection'



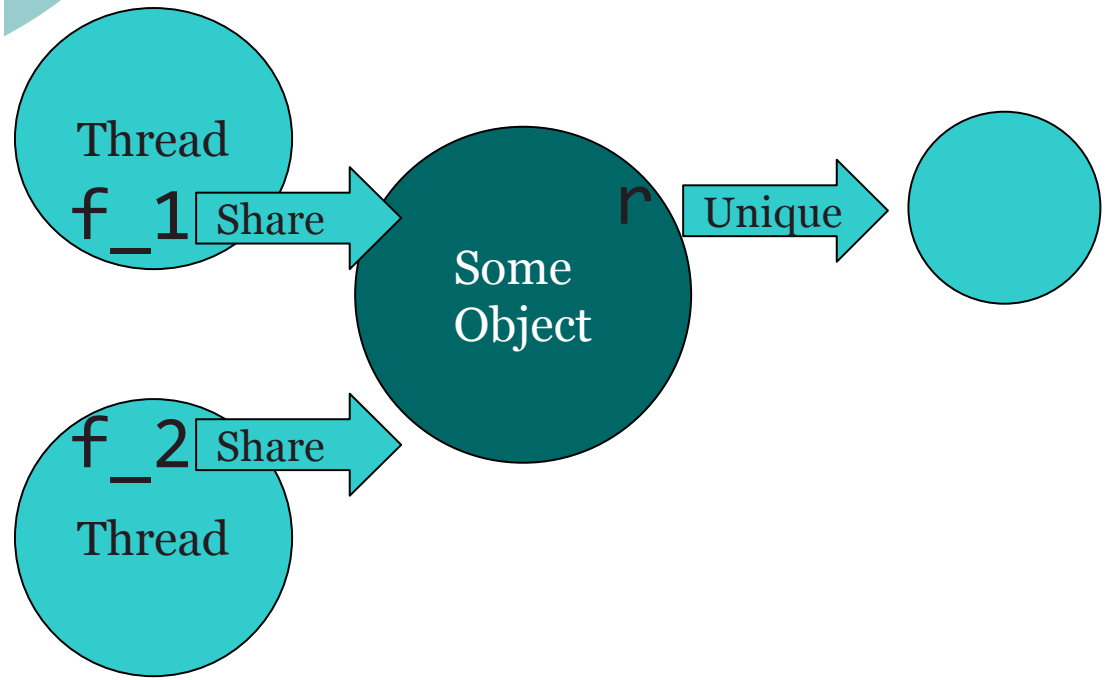
Discussion

- Opening outer object for writing creates 'zone of protection'



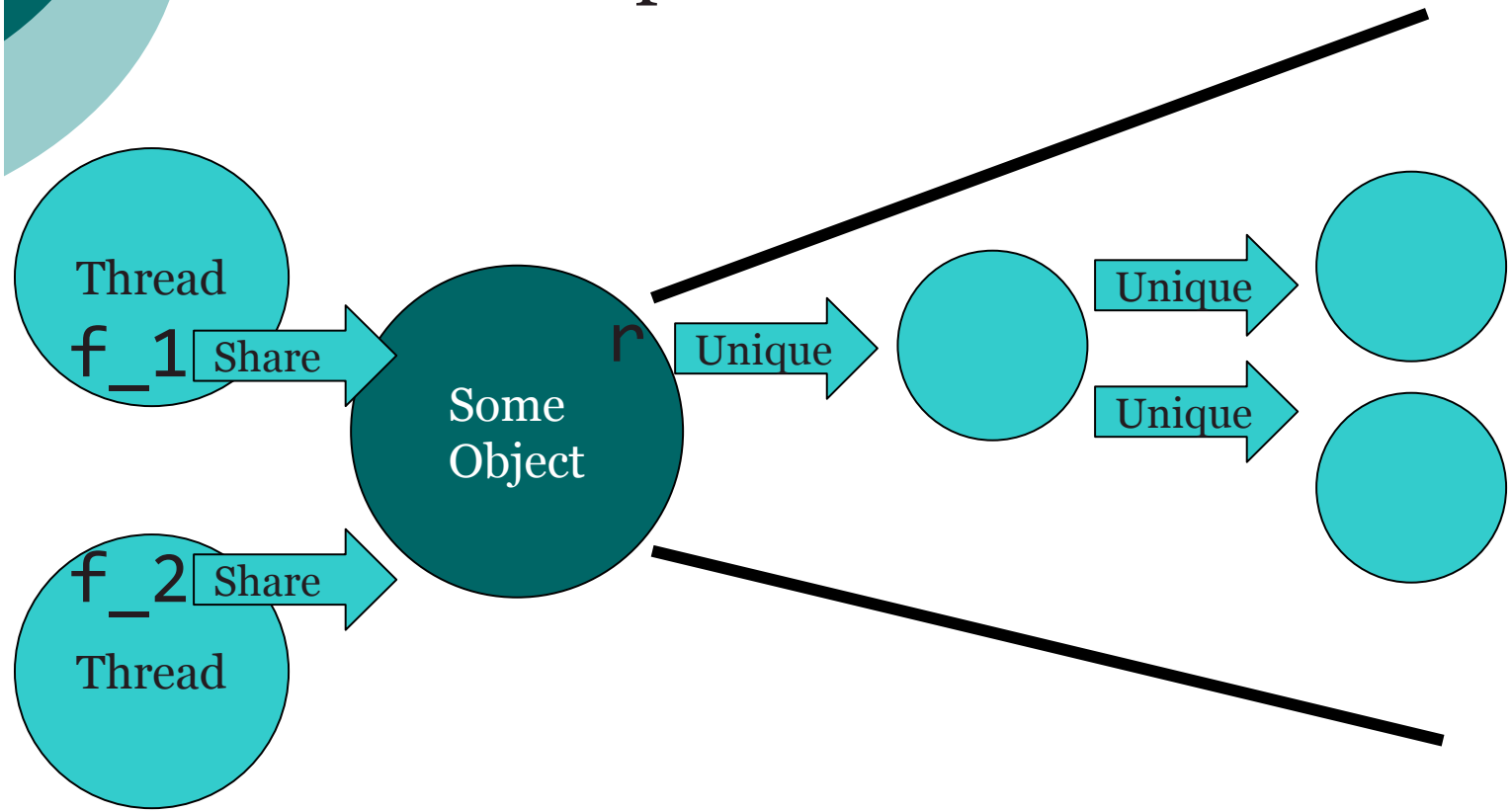
Discussion

- Opening outer object for writing creates 'zone of protection'



Discussion

- Opening outer object for writing creates 'zone of protection'



Discussion

- Opening outer object for writing creates 'zone of protection'
  - Plus
    - Lower overhead
  - Minus
    - Larger granularity

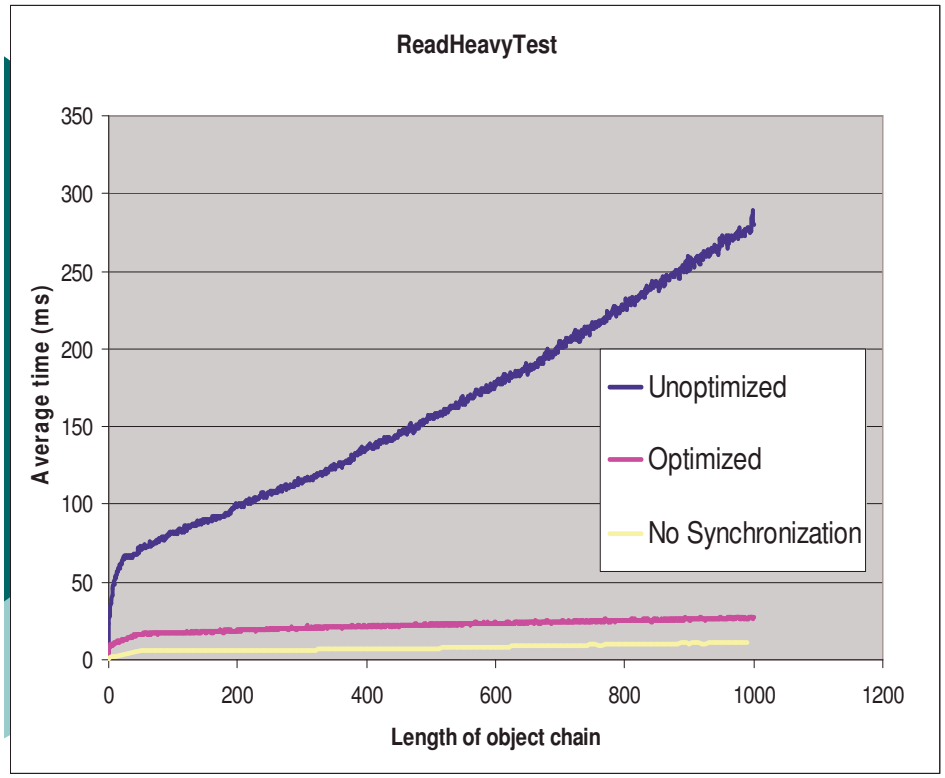
Approach: Recap

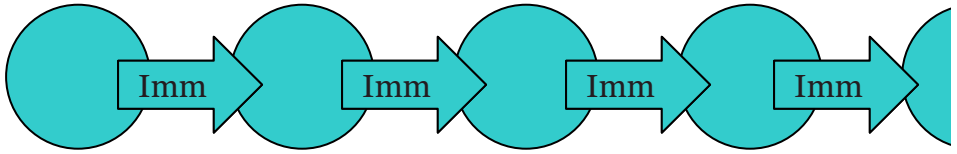
1. Immutable: Don't open for read.
2. Unique: Create undo entry but no synch.
3. Unique/Full: Don't open for read.
4. Full/Unique field of Share/Pure/Full object: Open outer object for writing.

Evaluation

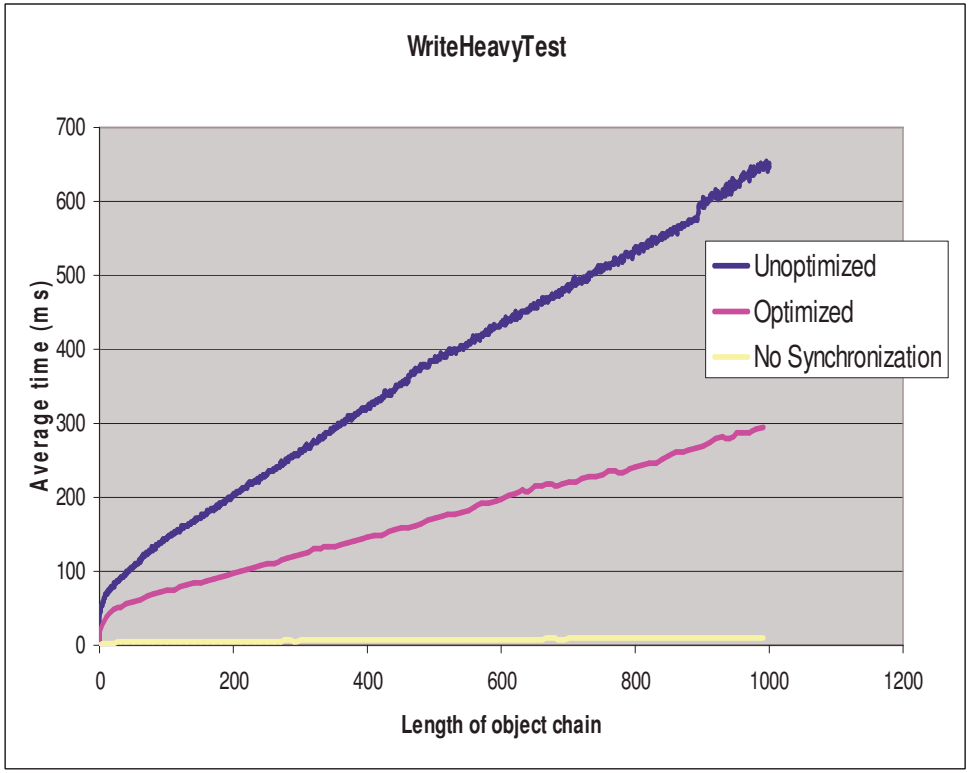
- Specify permissions & check consistency
  - Several small benchmarks
  - One larger video game
    - (Required adding atomic blocks)
- Measure benchmark performance with and without optimization.

Results

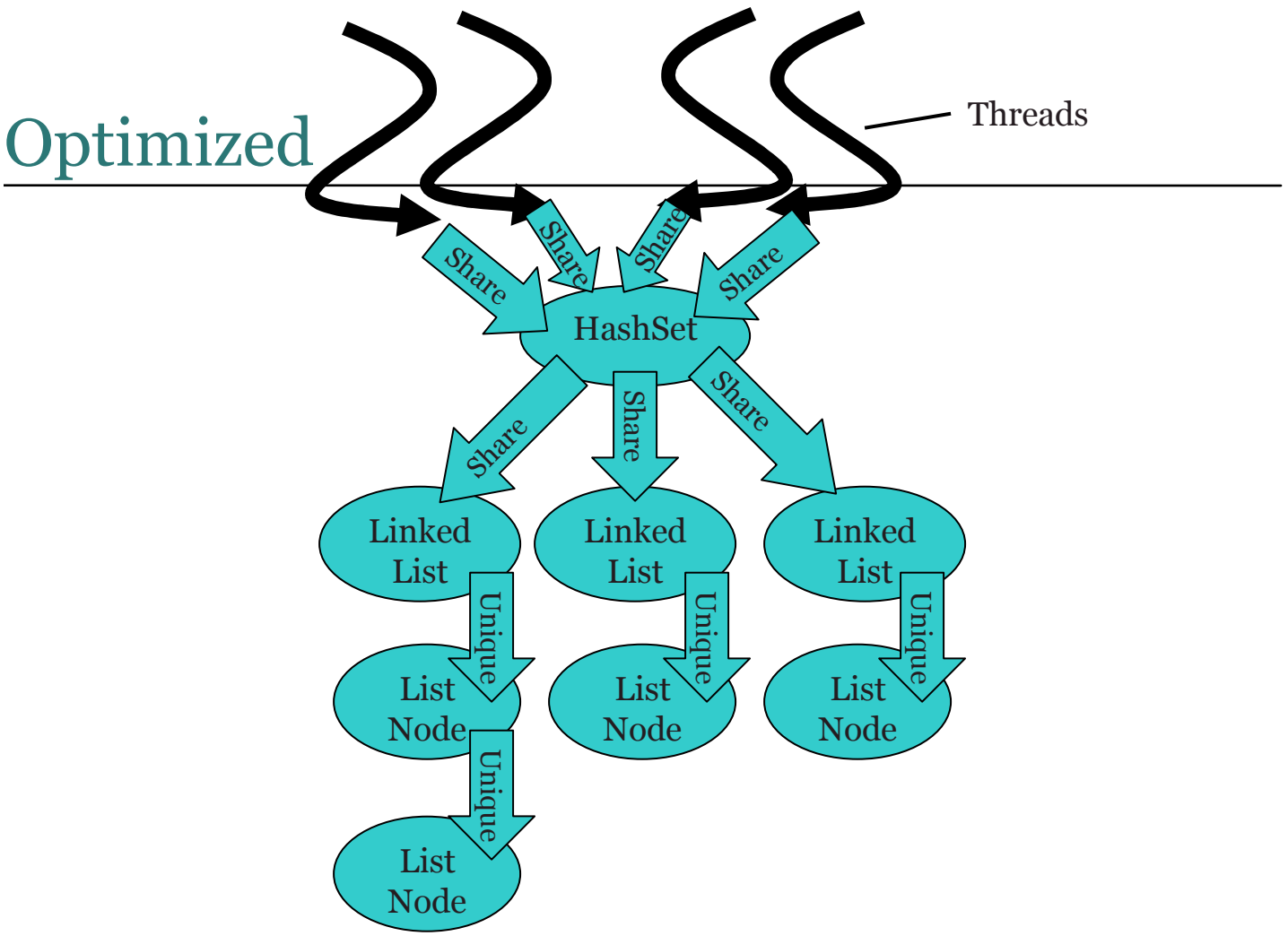




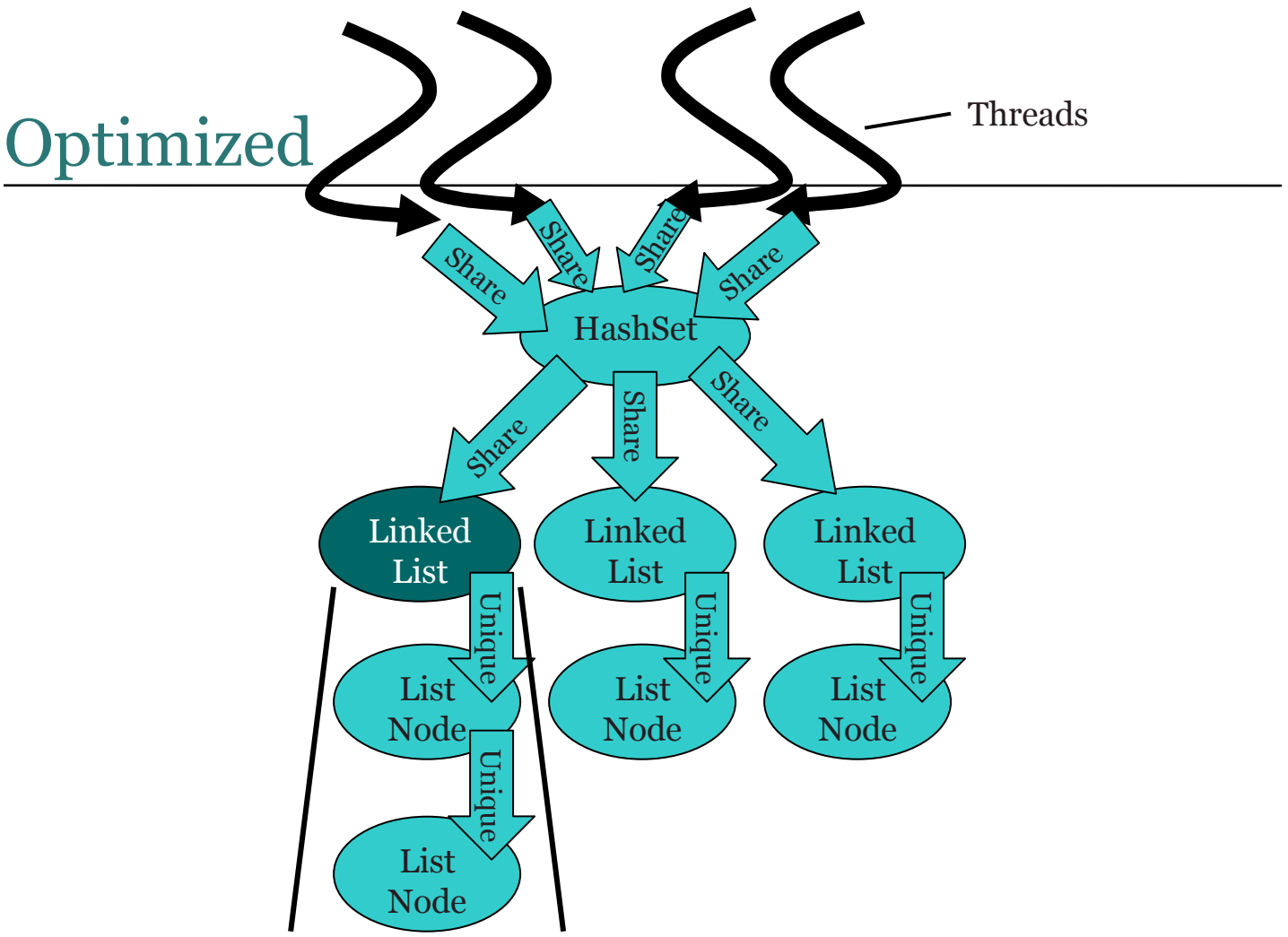
ReadHeavyTest & WriteHeavyTest



HashSet (w/ linked buckets)

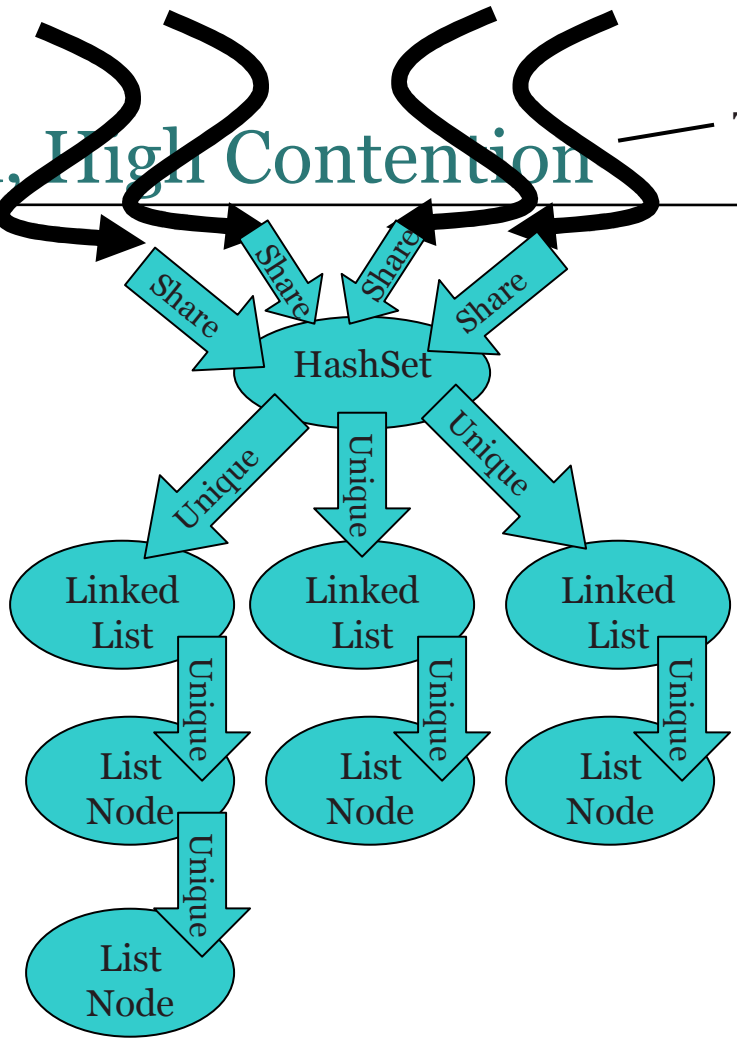


HashSet (w/ linked buckets)



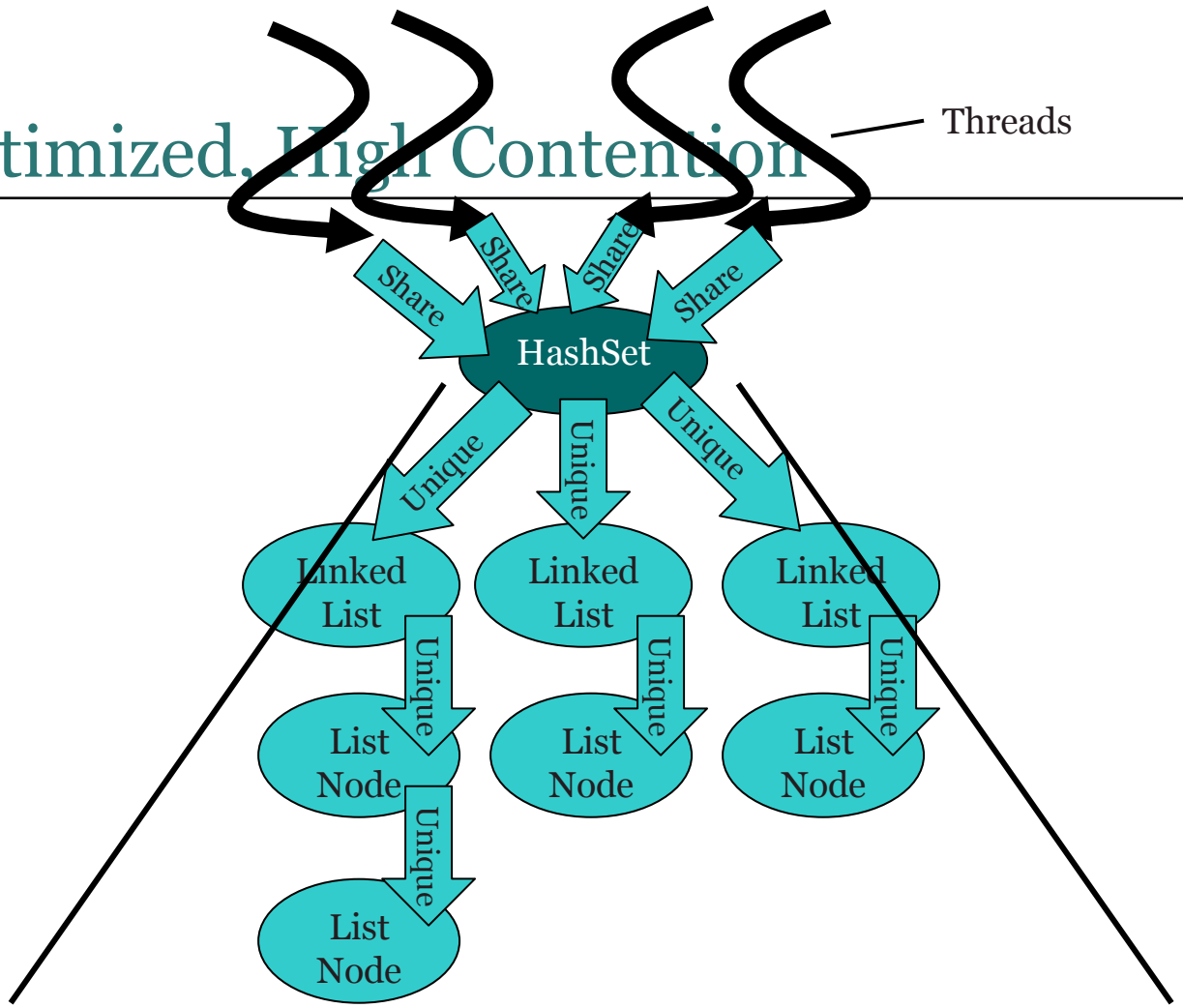
HashSet (w/ linked buckets)

Optimized, High Contention

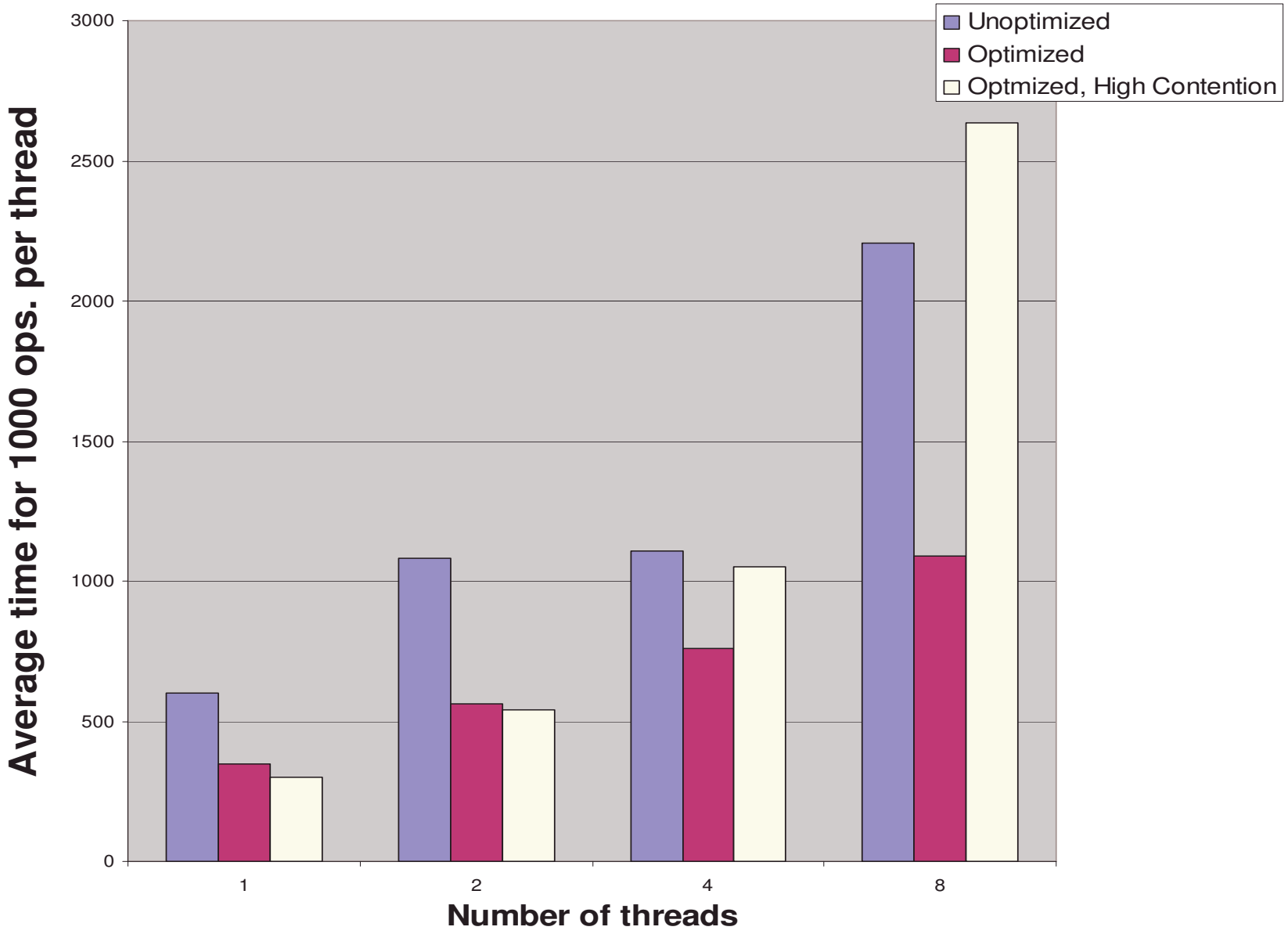


HashSet (w/ linked buckets)

Optimized, High Contention

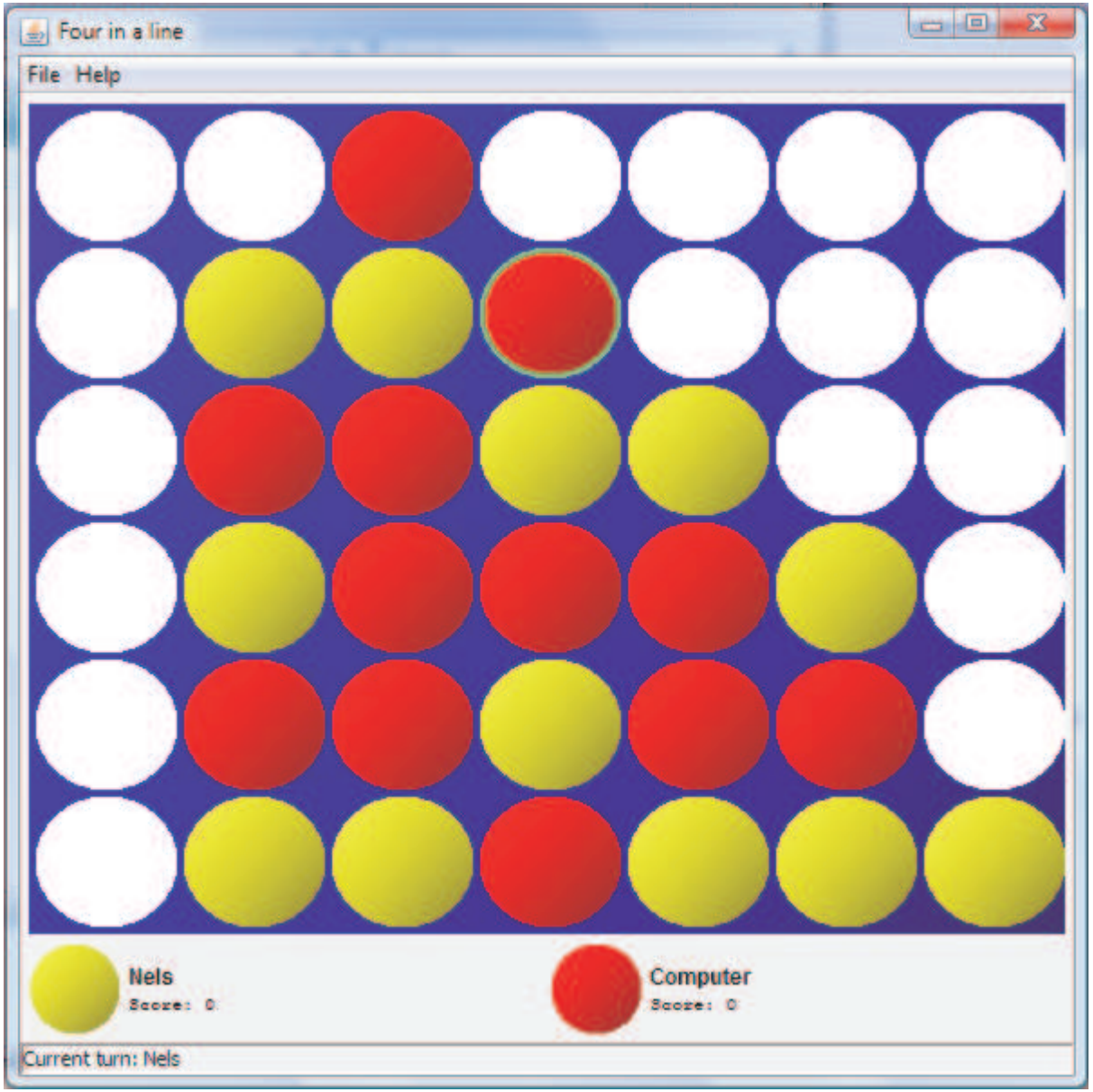


HashSet (w/ linked buckets)



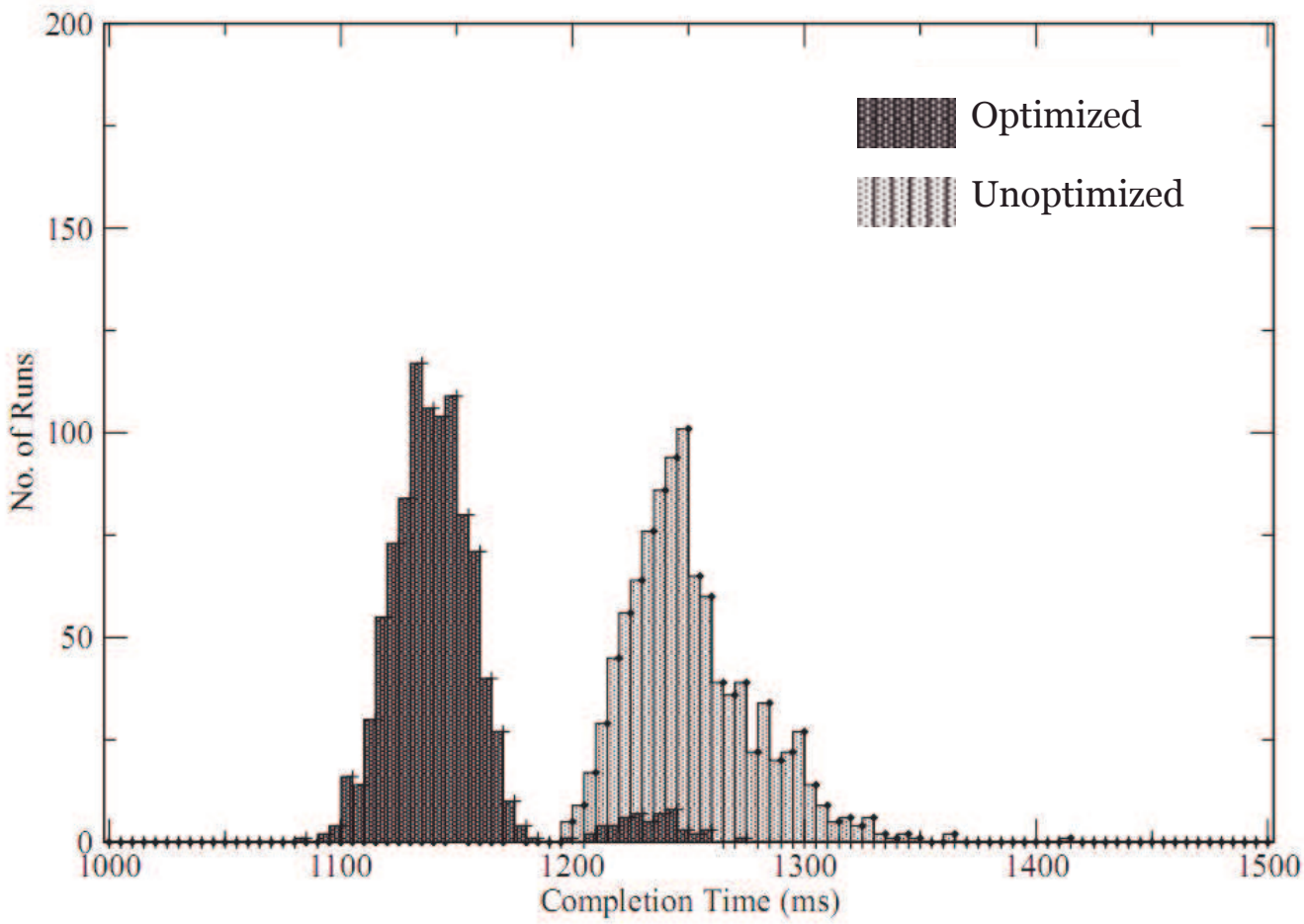
HashSet (w/ linked buckets)

4InALine


Four in a line
File Help
Nels
Score: 0
Computer
Score: 0
Current turn: Nels

4InALine

9.3% Performance Improvement



Conclusion

- Access permissions:
  - Modular description of aliasing
  - Tell us certain objects cannot be concurrently modified or are immutable
  - Thus we can reduce TM overhead

- We're not asking everyone to use access permissions
  - But if you're already verifying typestate…
  - Performance improvement is free