
Konrad Siek

# Distributed Pessimistic Transactional Memory: Algorithms and Properties

Public Doctoral Dissertation Defense

| | |
|---|---|
| Advisor: | Dr. Paweł T. Wojciechowski, PUT |
| Reviewers: | Prof. Marek Tudruj, IPI PAN |
| | Prof. Michel Raynal, IRISA, Univ. Rennes |

Poznań, 4 I 2017

# Transactional Memory

Locks, barriers, semaphores, monitors, etc.:

- interactions among disparate threads
- additional, invasive structural code
- error prone: deadlocks, livelocks, race conditions, ...

```
void move(Stack a, Stack b) {
  if (a.count() >= 1) {
    obj = a.pop();
    b.push(obj);
  }
}
```

```
void move(Stack a, Stack b) {
  a.lock.lock();
  b.lock.lock();
  if (a.count() >= 1) {
    obj = a.pop();
    a.lock.unlock();
    b.push(obj);
  } else
    a.lock.unlock();
  b.lock.unlock();
}
```

```
void move(Stack a, Stack b) {
  transaction.start();
  if (a.count() >= 1) {
    obj = a.pop();
    b.push(obj);
  }
  transaction.commit();
}
```

Transactional memory (TM):

- transaction abstraction for general-purpose computing
- easy to use (automation) and understand (properties)
- efficient implementation under the hood
- applicable to distributed systems

Herlihy, Moss. *Transactional Memory: Architectural Support for Lock-free Data Structures*. ISCA'93.

# Optimistic Concurrency Control

Speculative execution with abort capability

# Optimistic Concurrency Control

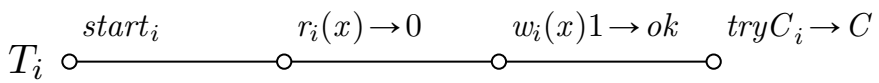
Speculative execution with abort capability

```
transaction_i.start();
int value = x.read();
x.write(value + 1);
transaction_i.commit();
```

# Optimistic Concurrency Control

Speculative execution with abort capability

```
transaction_i.start();
int value = x.read();
x.write(value + 1);
transaction_i.commit();
```

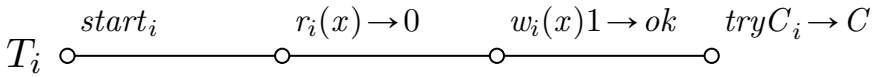


# Optimistic Concurrency Control

Speculative execution with abort capability

```
transaction_i.start();
int value = x.read();
x.write(value + 1);
transaction_i.commit();
```

```
transaction_j.start();
int value = x.read();
x.write(value + 1);
transaction_j.commit();
```

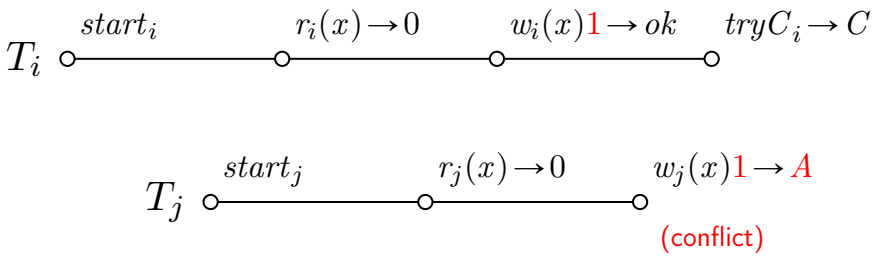


# Optimistic Concurrency Control

Speculative execution with abort capability

```
transaction_i.start();
int value = x.read();
x.write(value + 1);
transaction_i.commit();
```

```
transaction_j.start();
int value = x.read();
x.write(value + 1);
transaction_j.commit();
```

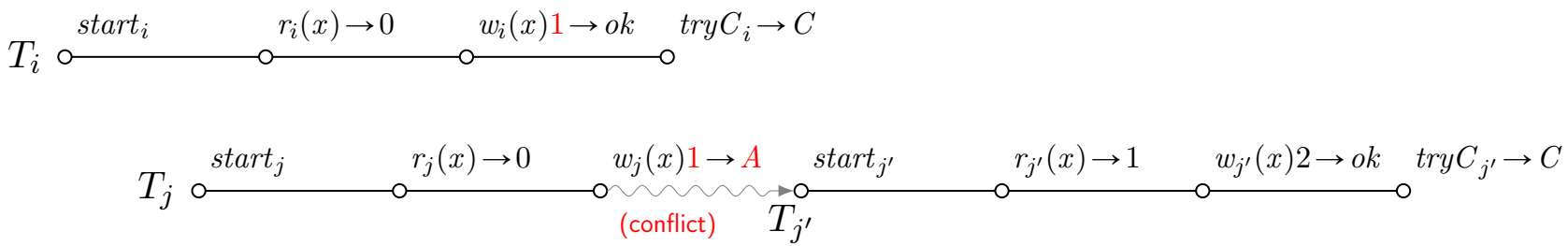


# Optimistic Concurrency Control

Speculative execution with abort capability

```
transaction_i.start();
int value = x.read();
x.write(value + 1);
transaction_i.commit();
```

```
transaction_j.start();
int value = x.read();
x.write(value + 1);
transaction_j.commit();
```



# Problems with Optimistic TM

Speculative execution of irrevocable operations

# Problems with Optimistic TM

Speculative execution of irrevocable operations

Irrevocable operations:

- have visible side effects
- effects cannot be withdrawn (must be compensated)
- examples: network communication, locks, system calls, I/O
- occur often in complex and distributed applications

# Problems with Optimistic TM

Speculative execution of irrevocable operations

Irrevocable operations:

- have visible side effects
- effects cannot be withdrawn (must be compensated)
- examples: network communication, locks, system calls, I/O
- occur often in complex and distributed applications

```
transaction_i.start();
int value = x.read();
System.out.println("incrementing");
x.write(value + 1);
transaction_i.commit();
```

```
transaction_j.start();
int value = x.read();
System.out.println("incrementing");
x.write(value + 1);
transaction_j.commit();
```

# Problems with Optimistic TM

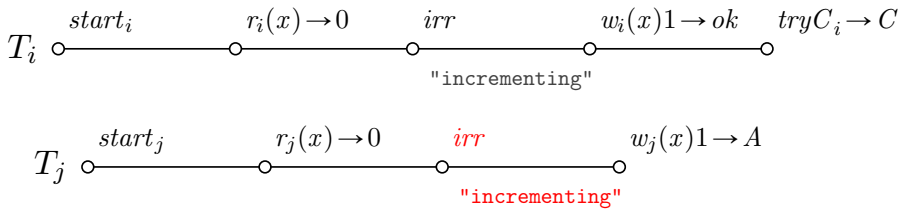
Speculative execution of irrevocable operations

Irrevocable operations:

- have visible side effects
- effects cannot be withdrawn (must be compensated)
- examples: network communication, locks, system calls, I/O
- occur often in complex and distributed applications

```
transaction_i.start();
int value = x.read();
System.out.println("incrementing");
x.write(value + 1);
transaction_i.commit();
```

```
transaction_j.start();
int value = x.read();
System.out.println("incrementing");
x.write(value + 1);
transaction_j.commit();
```



# Problems with Optimistic TM

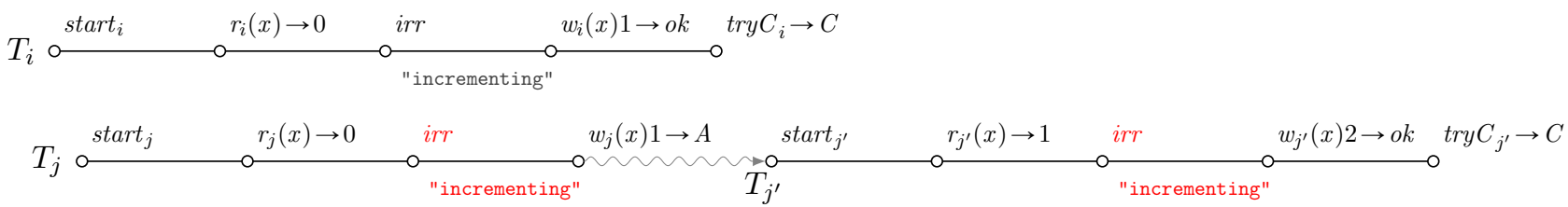
Speculative execution of irrevocable operations

Irrevocable operations:

- have visible side effects
- effects cannot be withdrawn (must be compensated)
- examples: network communication, locks, system calls, I/O
- occur often in complex and distributed applications

```
transaction_i.start();
int value = x.read();
System.out.println("incrementing");
x.write(value + 1);
transaction_i.commit();
```

```
transaction_j.start();
int value = x.read();
System.out.println("incrementing");
x.write(value + 1);
transaction_j.commit();
```



# Problems with Optimistic TM

## Speculative execution of irrevocable operations (2)

```
transaction_i.start();

FileChannel channel = /* ... */;
FileLock lock = channel.lock();
/* File I/O */

int value = x.read();
x.write(value + 1);

/* File I/O */
lock.unlock();

transaction_i.commit();
```

```
transaction_j.start();

FileChannel channel = /* ... */;
FileLock lock = channel.lock();
/* File I/O */

int value = x.read();
x.write(value + 1);

/* File I/O */
lock.unlock();

transaction_j.commit();
```

# Problems with Optimistic TM

## Speculative execution of irrevocable operations (2)

```
transaction_i.start();

FileChannel channel = /* ... */;
FileLock lock = channel.lock();
/* File I/O */

int value = x.read();
x.write(value + 1);

/* File I/O */
lock.unlock();

transaction_i.commit();
```

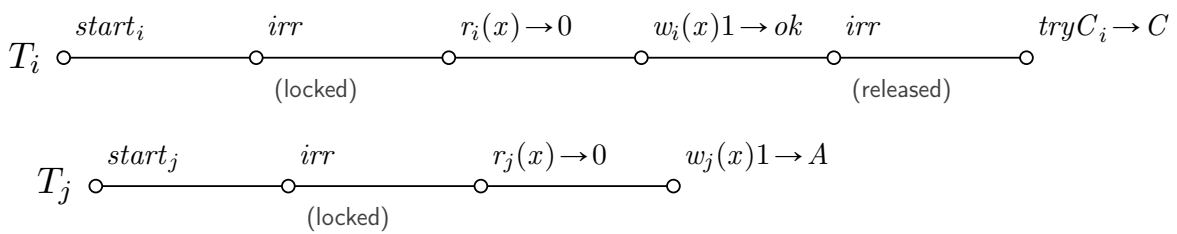
```
transaction_j.start();

FileChannel channel = /* ... */;
FileLock lock = channel.lock();
/* File I/O */

int value = x.read();
x.write(value + 1);

/* File I/O */
lock.unlock();

transaction_j.commit();
```



# Problems with Optimistic TM

## Speculative execution of irrevocable operations (2)

```
transaction_i.start();

FileChannel channel = /* ... */;
FileLock lock = channel.lock();
/* File I/O */

int value = x.read();
x.write(value + 1);

/* File I/O */
lock.unlock();

transaction_i.commit();
```

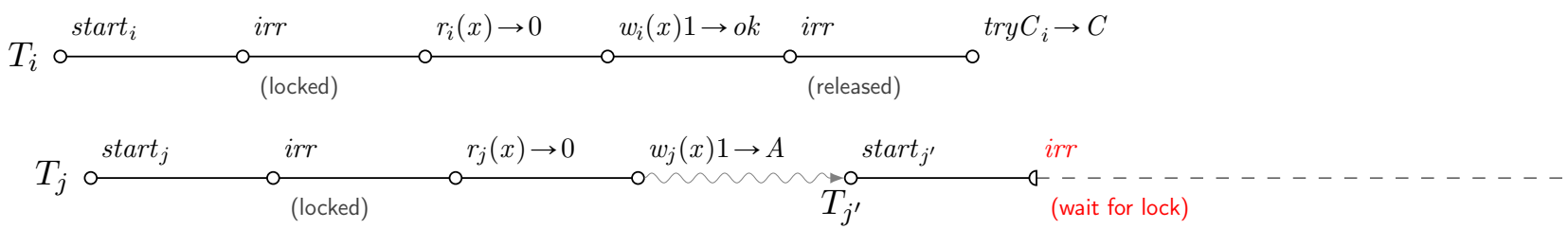
```
transaction_j.start();

FileChannel channel = /* ... */;
FileLock lock = channel.lock();
/* File I/O */

int value = x.read();
x.write(value + 1);

/* File I/O */
lock.unlock();

transaction_j.commit();
```



# Solutions

Welc, Saha, Adl-Tabatabai. Irrevocable Transactions and their Applications. SPAA'08.

Bocchino, Adve, Chamberlain. Software Transactional Memory for Large Scale Clusters. PPoPP'08.

Attiya, Hillel. Single-version STMs can be multi-version permissive. ICDCS'11.

Harris. Marlow. Jones. Herlihy. Composable Memory Transactions. PPoPP'05.

# Pessimistic Concurrency Control

Pessimistic TM:

- defer execution to prevent conflicts in place of speculation
- do not rely on aborting transactions

  $\therefore$ possible to avoid aborts altogether

# Pessimistic Concurrency Control

Pessimistic TM:

- defer execution to prevent conflicts in place of speculation
- do not rely on aborting transactions

  $\therefore$ possible to avoid aborts altogether

  $\therefore$ solves irrevocable operation problem (and others)

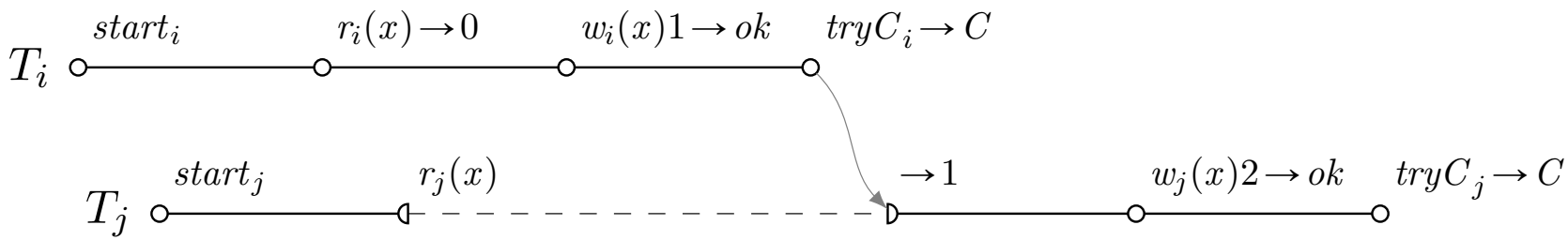
# Pessimistic Concurrency Control

Pessimistic TM:

- defer execution to prevent conflicts in place of speculation
- do not rely on aborting transactions

  $\therefore$ possible to avoid aborts altogether

  $\therefore$ solves irrevocable operation problem (and others)



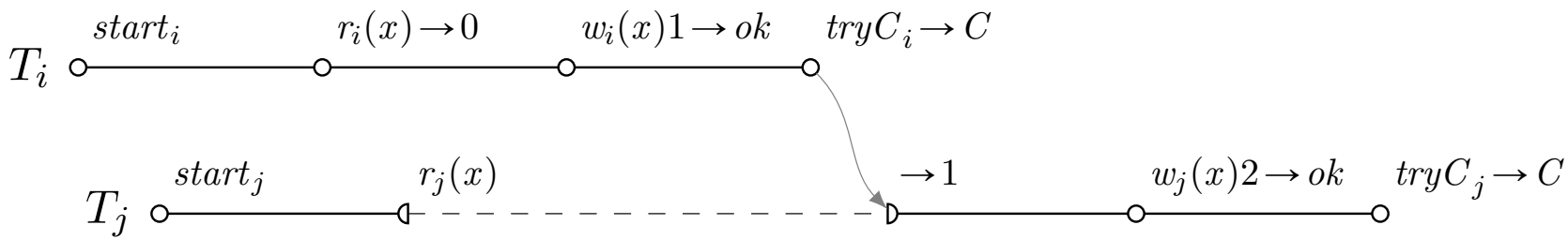
# Pessimistic Concurrency Control

Pessimistic TM:

- defer execution to prevent conflicts in place of speculation
- do not rely on aborting transactions

  $\therefore$ possible to avoid aborts altogether

  $\therefore$ solves irrevocable operation problem (and others)




Inferior performance to optimistic TM (serial execution of writers).

# Thesis

It is possible to propose a pessimistic TM concurrency control algorithm that:

- is safe
- is practical
- works in distributed systems
- executes irrevocable operations correctly
- and achieves high performance

# Related Work

Wojciechowski. Isolation-only Transactions by Typing and Versioning. PPDP'05.

Matveev, Shavit. Towards a Fully Pessimistic STM Model. TRANSACT'12.

Afek, Matveev, Shavit. Pessimistic Software Lock-Elision. DISC'12.

Crain, Imbs, Raynal. Towards a Universal Construction for Transaction-based Multiprocessor Programs. TCS vol. 496, 2013.

Avni, Dolev, Fatourou, Kosmas. Abort-free Semantic TM by Dependency Aware Scheduling of Transactional Instructions. NETYS'14.

Database transactional processing: two-phase locking.

# Related Work Analysis

| Algorithm | Approach | Progress | Updates | Aborts | A priori | Objects | DL | Safety | Release | Irrevocable |
|---|---|---|---|---|---|---|---|---|---|---|
| B2PL | pessimistic | blocking | encounter | on deadlock | ∅ | any | yes | strict serializable | yes | abortable |
| C2PL | pessimistic | blocking | encounter | abort-free | $RSet$, $WSet$ | any | no | strict serializable | yes | correct |
| S2PL | pessimistic | blocking | encounter | on deadlock | ∅ | any | yes | strict | reads | abortable |
| R2PL | pessimistic | blocking | encounter | on deadlock | ∅ | any | yes | rigorous | no | abortable |
| CS2PL | pessimistic | blocking | encounter | abort-free | $RSet$, $WSet$ | any | no | opaque | reads | correct |
| CR2PL | pessimistic | blocking | encounter | abort-free | $RSet$, $WSet$ | any | no | opaque | no | correct |
| CAS2PL | pessimistic | blocking | encounter | arbitrary abort | $RSet$, $WSet$ | any | no | opaque | reads | user abortable |
| CAR2PL | pessimistic | blocking | encounter | arbitrary abort | $RSet$, $WSet$ | any | no | opaque | no | user abortable |
| BVA | pessimistic | blocking | encounter | abort-free | $ASet$ | heterogeneous | no | opaque | no | correct |
| SVA | pessimistic | blocking | encounter | abort-free | $ASet$, suprema | heterogeneous | no | strict serializable | yes | correct |
| TL2/DTL2 | optimistic | blocking | commit | on conflict | ∅ | variable | no | opaque | no | abortable |
| TFA | optimistic | blocking | commit | on conflict | ∅ | homogeneous | no | opaque | no | abortable |
| MS-PTM | pessimistic | blocking | commit | abort-free | ∅ | variable | no | opaque | no | correct |
| PLE | pessimistic | blocking | encounter | abort-free | ∅ | variable | no | opaque | no | correct |
| SemanticTM | pessimistic | wait-free* | encounter | abort-free | $ASet$, deps. | variable | no | opaque | no | repeatable |
| DATM | optimistic | blocking | commit | on overwriting, deadlock, and cascade | ∅ | variable | yes | conflict serializable | yes | abortable |

# OptSVA

Optimized Supremum Versioning Algorithm

Based on SVA:

- version-based concurrency control
- early release based on *a priori* knowledge
- decentralized, disjoint access parallel

Key ideas:

- discern reads and writes
- delay blocking as much as possible
- expedite release as much as possible

Key mechanisms:

- specific operations delegated to helper threads
- heavy use of buffering
- fine-grained locking

# Basic Versioning

$T_i$ **starts**:

- atomically get the next free version ticket for each object

$T_i$ **executes operation on** $x$:

- wait until $T_i$'s ticket matches $x$'s version counter
- execute operation

$T_i$ **commits**:

- wait until all transactions with lower versions for $x, y, z$ commit
- release each object by incrementing version counter

# Basic Versioning

$T_i$ **starts**:

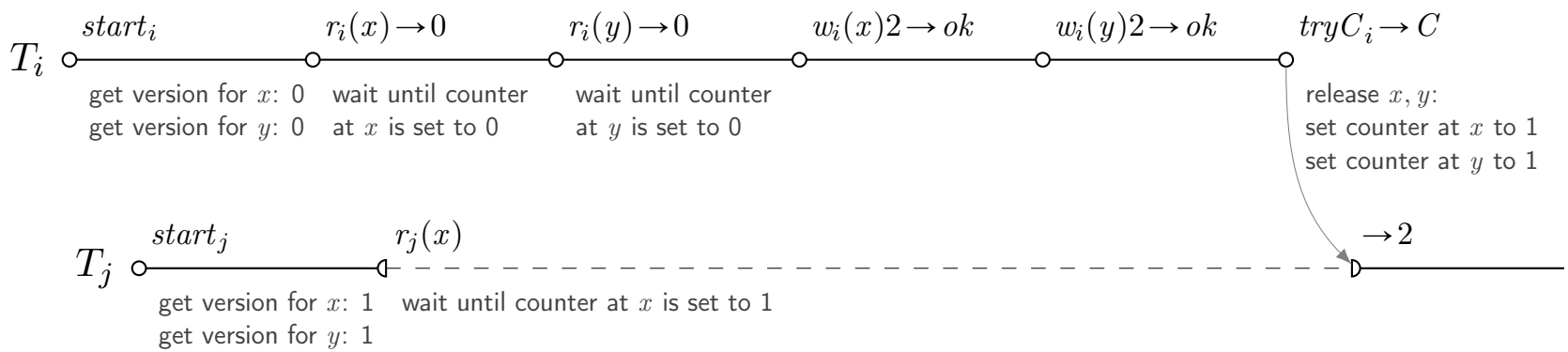
- atomically get the next free version ticket for each object

$T_i$ **executes operation on** $x$:

- wait until $T_i$'s ticket matches $x$'s version counter
- execute operation

$T_i$ **commits**:

- wait until all transactions with lower versions for $x, y, z$ commit
- release each object by incrementing version counter



# Early Release

$T_i$ **starts**:

- atomically get the next unclaimed version ticket for each variable

$T_i$ **executes operation on** $x$:

- wait until $T_i$'s ticket matches $x$'s version counter
- execute operation
- if execution counter reached declared upper bound, release $x$ by incrementing its version counter

$T_i$ **commits**:

- wait until all transactions with lower versions for $x, y, z$ commit
- release each variable by incrementing its version counter (if necessary)

# Early Release

$T_i$ **starts**:

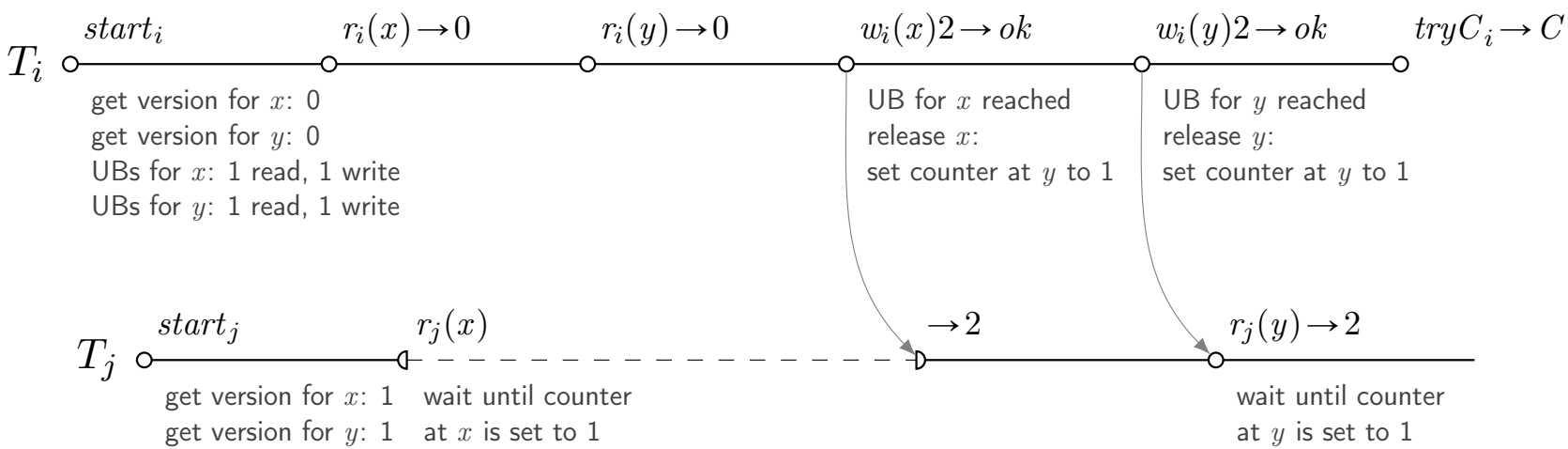
- atomically get the next unclaimed version ticket for each variable

$T_i$ **executes operation on** $x$:

- wait until $T_i$'s ticket matches $x$'s version counter
- execute operation
- if execution counter reached declared upper bound, release $x$ by incrementing its version counter

$T_i$ **commits**:

- wait until all transactions with lower versions for $x, y, z$ commit
- release each variable by incrementing its version counter (if necessary)



# Deriving *a priori* upper bounds

Static analysis:

- value analysis and control flow prediction on control flow graph
- region finding and region analysis to count method calls
- generate upper bound values in source code
- implemented in Java using Soot/Jimple

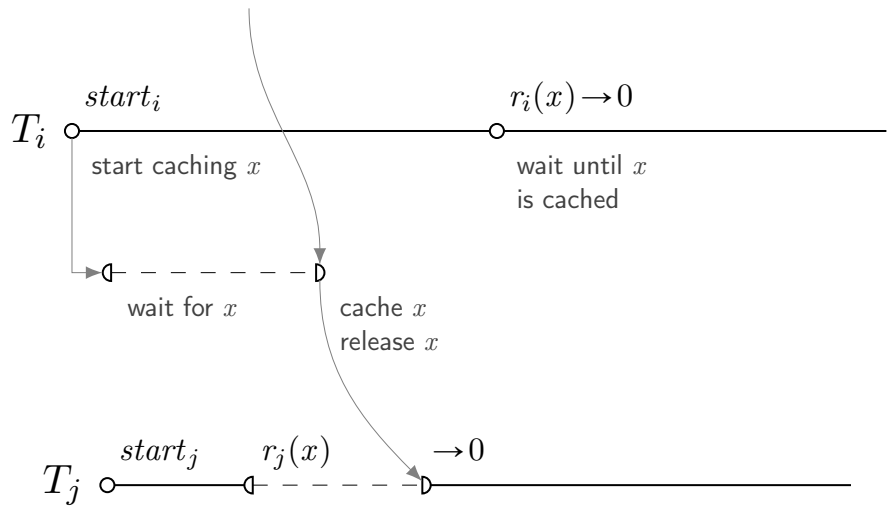
# Read-only Variable Optimization

$T_i$ **starts**:

- (atomically) get the next unclaimed version ticket for each variable
- cache all read-only variables in separate threads:
  - wait until $T_i$'s ticket matches $x$'s version counter
  - make a local copy of $x$
  - release $x$

$T_i$ **executes operation on** $x$ **and** $x$ **is read-only**:

- wait until $x$ finished caching
- execute operation on local copy



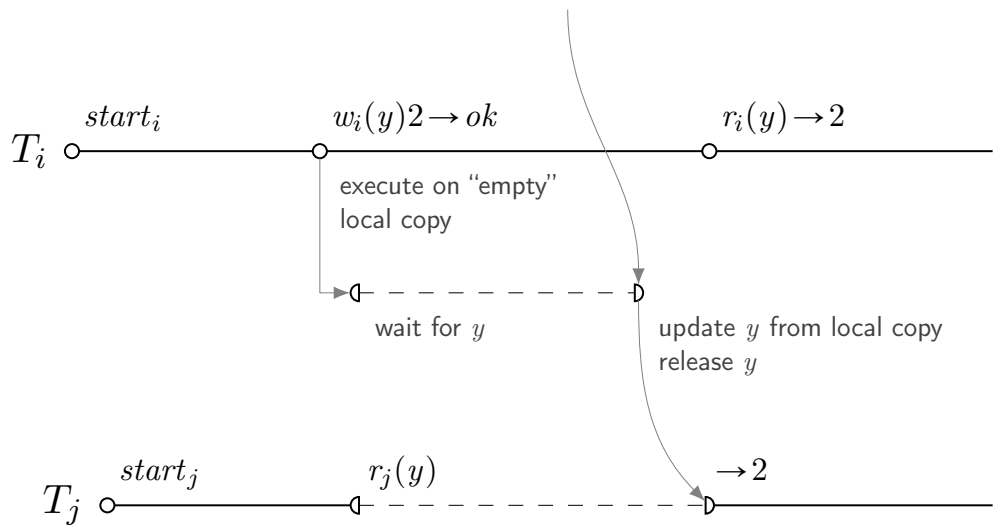
# Initial Write Optimization

**$T_i$ executes first operation on $x$ and operation is a write**

- do **not** wait until $T_i$'s ticket matches $x$'s version counter
- execute write operation on "empty" local copy

**$T_i$ executes last write peration on $x$**

- if local copy is available, in separate thread:
  - wait until $T_i$'s ticket matches $x$'s version counter
  - update $x$ from local copy
  - release $x$
- otherwise: release $x$



# OptSVA+R: programmatic abort capability

$T_i$ **executes operation on** $x$:

- wait until $T_i$'s ticket matches $x$'s version counter
- if any declared variable is invalidated: force abort
- if first operation on $x$: make backup copy
- execute the operation
- if reached declared upper bound for $x$: release $x$

$T_i$ **commits**:

- wait until all transactions with lower versions for $x, y, z$ finish
- if any declared variable is invalidated: force abort
- release each variable (if necessary)

$T_i$ **aborts**:

- wait until all transactions with lower versions for $x, y, z$ finish
- invalidate modified variable and revert them from backup
- release each variable (if necessary)

# OptSVA+R: programmatic abort capability

$T_i$ **executes operation on** $x$:

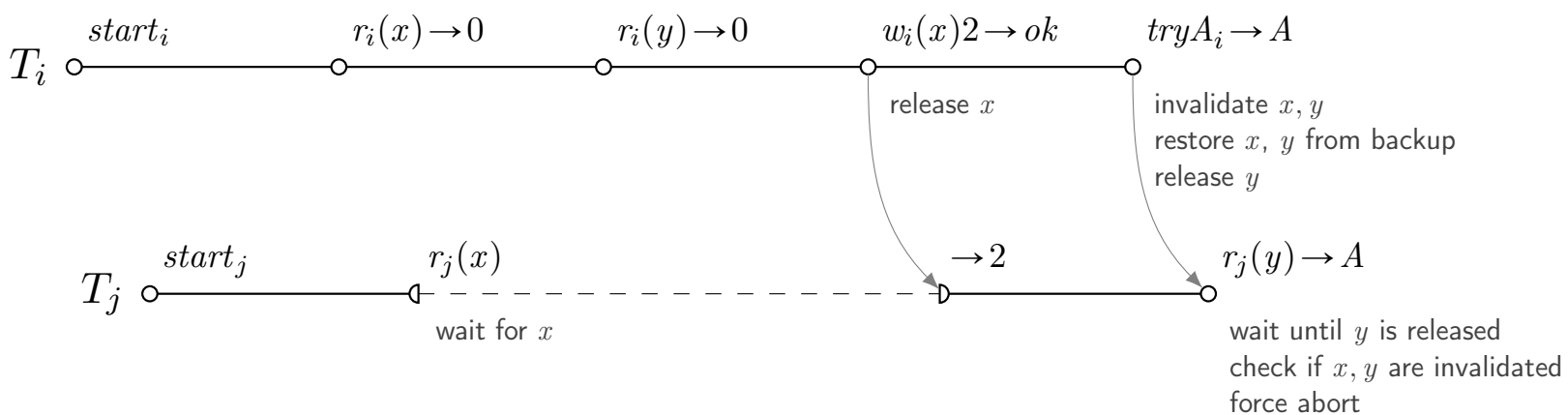
- wait until $T_i$'s ticket matches $x$'s version counter
- if any declared variable is invalidated: force abort
- if first operation on $x$: make backup copy
- execute the operation
- if reached declared upper bound for $x$: release $x$

$T_i$ **commits**:

- wait until all transactions with lower versions for $x, y, z$ finish
- if any declared variable is invalidated: force abort
- release each variable (if necessary)

$T_i$ **aborts**:

- wait until all transactions with lower versions for $x, y, z$ finish
- invalidate modified variable and revert them from backup
- release each variable (if necessary)



# OptSVA Variants

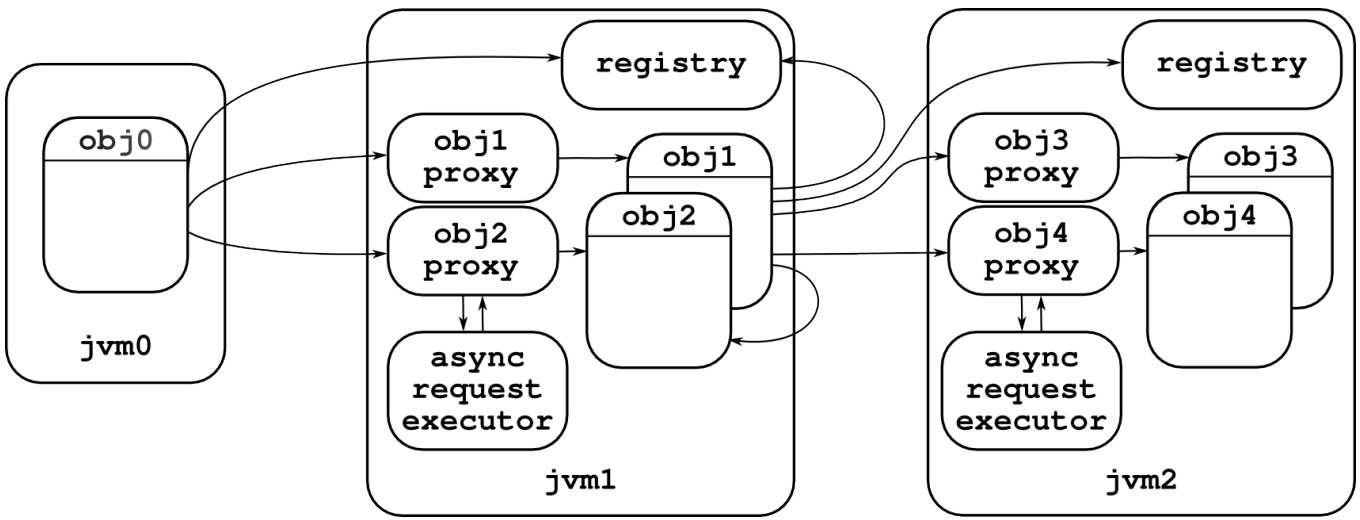
- OptSVA – variables (reads and writes only)
- OptSVA+R – manual abort capability (cascading aborts)
- ROptSVA+R – reluctant (irrevocable) transactions

- OptSVA-CF – arbitrary objects
- OptSVA-CF+R
- ROptSVA-CF+R

- SVA+R
- RSVA+R

# Atomic RMI v2

Implements OptSVA (ROptSVA-CF+R) on top of Java RMI



# Evaluation

Throughput measurement

Frameworks:

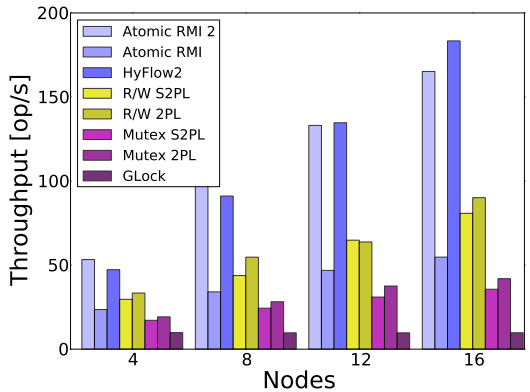
- Atomic RMI (SVA+R)
- Atomic RMI v2(ROptSVA-CF+R)
- Fine grained locking (variants of 2PL):
  - exclusion locks
  - R/W locks
  - single global lock
- HyFlow2 (TFA) – optimistic distributed TM

Benchmark:

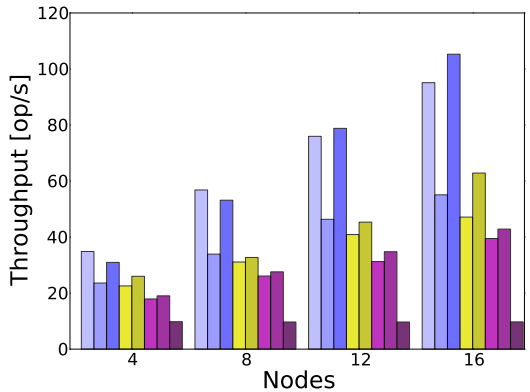
- Distributed version of EigenBench

# Results

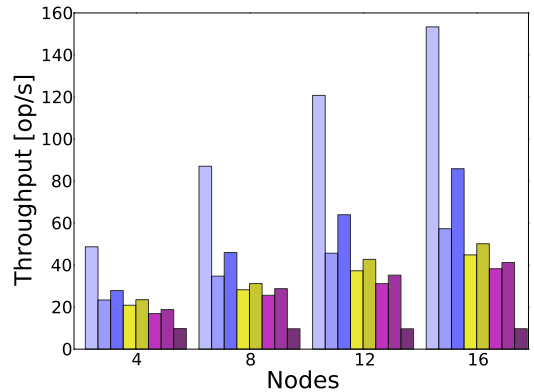
5 objects per node:




80% reads

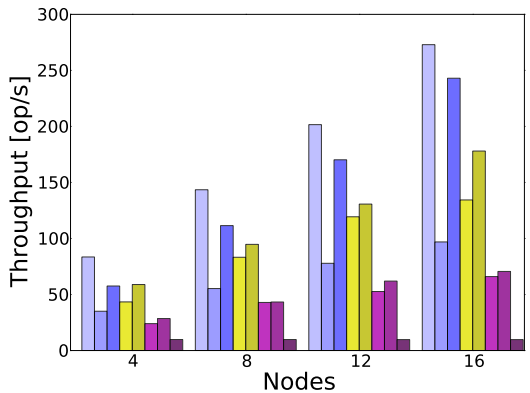



50% reads

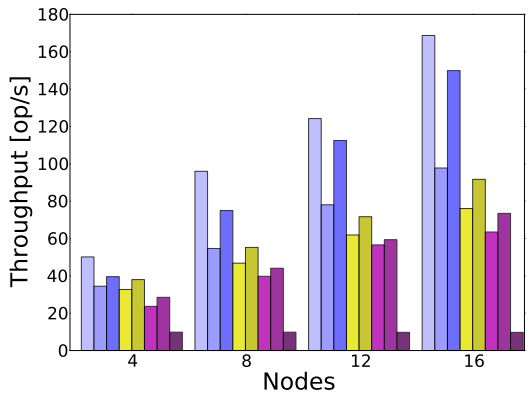



20% reads

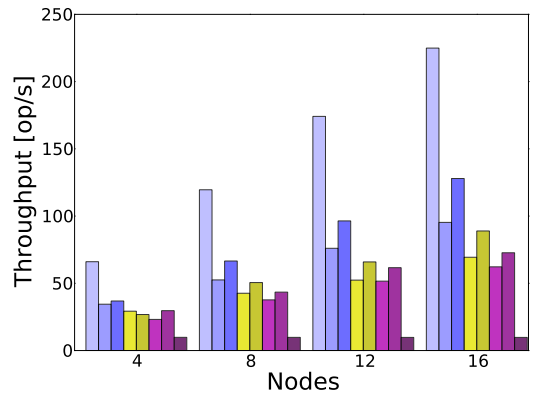
10 objects per node:




80% reads




50% reads




20% reads

# Correctness of Transactional Executions

Safety properties define allowed and prohibited behaviors

Opacity:

- Serializability:
  execution is equivalent to some serial execution
- Real-time order:
  execution preserves the order of transactions
- Consistency:
  transactions only view effects of committed transactions

Guerraoui, Kapałka. *Principles of Transactional Memory*. 2010.

# Correctness of Transactional Executions

Safety properties define allowed and prohibited behaviors

Opacity:

- Serializability:
  execution is equivalent to some serial execution
- Real-time order:
  execution preserves the order of transactions
- Consistency:
  transactions only view effects of committed transactions

Guerraoui, Kapałka. *Principles of Transactional Memory*. 2010.

$\therefore$ Neither SVA nor OptSVA are opaque.

# Correctness of Transactional Executions

Safety properties define allowed and prohibited behaviors

Opacity:

- Serializability:
  execution is equivalent to some serial execution
- Real-time order:
  execution preserves the order of transactions
- Consistency:
  transactions only view effects of committed transactions

Guerraoui, Kapałka. *Principles of Transactional Memory*. 2010.

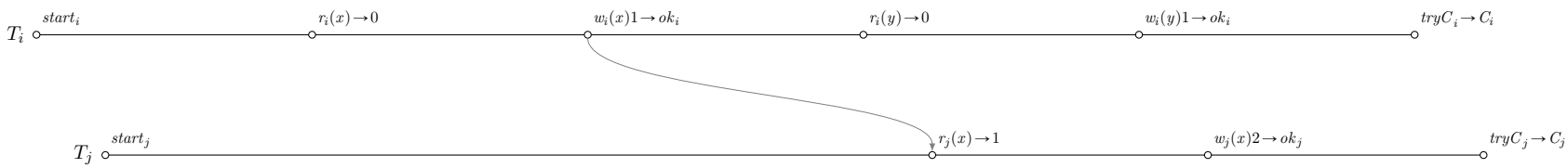
$\therefore$ Neither SVA nor OptSVA are opaque.

# Decomposition

Abort-free TM algorithms produce executions that are “equivalent” to opaque ones, but read from uncommitted transactions.

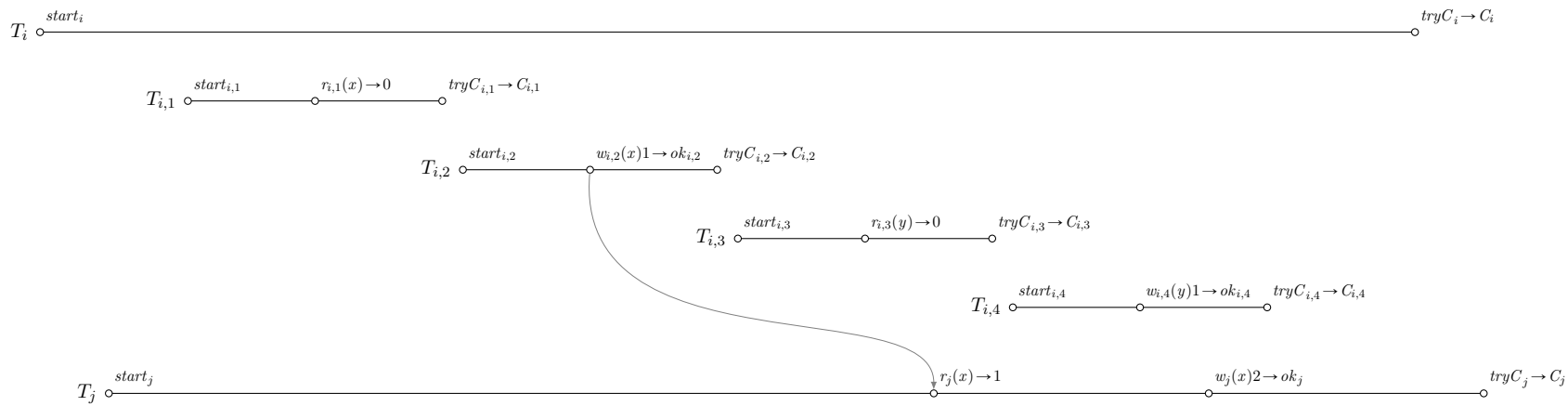
# Decomposition

Abort-free TM algorithms produce executions that are "equivalent" to opaque ones, but read from uncommitted transactions.



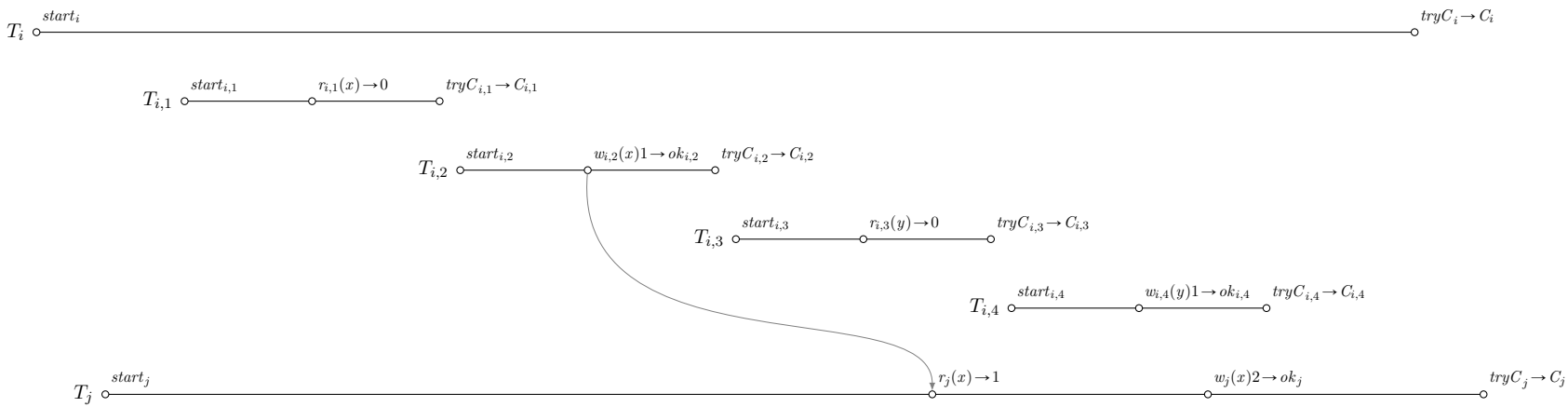
# Decomposition

Abort-free TM algorithms produce executions that are "equivalent" to opaque ones, but read from uncommitted transactions.



# Decomposition

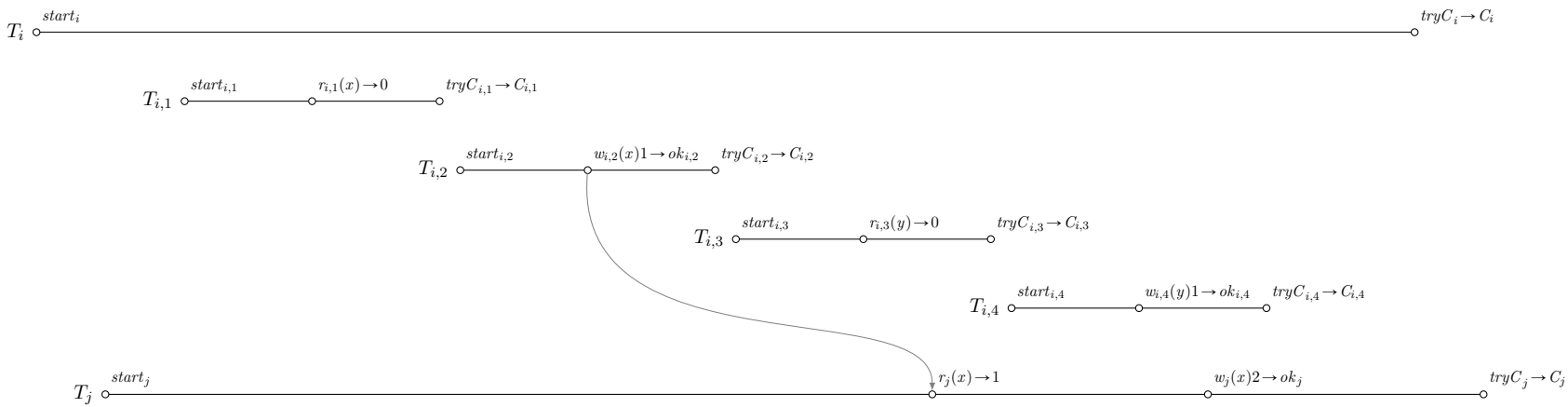
Abort-free TM algorithms produce executions that are "equivalent" to opaque ones, but read from uncommitted transactions.




- Decomposed execution of a non-aborting TM observationally refines the original execution
- Decomposed execution is opaque $\Rightarrow$ original execution observationally refines an opaque execution
- Original execution is opaque by decomposition.

# Decomposition

Abort-free TM algorithms produce executions that are "equivalent" to opaque ones, but read from uncommitted transactions.




- Decomposed execution of a non-aborting TM observationally refines the original execution
- Decomposed execution is opaque $\Rightarrow$ original execution observationally refines an opaque execution
- Original execution is opaque by decomposition.

SVA and OptSVA are opaque by decomposition

# Property Applicability for Early Release

| Property | Application | Early Release Support | Overwriting Support | Aborting Support | ⊆Serializable |
|---|---|---|---|---|---|
| Serializability | database, TM | ✓ | ✓ | ✓ | ✓ |
| CO | database | ✓ | ✓ | ✓ | × |
| Recoverability | database | ✓ | ✓ | ✓ | × |
| Cascadelessness | database | × | × | × | × |
| Strictness | database | × | × | × | ✓ |
| Rigorousness | database | × | × | × | ✓ |
| Opacity | TM | × | × | × | ✓ |
| Markability | TM | × | × | × | ✓ |
| TMS1 | TM | × | × | × | ✓ |
| TMS2 | TM | × | × | × | ✓ |
| VWC | TM | ✓ | × | × | ✓ |
| Live opacity | TM | ✓ | × | × | ✓ |
| Elastic opacity | TM | ✓ | × | × | × |

# Last-use Opacity

Last-use Opacity:

- Serializability:

  execution is equivalent to some serial execution
- Real-time order:

  execution preserves the order of transactions
- Last-use Consistency:

  committed transactions only view effects of committed transactions

  all transactions only view effects of decided transactions

# Last-use Opacity

Last-use Opacity:

- Serializability:

  execution is equivalent to some serial execution
- Real-time order:

  execution preserves the order of transactions
- Last-use Consistency:

  committed transactions only view effects of committed transactions

  all transactions only view effects of decided transactions

Transaction cannot execute more writes on $x \Rightarrow$ decided on $x$

# Last-use Opacity

Last-use Opacity:

- Serializability:

  execution is equivalent to some serial execution
- Real-time order:

  execution preserves the order of transactions
- Last-use Consistency:

  committed transactions only view effects of committed transactions

  all transactions only view effects of decided transactions

Transaction cannot execute more writes on $x \Rightarrow$ decided on $x$

$\therefore$ All SVA+R, OptSVA+R, etc. executions are last-use opaque.

# Last-use Opacity

Last-use Opacity:

- Serializability:

  execution is equivalent to some serial execution
- Real-time order:

  execution preserves the order of transactions
- Last-use Consistency:

  committed transactions only view effects of committed transactions

  all transactions only view effects of decided transactions

Transaction cannot execute more writes on $x \Rightarrow$ decided on $x$

$\therefore$ All SVA+R, OptSVA+R, etc. executions are last-use opaque.

All opaque histories are last-use opaque.

# Strong Last-use Opacity

Strong Last-use Opacity:

- Serializability:

  execution is equivalent to some serial execution
- Real-time order:

  execution preserves the order of transactions
- Strong Last-use Consistency:

  committed transactions only view effects of committed transactions

  all transactions only view effects of strongly decided transactions

Transaction cannot execute more writes on $x$ or abort $\Rightarrow$ strongly decided on $x$

All opaque executions are strong last-use opaque.

# Contributions

- OptSVA and variants for various system models
- Pessimistic distributed TM capable of outperforming state-of-the-art optimistic TM
- Analysis of existing safety conditions vs early release
- Safety properties for TM with early release

# Journal Publications

K. Siek and P. T. Wojciechowski. Proving opacity of transactional memory with early release. Foundations of Computing and Decision Sciences, Volume 40, Issue 4. December 2015.

K. Siek and P. T. Wojciechowski. Atomic RMI: A distributed transactional memory framework. International Journal of Parallel Programming, Volume 44, Issue 3. June 2015.

K. Siek, P. T. Wojciechowski. A Formal Design of a Tool for Static Analysis of Upper Bounds on Object Calls in Java. In Proceedings of FMICS 2012: the 17th International Workshop on Formal Methods for Industrial Critical Systems (co-located with FM 2012). Lecture Notes in Computer Science, pages 192–206. August 2012.

A. Wojciechowski and K. Siek. Barcode scanning from mobile-phone camera photos delivered via MMS: Case study. In Advances in Conceptual Modeling—Challenges and Opportunities. Lecture Notes in Computer Science, pages 218–227. October 2008.

# In Preparation

K. Siek and P. T. Wojciechowski. Atomic RMI 2: Highly parallel pessimistic distributed transactional memory. ArXiv:1606.03928 [cs.DC].

J. Baranowski, P. Kobyliński, K. Siek, and P. T. Wojciechowski. Helenos: A realistic benchmark for distributed transactional memory. ArXiv:1603.07899 [cs.DC].

K. Siek and P. T. Wojciechowski. Last-use opacity: A strong safety property for transactional memory with early release support. ArXiv:1506.06275 [cs.DC].

P. T. Wojciechowski and K. Siek. The optimal pessimistic transactional memory algorithm. ArXiv:1605.01361 [cs.DC].

K. Siek and P. T. Wojciechowski. Transactions scheduled while you wait.

# Conference and Workshop Papers


P. T. Wojciechowski and K. Siek. *Atomic RMI 2: Distributed Transactions for Java.* In proceedings of AGERE'16: the 6th Workshop on Programming based on Actors, Agents, and Decentralized Control (in conjunction with SPLASH '16), October 2016.

K. Siek, P. T. Wojciechowski. *Atomic RMI: a Distributed Transactional Memory Framework*. In proceedings of HLPP 2014: the 7th International Symposium on High-level Parallel Programming and Applications. July 2014.

K. Siek, P. T. Wojciechowski. *Brief Announcement: Relaxing Opacity in Pessimistic Transactional Memory*. In Proceedings of DISC'14: the 28th International Symposium on Distributed Computing. October 2014.

K. Siek, P. T. Wojciechowski. *Zen and the Art of Concurrency Control: An Exploration of TM Safety Property Space with Early Release in Mind*. In Proceedings of WTTM'14: the 6th Workshop on the Theory of Transactional Memory. July 2014.

P. T. Wojciechowski, K. Siek. *Having Your Cake and Eating it Too: Combining Strong and Eventual Consistency.* In Proceedings of PaPEC 2014: the 1st Workshop on the Principles and Practice of Eventual Consistency. April 2014.

K. Siek, P. T. Wojciechowski. *Towards a Fully-Articulated Pessimistic Distributed Transactional Memory (Brief announcement).* In Proceedings of SPAA 2013: the 25th ACM Symposium on Parallelism in Algorithms and Architectures. July 2013.

P. T. Wojciechowski, K. Siek. *Transaction Concurrency Control via Dynamic Scheduling Based on Static Analysis (Extended Abstract).* In Proceedings of WTM 2012: Euro-TM Workshop on Transactional Memory (co-located with ACM SIGOPS EuroSys 2012). April 2012.

K. Siek, P. T. Wojciechowski. *Statically Computing Upper Bounds on Object Calls for Pessimistic Concurrency Control (Extended Abstract).* In Proceedings of $EC^2$ 2010: Workshop on Exploiting Concurrency Efficiently and Correctly (co-located with CAV 2010). July 2010.