Atomic RMI 2: Distributed Transactions for Java

Paweł T. Wojciechowski and Konrad Siek

Poznań University of Technology

{pawel.t.wojciechowski,konrad.siek}@cs.put.edu.pl

30 X 2016

Distributed Systems Group

http://dsg.cs.put.poznan.pl

Transactional memory

Concurrency control is notoriously difficult:

- interaction between unrelated threads
- additional structural code
- deadlocks, livelocks, priority inversion

```
synchronized{aLock} {
    synchronized{bLock} {
        a = b;
    }
    b = b + 1;
}
```

Transactional memory

Concurrency control is notoriously difficult:

- interaction between unrelated threads
- additional structural code
- deadlocks, livelocks, priority inversion

```
synchronized{aLock} {
    synchronized{bLock} {
        a = b;
    }
    b = b + 1;
}
```

```
transaction.start();
a = b;
b = b + 1;
transaction.commit();
```

Transactional memory (TM):

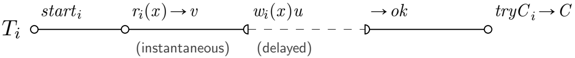
- ease of use on top
- efficient concurrency control under the hood

Transaction abstraction

Transaction:

$$
\begin{array}{lcl}
T_i & = & [\ op_1,\ op_2,\ ....,\ op_n\ ] \\
op_1 & = & start_i \\
op_i & = & r_i(x) \rightarrow v \mid w_i(x)v \rightarrow ok \mid ... \\
op_n & = & tryC_i \rightarrow C \mid tryC_i \rightarrow A \mid tryA_i \rightarrow A \mid \\
 & & r_i(x) \rightarrow A \mid w_i(x)v \rightarrow A \mid ...
\end{array}
$$

Execution:

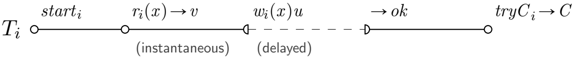


Transaction abstraction

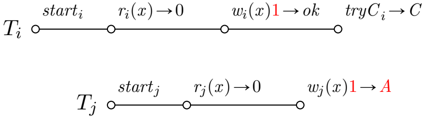
Transaction:

$$
\begin{array}{rcl}
T_i & = & [\ op_1,\ op_2,\ ....,\ op_n\ ] \\
op_1 & = & start_i \\
op_i & = & r_i(x) \rightarrow v \mid w_i(x)v \rightarrow ok \mid ... \\
op_n & = & tryC_i \rightarrow C \mid tryC_i \rightarrow A \mid tryA_i \rightarrow A \mid \\
 & & r_i(x) \rightarrow A \mid w_i(x)v \rightarrow A \mid ...
\end{array}
$$

Execution:



Conflict resolution (optimistic TM, increment of x):

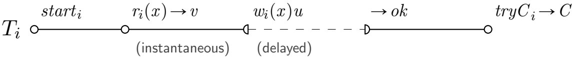


Transaction abstraction

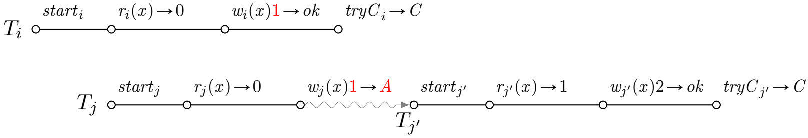
Transaction:

$$
\begin{aligned}
T_i &= [\ op_1,\ op_2,\ ....,\ op_n\ ] \\
op_1 &= start_i \\
op_i &= r_i(x) \rightarrow v \mid w_i(x)v \rightarrow ok \mid ... \\
op_n &= tryC_i \rightarrow C \mid tryC_i \rightarrow A \mid tryA_i \rightarrow A \mid \\
&\quad\ r_i(x) \rightarrow A \mid w_i(x)v \rightarrow A \mid ...
\end{aligned}
$$

Execution:



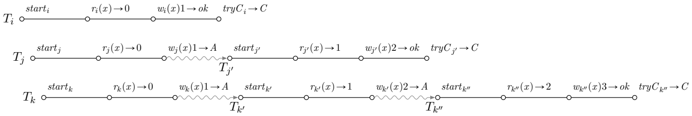
Conflict resolution (optimistic TM, increment of x):



Problems with optimistic TM

Optimistic TM relies on aborts:

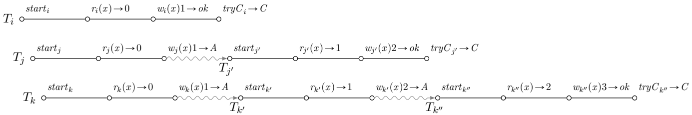
- low performance in high contention



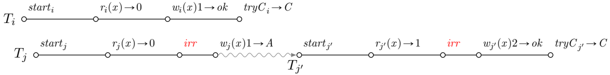
Problems with optimistic TM

Optimistic TM relies on aborts:

- low performance in high contention



- problems with irrevocable operations:
  - do not operate on shared data
  - have visible side effects
  - effects cannot be withdrawn (must be compensated)
  - examples: network communication, locks, system calls, I/O



Pessimistic TM

Optimistic TM:

- run simultaneously in case there are no conflicts
- abort and retry if there are conflicts

Pessimistic TM

~~Optimistic TM:~~ Pessimistic TM:

- run simultaneously in case there are no conflicts
- abort and retry if there are conflicts

Pessimistic TM

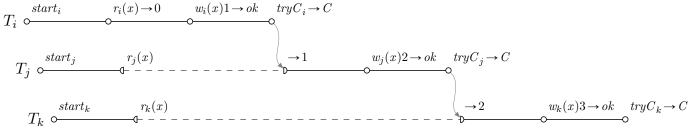
~~Optimistic TM:~~ Pessimistic TM:

- defer execution to prevent conflict
- abort and retry if there are conflicts

Pessimistic TM

~~Optimistic TM:~~ Pessimistic TM:

- defer execution to prevent conflict
- avoid (most) forced aborts



- less waste of CPU (more waiting)
- performs better in high contention
- easy handling of irrevocable operations

Atomic RMI 2

A Java framework implementing distributed pessimistic TM

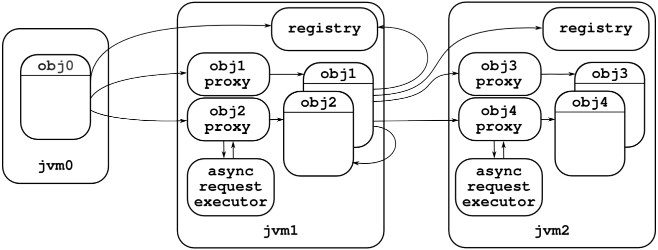
Implements the Optimized Supremum Versioning Algorithm

- completely distributed
- early release
- irrevocable operations
- rollback support
- fault tolerance

(completely distributed, early release, irrevocable operations, rollback support } OptSVA)

Backend: Java RMI

Atomic RMI 2 architecture



Remote object definition

```
interface Resource extends Remote {
    @Access(Mode.READ)
    int get() throws RemoteException;

    @Access(Mode.WRITE)
    void set(int value) throws RemoteException;

    @Access(Mode.ANY)
    void increment() throws RemoteException;
}

class ResourceImpl implements Resource extends TransactionalUnicastRemoteObject {
    private int value = 0;

    void set(int value) {
        this.value = value;
    }
    int get() {
        return this.value;
    }
    void increment() {
        this.value += 1;
    }
}

class Server {
    public static void main(String[] args) throws Exception {
        Registry registry = LocateRegistry.createRegistry(9001);
        registry.bind("x", new ResourceImpl());
        registry.bind("y", new ResourceImpl());
    }
}
```

Transaction example

```
Registry registry = LocateRegistry.getRegistry(9001);

Transaction transaction = new Transaction();

Resource x = transaction.accesses(registry.lookup("x"));
Resource y = transaction.accesses(registry.lookup("y"));

transaction.start();

int xv = x.get();
int yv = y.get();
x.set(xv + 2);
y.set(yv + 2);

transaction.commit();
```

Transaction example (Transactional)

```java
Registry registry = LocateRegistry.getRegistry(9001);

Transaction transaction = new Transaction();

Resource x = transaction.accesses(registry.lookup("x"));
Resource y = transaction.accesses(registry.lookup("y"));

transaction.start(
    new Transactional() {
        void atomic (Transaction transaction) {
            int xv = x.get();
            int yv = y.get();
            x.set(xv + 2);
            y.set(yv + 2);
        }
    }
);
```

OptSVA: basic versioning

T_i **starts**:

- atomically get the next free version ticket for each object

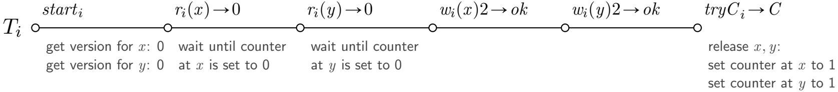
T_i **executes a method on** x:

- wait until T_i's ticket matches x's version counter
- execute the method

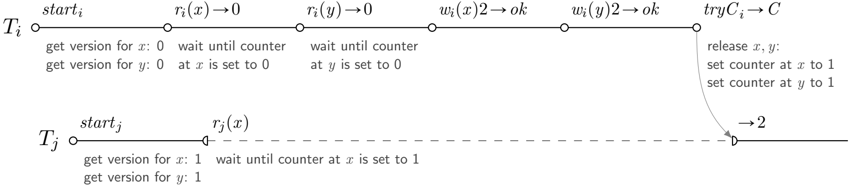
T_i **commits**:

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

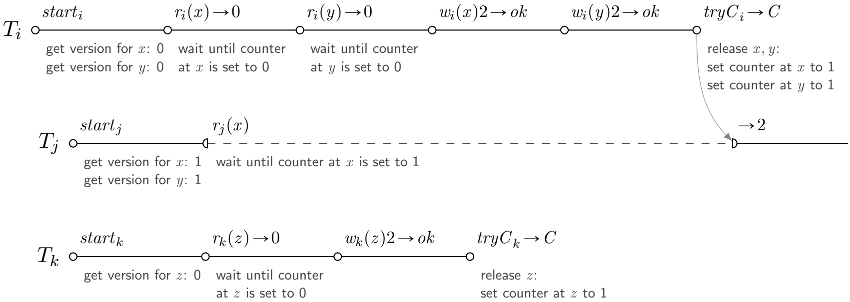
Transaction execution: basic versioning



Transaction execution: basic versioning



Transaction execution: basic versioning



Transaction example: upper bounds

```
Transaction transaction = new Transaction();

Resource x = transaction.accesses(registry.lookup("x"), 2);
Resource y = transaction.accesses(registry.lookup("y"), 2);

transaction.start();

int xv = x.get();
int yv = y.get();
x.set(xv + 2);
y.set(yv + 2);

transaction.commit();
```

OptSVA: early release

T_i **starts**:

- atomically get the next unclaimed version ticket for each object

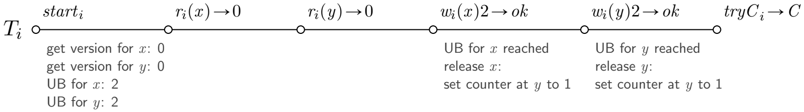
T_i **executes a method on** x:

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

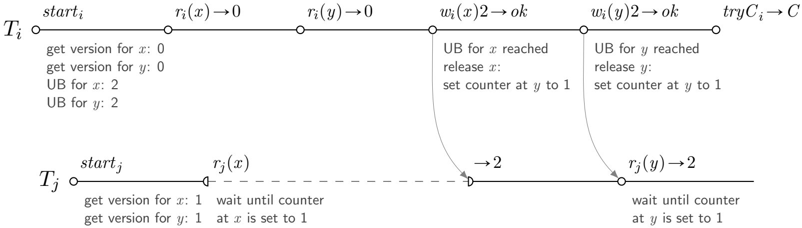
T_i **commits**:

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

Transaction execution: early release



Transaction execution: early release



Deriving upper bounds

Upper bounds can be derived by static analysis (precompiler)

Supplemented by manual early release

Transaction example: manual early release

```
Transaction transaction = new Transaction();

Resource x = transaction.accesses(registry.lookup("x"));
Resource y = transaction.accesses(registry.lookup("y"));

transaction.start();

int xv = x.get();
int yv = y.get();

if (xv < 10)
    x.set(xv + 2);
else
    transaction.release(x);

y.set(yv + 2);

transaction.commit();
```

Transaction example: manual abort

```
Transaction transaction = new Transaction();

Resource x = transaction.accesses(registry.lookup("x"), 2);
Resource y = transaction.accesses(registry.lookup("y"), 2);

transaction.start();

int xv = x.get();
int yv = y.get();

if (xv < 10)
    x.set(xv + 2);
else
    transaction.abort();

y.set(yv + 2);

transaction.commit();
```

OptSVA: abort support

T_i **executes a method on** x:

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

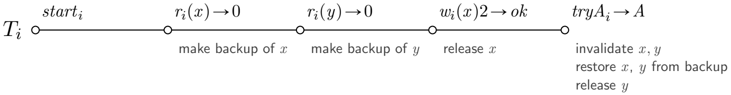
T_i **commits**:

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

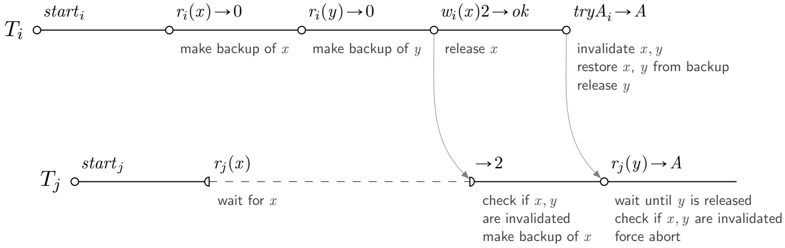
T_i **aborts**:

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

Transaction execution: abort



Transaction execution: cascading abort



Transaction example: prevent cascading aborts

```
Transaction transaction = new Transaction(true); // reluctant transaction

Resource x = transaction.accesses(registry.lookup("x"), 2);
Resource y = transaction.accesses(registry.lookup("y"), 2);

...
```

OptSVA: reluctant transactions

Reluctant T_i executes a method on x:

- wait until all transactions with lower versions for x finish
- if any declared object is invalidated: force abort
- if first operation on x: make backup copy
- execute the method
- if reached declared upper bound for x: release x

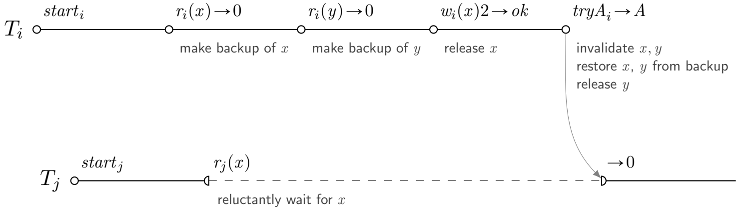
T_i **commits**:

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

T_i **aborts**:

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

Transaction execution: prevented cascading aborts



Example: a transaction treating objects as read-only

```java
Transaction transaction = new Transaction();

Resource x = transaction.reads(registry.lookup("x"), 1);
Resource y = transaction.accesses(registry.lookup("y"));

transaction.start();

int xv = x.get();
y.set(xv + 2);
System.out.println("new value: " + y.get());

transaction.commit();
```

OptSVA: a transaction treating objects as read-only

T_i **starts**:

- (atomically) get the next unclaimed version ticket for each object
- cache all read-only objects in parallel
- once object x is cached, release x

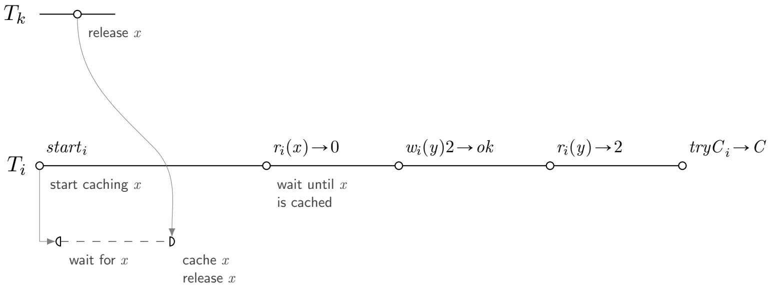
T_i **executes a read method on read-only object** x:

- wait until x object finished caching
- if any declared object is invalidated: force abort
- if first operation on x: make backup copy
- execute the method

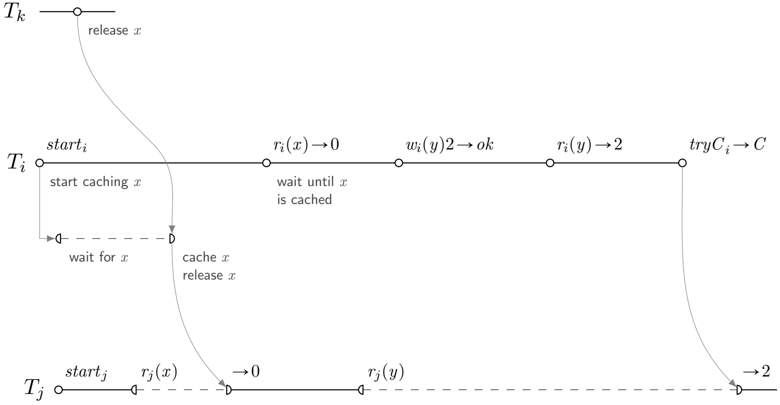
T_i **commits**:

- wait until all transactions with lower versions for x, y, z commit
- if any declared object is invalidated: force abort
- increment version counter for each object (if necessary)

Transaction execution: read-only objects



Transaction execution: read-only objects



Transaction example: write optimizations

```
Transaction transaction = new Transaction();

Resource x = transaction.reads(registry.lookup("x"), 1);
Resource y = transaction.accesses(registry.lookup("y"),
                                  1 /*write*/, 1 /*read*/);

transaction.start();

int xv = x.get();
y.set(xv + 2);
System.out.println("new value: " + y.get());

transaction.commit();
```

OptSVA: first write

T_i **executes a write method on** x:

- if first operation of any kind on x: create log
- execute the method on log (if available)

T_i **executes other methods on** x:

- wait until T_i's ticket matches x's version counter
- if log for x has operations: apply log to x and discard the log
- execute the method

T_i **commits**:

- wait until all transactions with lower versions for x, y, z commit
- if any declared object is invalidated: force abort
- apply log to x (if necessary)
- increment version counter for each object (if necessary)

OptSVA: first write, last write

T_i **executes a write method on** x:

- if first operation of any kind on x: create log
- execute the method on log (if available)
- if last write on x:

  if log is empty: release x

  otherwise: wait for x, apply log, cache x, release x (in parallel)

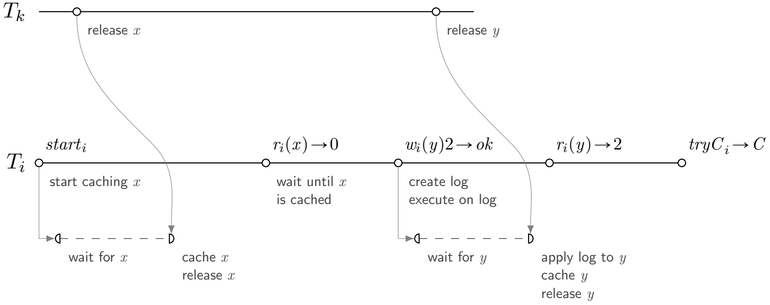
T_i **executes other methods on** x:

- wait until T_i's ticket matches x's version counter
- if log for x has operations: apply log to x and discard the log
- execute the method

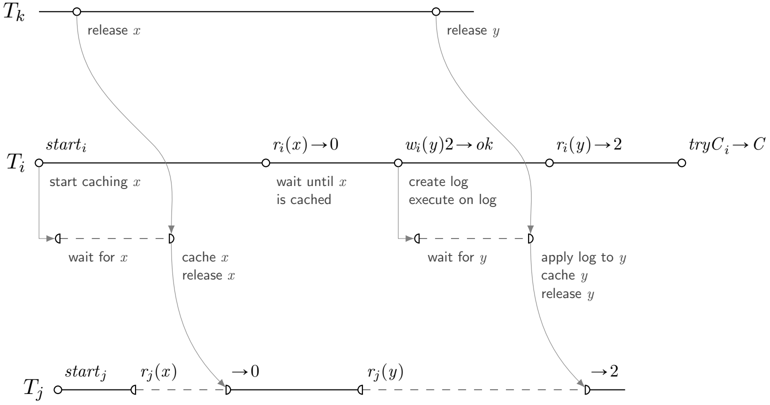
T_i **commits**:

- wait until all transactions with lower versions for x, y, z commit
- if any declared object is invalidated: force abort
- apply log to x (if necessary)
- increment version counter for each object (if necessary)

Transaction execution: write operations



Transaction execution: write operations



Transactions for Actors?

Actors: a_1, a_2, ...

Transaction:

$$
\begin{array}{lcl}
T_i & = & [\ op_1,\ op_2,\ ....,\ op_n\ ] \\
op_1 & = & start_i \\
op_i & = & send(a_j)[r_i(x)] \rightarrow ok \mid recv[p] \rightarrow v \mid \\
 & & send(a_j)[w_i(x)v] \rightarrow ok \mid ... \\
op_n & = & tryC_i \rightarrow C \mid tryC_i \rightarrow A \mid tryA_i \rightarrow A \mid \\
 & & send(a_j)[r_i(x)] \rightarrow A \mid recv[p] \rightarrow A \mid \\
 & & send(a_j)[w_i(x)v] \rightarrow A \mid ...
\end{array}
$$

Transactions for Actors?

Actors: a_1, a_2, ...

Transaction:

$$
\begin{array}{lcl}
T_i & = & [\ op_1,\ op_2,\ ....,\ op_n\ ] \\
op_1 & = & start_i \\
op_i & = & send(a_j)[r_i(x)] \rightarrow ok \mid recv[p] \rightarrow v \mid \\
 & & send(a_j)[w_i(x)v] \rightarrow ok \mid ... \\
op_n & = & tryC_i \rightarrow C \mid tryC_i \rightarrow A \mid tryA_i \rightarrow A \mid \\
 & & send(a_j)[r_i(x)] \rightarrow A \mid recv[p] \rightarrow A \mid \\
 & & send(a_j)[w_i(x)v] \rightarrow A \mid ...
\end{array}
$$

Pros and cons:

- allow for consistent behavior on multiple nodes
- introduce dependeces between asynchronous messages

?

TM safety property primer

Serializability:

The outcome of all committed transactions is equivalent to the outcome of some serial execution of these transactions

Real-time order:

Transactions executing one after another cannot be re-arranged to justify their correctness

Opacity:

- Serializability and real-time order
- Transactions only view the effects of committed transactions

Last-use opacity:

- Serializability and real-time order
- Committed transactions only view the effects of committed transactions, but
- Committed and uncommitted transactions only view the effects of the final modifications in transactions

Atomic RMI 2 (OptSVA) properties

- Serializable and real-time order
- If transactions don't invoke manual aborts:
  - opaque from programmers' point of view
  - irrevocable operations always correct
- Otherwise:
  - last-use opaque
  - irrevocable operations in reluctant transactions always correct

Evaluation

Frameworks:

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

Environment:

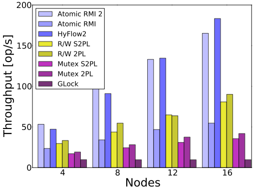
- 10 × 2 × quad-core Intel Xeon L3260 (2.83 GHz), 4 GB RAM
- OpenSUSE 13.1
- JRE (64 bit): Oracle 1.8.0_05-b13, Hotspot 25.5-b02

Benchmark:

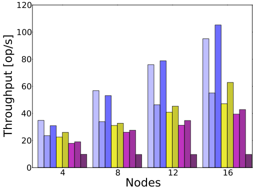
- Distributed version of EigenBench

Throughput

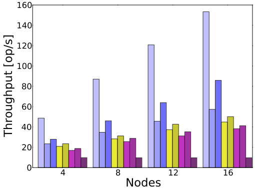
Short transactions, 5 objects per node:



80% reads

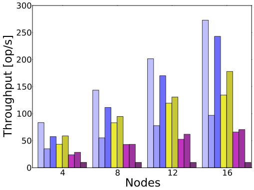


50% reads

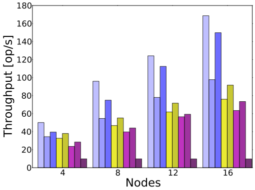


20% reads

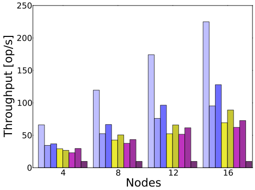
Short transactions, 10 objects per node:



80% reads



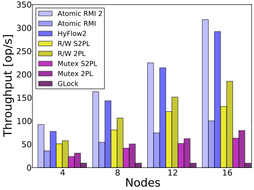
50% reads



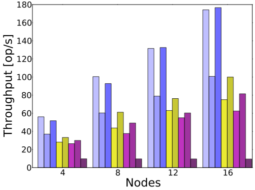
20% reads

Throughput

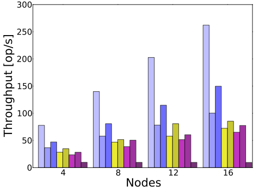
Long transactions, 10 objects per node



80% reads

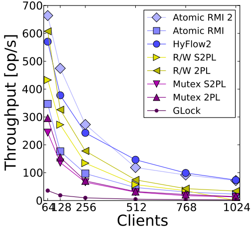


50% reads

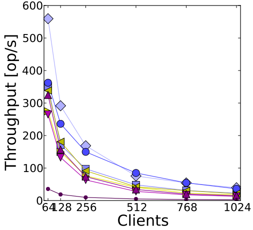


20% reads

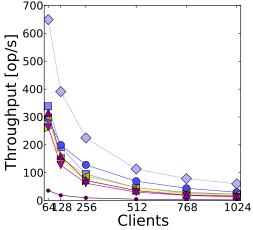
Scalability



80% reads



50% reads



20% reads

Manual early release vs UB

```
Transaction transaction = new Transaction();

Resource x = transaction.accesses(registry.lookup("x"));
Resource y = transaction.accesses(registry.lookup("y"));

transaction.start();

for (i = 0; i < n; i++) {
    x.increment();
    y.increment();
}
transaction.release(x);
transaction.release(y);
// local operations

transaction.commit();
```

Manual early release vs UB

```
Transaction transaction = new Transaction();

Resource x = transaction.accesses(registry.lookup("x"), n);
Resource y = transaction.accesses(registry.lookup("y"), n);

transaction.start();

for (i = 0; i < n; i++) {
    x.increment(); // x released before calling y
    y.increment();
}
// local operations

transaction.commit();
```

Manual early release vs UB

```
Transaction transaction = new Transaction();

Resource[] resources = new Resource[n];
resources[0] = transaction.accesses(registry.lookup("r1"), 2);
resources[1] = transaction.accesses(registry.lookup("r2"), 2);
// ...
resources[n] = transaction.accesses(registry.lookup("rn"), 2);

transaction.start();

for (i = 0; i < n; i++) {
    if (resources[i].get() == 0) {
        resources[i].set(1);
        break;
    } else
        transaction.release(resources[i]); // released with no delay
}

transaction.commit();
```