Transaction Concurrency Control via Dynamic Scheduling Based on Static Analysis

Paweł T. Wojciechowski, Konrad Siek
Poznań University of Technology

Bern, 10 April 2012

Motivations

Transactional Memory (TM) traditionally employs optimistic concurrency control + contention management.

Advantages of fully pessimistic approaches were recently recognized (Shavit, Matveev WTTM '11): I/O and system calls, debugging, ...

A lot fewer results are about *distributed TMs*.

Our work

- **optimistic** locally scoped transactions + replicated objects => Paxos STM (Euro-TM WDTM '12),
- **pessimistic** distributed transactions on remote objects => Atomic RMI (this talk).

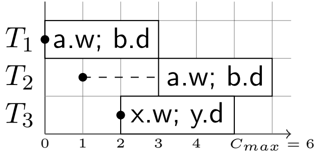
Example: "withdraw-deposit" transaction

Below is a program in Atomic RMI, where a and b are remote objects (in the sense of Java RMI):

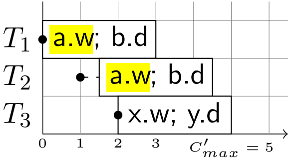
```
Transaction t = new Transaction(registry);

t.start();
int balance = a.getBalance();
if (balance < sum) {
    t.rollback();
} else {
    a.withdraw(sum);
    b.deposit(sum);
    t.commit();
}
```

Pessimistic concurrency by versioning[1]

Basic Versioning Algorithm:



Supremum Versioning Algorithm:

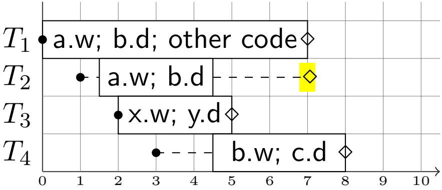


Static (a priori) scheduling: T_2 is temporarily blocked by T_1.
T_3 is **not** blocked since it accesses other objects.

Both schedules are serializable but $C'_{max} < C_{max}$.

[1] Wojciechowski *et al.* IPDPS '04, PPDP '05

Transaction commit and rollback (SVA)



Transaction thread is blocked till <u>transaction commit</u> ($\diamond$).

T_1's <u>rollback</u> causes T_2 and T_4 to rollback =>
T_2's commit must be deferred.
T_2's rollback must cause T_4 to rollback.
T_3 is not affected since it accesses other objects.

Static analysis

When to release an object held by a transaction?

We use 3 methods:

1. The programmer releases an object (simple but not safe!).
2. **Supremum analysis**: Release an object if # of object calls reached an upper bound (supremum S).
3. **Last-use analysis**: If S cannot be inferred, release an object if no more calls can be made by a client (work-in-progress).

We developed a precompiler that instruments Java code after analyzing its translation to Jimple (more suitable than Java bytecode).

Example code after precompilation round

Below is our example program, modified by the precompiler:

```
Transaction t = new Transaction(registry);

a = t.accesses(a, 2); // upper bound = 2
b = t.accesses(b, 1); // upper bound = 1

t.start();
int balance = a.getBalance();
if (balance < sum) {
    t.rollback();
} else {
    a.withdraw(sum);
    b.deposit(sum);
    t.commit();
}
```

Supremum vs. last-use analysis

The object last-use analysis applied:

```
transaction.start();
for (i=0; i < n; i++) {
  a.m();
  b.m();  // a unnecessary blocked in the last loop
}
::release a, b;
transaction.commit();
```

The object call supremum analysis applied:

```
transaction.start();
for (i=0; i < n; i++) {
  a.m(); // a will be released in nth iteration
  b.m(); // b will be released in nth iteration
}
transaction.commit();
```

Object release based on last-use analysis

Clients on nodes i ($i = 2..n$):

```
transaction.start();
a.m();
::release a;
transaction.commit();
```

A method of a remote object a on node 1:

```
m() {
   ::while (object blocked) {};
   ...
   return res;
}
```

Release of object a by every client requires a network message.

Object release based on supremum analysis

Clients on nodes i ($i = 2..n$):

```
transaction.start();
a.m();
transaction.commit();
```

A method of a remote object a on node 1:

```
m() {
  ::while (ticket(i) − obj_count > supremum(i)) {};
  ...
  ::obj_count ← obj_count + 1
  return res;
}
```

Release of object a is a local operation (no network overhead).

Dynamic transaction scheduling - algorithm

1. On transaction start-up, request a globally unique ticket (required by static scheduling of object calls).
2. Sort the queue of ticket requests, either:
   - long transactions first, or
   - short transactions first.
3. Use a simple heuristic: the transaction length is proportional to statically inferred # of object calls.

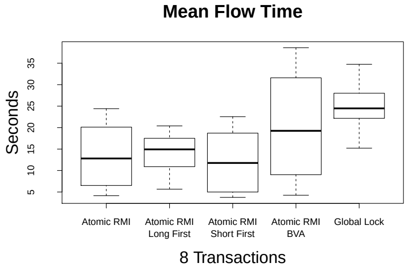
Micro-benchmark

Designed to check if experimental results match a theoretical schedule (a larger benchmark is an ongoing work):

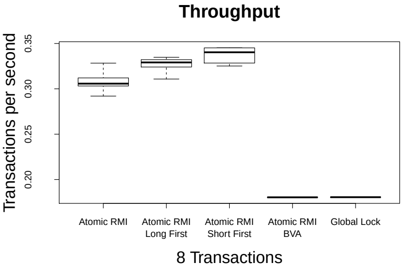
- four remote objects `a`, `b`, `c`, and `d`,
- each object has one method "sleep 1 second and return",
- 4-24 concurrent transactions spawned on 8 network nodes at the same time:
  - 50% **short transactions** (3 sec.): calls of `a`, `b`, and `c`,
  - 50% **long transactions** (6 sec.): calls of `c`, `d`, `c`, `d`, `a`, and `b`.

Evaluation environment:

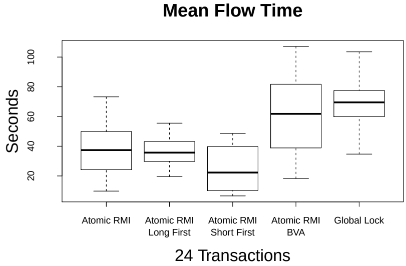
- cluster of nodes: quad-core Xeon X3230 CPU with 4 GB RAM,
- 1Gb network.



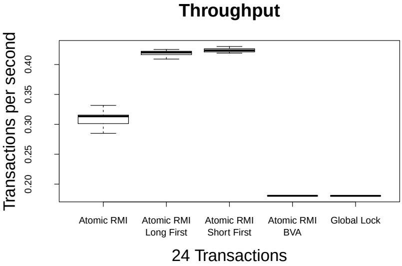
Mean Flow Time
Seconds
5
10
15
20
25
30
35
Atomic RMI
Atomic RMI Long First
Atomic RMI Short First
Atomic RMI BVA
Global Lock
8 Transactions



Throughput
Transactions per second
0.20
0.25
0.30
0.35
Atomic RMI
Atomic RMI Long First
Atomic RMI Short First
Atomic RMI BVA
Global Lock
8 Transactions



Mean Flow Time
Seconds
20
40
60
80
100
Atomic RMI
Atomic RMI Long First
Atomic RMI Short First
Atomic RMI BVA
Global Lock
24 Transactions



Throughput
Transactions per second
0.20
0.25
0.30
0.35
0.40
Atomic RMI
Atomic RMI Long First
Atomic RMI Short First
Atomic RMI BVA
Global Lock
24 Transactions

Conclusions and future work

- Pessimistic concurrency by versioning outperforms global lock.
- Static analysis can make this approach efficient and safe.
- Dynamic scheduling of transactions can further improve it for almost null cost at runtime (done in background).
- Upper bounds on calls are used to estimate transaction length.

Future work:

- Problems of distributed deadlock and partial faults.

Project page:

- `http://www.it-soa.eu/atomicrmi`