On the COST of concurrency In Transactional Memory

Petr Kuznetsov

TU Berlin/DT-Labs

Joint work with Srivatsan Ravi

Euro-TM, 2011

STM is about ease-of-programming
and efficient use of concurrency

Does it come with a cost?

- Ease-of-programming: transactions with all-or-nothing semantics
  - ✓Opacity: total order on transactions, including aborted ones, every read returns the last committed value

- Concurrency: running multiple transactions in parallel
  - ✓When a transaction must commit?

Progress conditions: when to commit?

- Single-lock: if no overlap with another transaction
  - ✓Zero concurrency
- Progressiveness: if no overlap with a conflicting transaction
  - ✓Some concurrency
- Permissiveness: whenever possible (without affecting correctness)
  - ✓Maximal (?) concurrency

How to measure the cost of concurrency?

- The number of expensive synchronization patterns [Attiya et al., POPL 2011] :
  - ✓Read-after-write (RAW)
  - ✓Atomic-write-after-read (AWAR)

Our results

- In every permissive STMs, a transaction performs $\Omega(k)$ RAW/AWAR patterns for a read set of size k
- There exist progressive STMs that incur at most one RAW or AWAR per transaction

Read-after-Write: imposing order in relaxed memory models

In the code

```
  ...
write(X,1)
read(Y)
  ...
```

In almost all modern architectures

```
...
read(Y)
...
write(X,1)
...
```

Read-after write reordering

Enforcing the order

- *read-after-write (RAW) fence*

```
    ...
write(X,1)
 fence()  // enforce the order
read(Y)
    ...
```

Enforcing the order

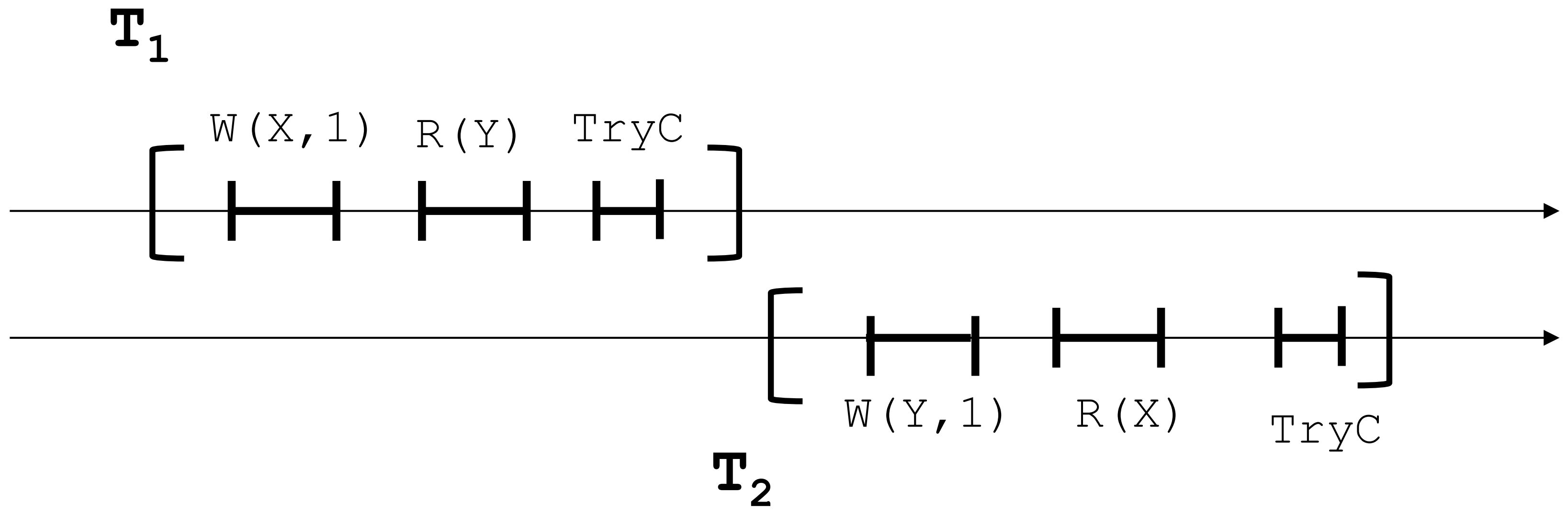
- *atomic-write-after-read (AWAR)*
  - ✓E.g., CAS, TAS, fetch&add,…

```
atomic{
    read(Y)
    ...
    write(X,1)
}
```

Why care about RAW/AWARs?

- One RAW fences/AWAR takes ~50 RMRs!
- But [Attiya et al., POPL 2011] any implementation that exports *strongly non-commutative* methods must use RAW/AWARs
  - ✓ Queues, counters, sets,…
  - ✓ Mutual exclusion
- … and transactions in STMs

Non-commutative transactions



T_1 influences T_2 and T_2 influences T_1 =>

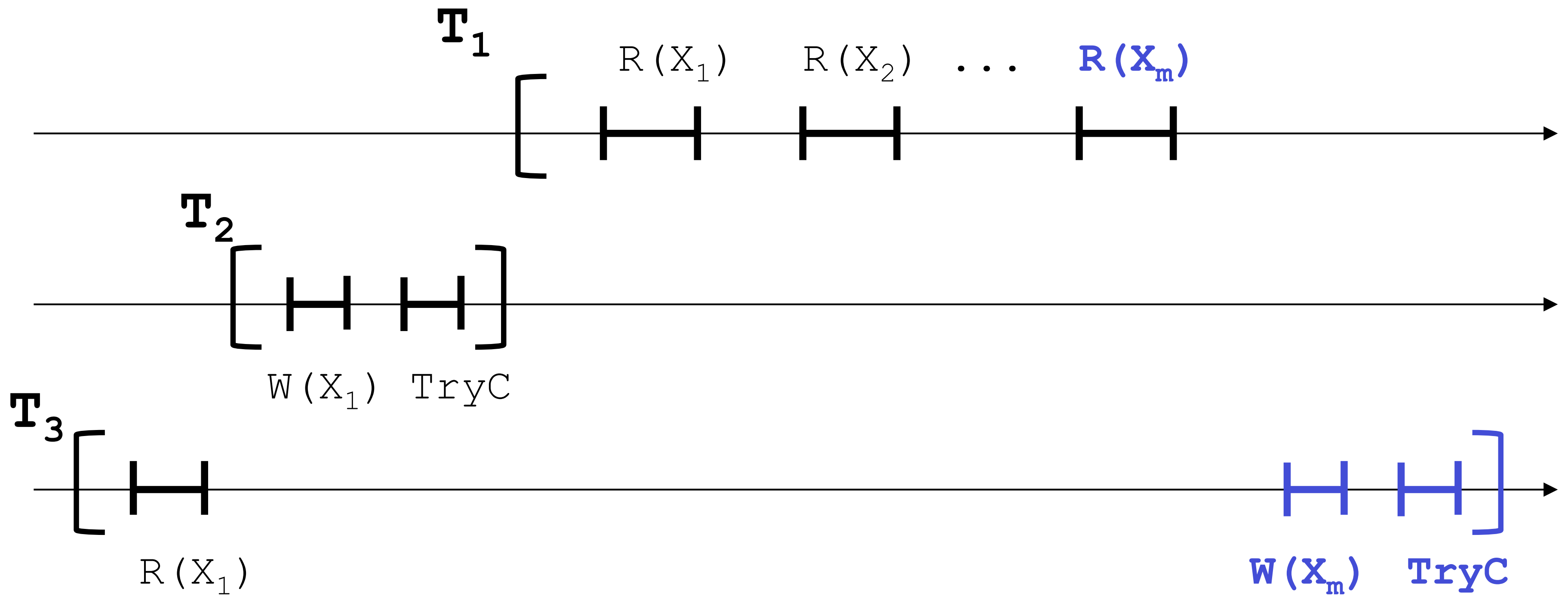
- T_1 must write to some base object x
- … and then read from some y≠x
- (or AWAR is performed)

At least one RAW/AWAR is used in T_1

In single-lock STMs, every reading and updating committed transaction performs at least one RAW/AWAR

Extends to progressive and permissive STMs

Permissive STMs



[R(X_m)] and [W(X_m);TryC] are strongly non-commutative

=> T_3 enforces R(X_m) to perform a RAW/AWAR

=> T_1 performs Ω(m) RAW/AWARs

In permissive STMs, a transaction performs at least one RAW/AWAR per read

What about weaker progress conditions?

What about progressiveness?

Constant (multi) RAW/AWAR implementations:

1. Multi-trylocks: acquire a lock on a set of objects using one multi-RAW
2. mCAS: atomically verify the read-set and update the write set using at most one AWAR

No RAW/AWAR in read-only or aborted transactions

Protected data

- Intuition: at some moment, every transaction must protect all objects in its write set
  - ✓E.g., by acquiring locks or using time-outs

- In every progressive disjoint-access-parallel STM, each transaction has to protect $\Omega(k)$ objects for a write set of size k

Summary

Trade-offs between the degree of concurrency and the cost of implementation

- Linear RAW/AWAR complexity for permissive STMs
- Constant RAW/AWAR complexity for progressive STMs (optimal)
  - ✓Extends to strong progressiveness
- But even a progressive STM requires to protect a linear number of objects (e.g., acquire a linear number of locks)

Future challenges

- Exploring the space of progress conditions
  - ✓Obstruction-freedom, …
- Relaxing opacity
  - ✓View transactions, elastic transactions, virtual world consistency, snapshot isolation, etc.
- Refining the notion of protected data

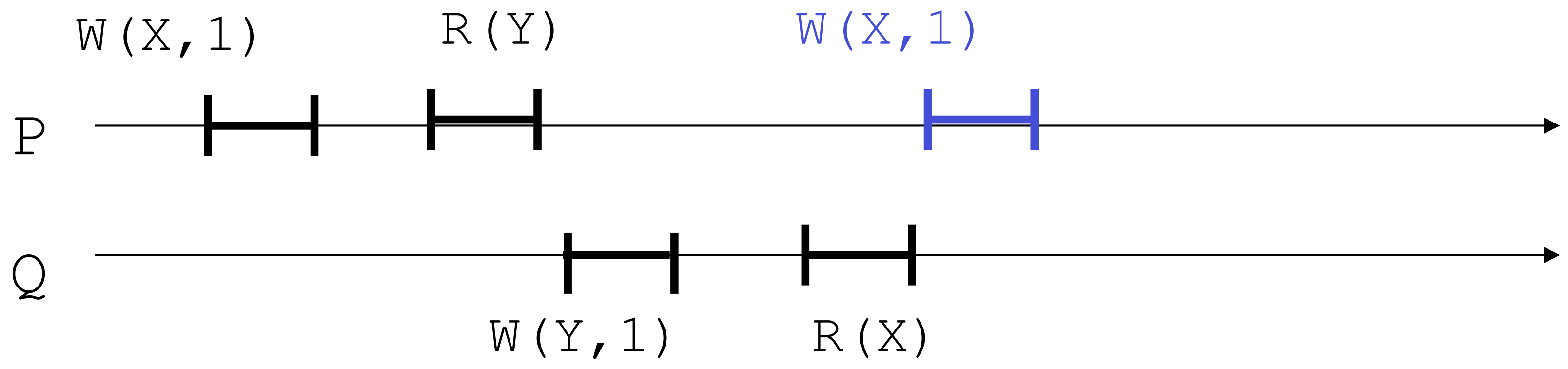
More in TR, CoRR, http://arxiv.org/abs/1103.1302, 2011

THANK YOU!

QUESTIONS?

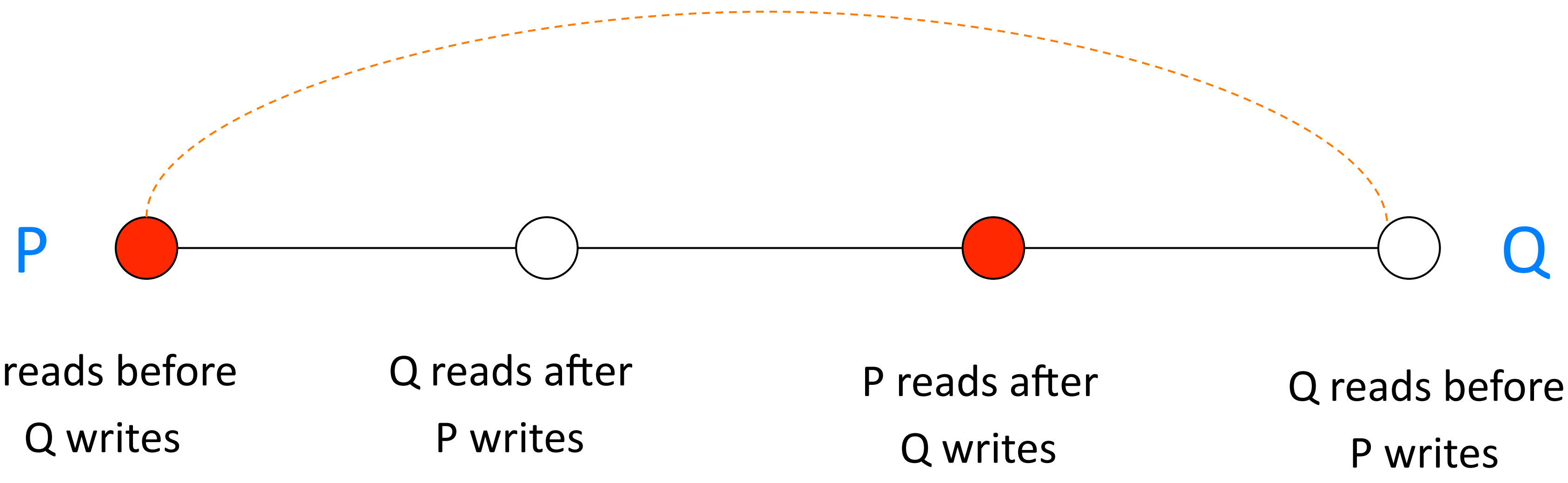
What’s wrong with reordering?

```
Process P:              Process Q:
...                     ...
write(X,1)              write(Y,1)
read(Y)                 read(X)
...                     ...
```



Possible outcomes

Out-of-order: both think they run solo



Read-after-Write: imposing order in relaxed memory models

```
 ...
write A
read B
 ...
```

In almost all modern architectures

```
read B
write A
```

Read-after write reordering

```
 ...
write A
fence
read B
 ...
```

RAW fence: enforce order