Accurate and Efficient Runtime Detection of Atomicity Errors in Concurrent Programs

Liqiang Wang

Joint work with Scott D. Stoller

STONY BROOK
STATE UNIVERSITY OF NEW YORK

A Typical Concurrency Error: Atomicity Violation

Duplicate vector v1

```
lock(v1);
allocate space for v2 according to the size of v1;
unlock(v1);

lock(v1);
copy elements from v1 to v2;
unlock(v1);
```

```
lock(v1);
Remove all elements in v1;
unlock(v1);
```

An Atomicity Violation in Sun JDK 1.4.2 and 1.5.0

```
public class Vector extends ... implements ... {
    int elementCount;
    Object[] elementData;
    public Vector(Vector v1) {
        elementCount = v1.size();
        elementData = new Object[elementCount];
        v1.toArray(elementData);
    }
    public synchronized int size() { return elementCount; }
    public synchronized Object[] toArray(Object a[]) { ... }
    public synchronized void removeAllElements() { ... }
}
```

Another thread may execute v1.removeAllElements()

One thread executes: Vector v2= new Vector(v1);

Outcome: v2 may be full of null elements (behavior of toArray with wrong size argument). No exception is thrown. No data race.

First shown by Wang and Stoller 04

Atomicity and Serializability

- **Serializability of database systems**
  - A general requirement for all transactions.
  - The DBMS enforces a standard concurrency control (synchronization) policy on all transactions.
  - Example: 2-phase locking.

- **Atomicity of concurrent programs**
  - A common but not universal requirement for code blocks.
  - A program may use multiple synchronization policies.
  - Synchronization primitives are scattered throughout the program.

Outline

- Introduction to atomicity
- **Conflict-atomicity and view-atomicity**
- Runtime analysis
- Commit-node algorithm
- Experiments and conclusions

Conflict-Equivalence and View-Equivalence

As for serializability in database, we define two equivalences and then two notions of atomicity.

- Two executions are **conflict-equivalent** if
  - Every two conflicting events (read and write to the same variable, or write and write to the same variable) appear in the same order.
- Two executions are **view-equivalent** if
  - Each read has the same predecessor write. (The predecessor write is assumed to be the initialization if it does not exist)
  - The final write to each shared variable is the same.

Conflict Atomicity and View Atomicity

- **Transaction**: an execution of a code block expected to be atomic.

- A set of transactions is **conflict-atomic** **(view-atomic)** if every concurrent execution of the program is **conflict-equivalent** **(view-equivalent)** to a serial execution (i.e., the transactions are executed without interruption by other threads).

Is {t1, t2} Atomic?

- **Example 1**:

  t1= W(x) W(x), t2 = W(x).

  All feasible executions:

  E1: W(x) W(x) W(x) serial

  E2: W(x) W(x) W(x) serial

  E3: W(x) W(x) W(x) not serial

Conflict-atomicity: E3 is not conflict-equivalent to E1.

Is {t1, t2} Atomic?

- **Example 1**:

  t1= W(x) W(x), t2 = W(x).

  All feasible executions:

  E1: W(x) W(x) W(x) serial

  E2: W(x) W(x) W(x) serial

  E3: W(x) W(x) W(x) not serial

Conflict-atomicity: E3 is not conflict-equivalent to E1 and E2,

{t1, t2} is not conflict-atomic.

Is {t1, t2} Atomic?

- **Example 1**:

  t1= W(x) W(x), t2 = W(x).

  All feasible executions:

  E1: W(x) W(x) W(x) serial

  E2: W(x) W(x) W(x) serial

  E3: W(x) W(x) W(x) not serial

View-atomicity: E3 is not view-equivalent to E1.

Is {t1, t2} Atomic?

- **Example 1**:

  t1= W(x) W(x), t2 = W(x).

  All feasible executions:

  E1: W(x) W(x) W(x) serial

  E2: W(x) W(x) W(x) serial

  E3: W(x) W(x) W(x) not serial

View-atomicity: Although E3 is not view-equivalent to E1, E3 is view-equivalent to E2, {t1, t2} is view-atomic.

Conflict-atomicity: {t1, t2} is not conflict-atomic.

Is {t1, t2} Atomic?

- **Example 2**:

  t1= R(x) W(x), t2 = W(x).

  All feasible executions:

  E1: R(x) W(x) W(x) serial

  E2: W(x) R(x) W(x) serial

  E3: R(x) W(x) W(x) not serial

Conflict-atomicity: E3 is not conflict-equivalent to E1.

Is {t1, t2} Atomic?

- **Example 2**:

  t1= R(x) W(x), t2 = W(x).

  All feasible executions:

  E1: R(x) W(x) W(x) serial

  E2: W(x) R(x) W(x) serial

  E3: R(x) W(x) W(x) not serial

Conflict-atomicity: E3 is not conflict-equivalent to E1 and E2, {t1, t2} is not conflict-atomic.

Is {t1, t2} Atomic?

- **Example 2**:

  t1= R(x) W(x), t2 = W(x).

  All feasible executions:

  E1: R(x) W(x) W(x) serial

  E2: W(x) R(x) W(x) serial

  E3: R(x) W(x) W(x) not serial

View-atomicity: E3 is not view-equivalent to E1.

Is {t1, t2} Atomic?

- **Example 2**:

  t1= R(x) W(x), t2 = W(x).

  All feasible executions:

  E1: R(x) W(x) W(x) serial

  E2: W(x) R(x) W(x) serial

  E3: R(x) W(x) W(x) not serial

View-atomicity: E3 is not view-equivalent to E1 and E2, {t1, t2} is not view-atomic.

Conflict-atomicity: {t1, t2} is not conflict-atomic.

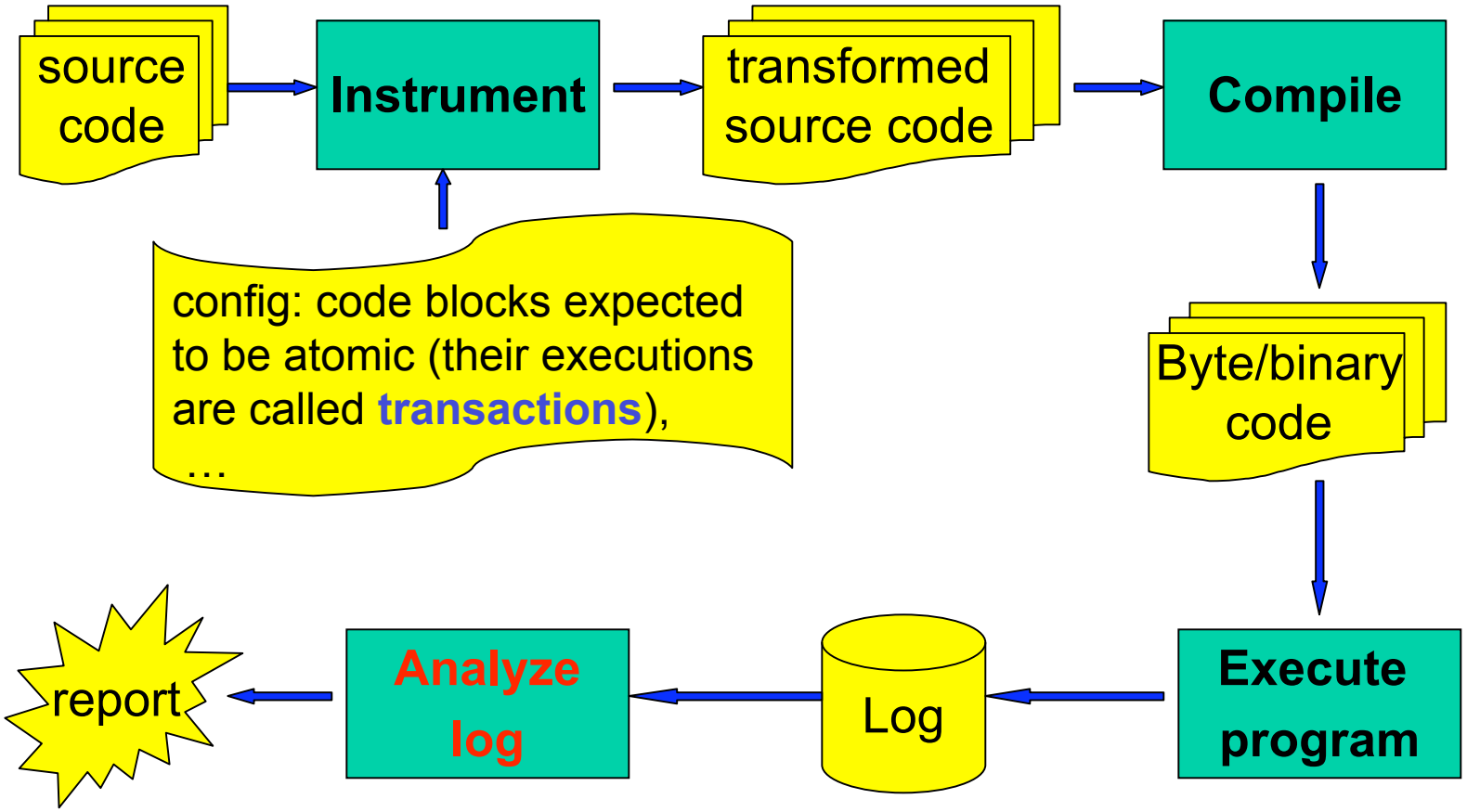
Polynomial Equivalence of Conflict-Atomicity and View Atomicity

- **Theorem**: The problems of checking conflict-atomicity and checking view-atomicity are **polynomially** reducible to each other.
  - This is surprising because
    - checking conflict serializability is in P.
    - checking view serializability is NP-complete.
  - Proof sketch: define polytime functions f and g, such that
    - An execution E is conflict-atomic iff f(E) is view-atomic.
    - An execution E is view-atomic iff g(E) is conflict atomic.
- **Future work**: is checking conflict/view atomicity NP-complete?

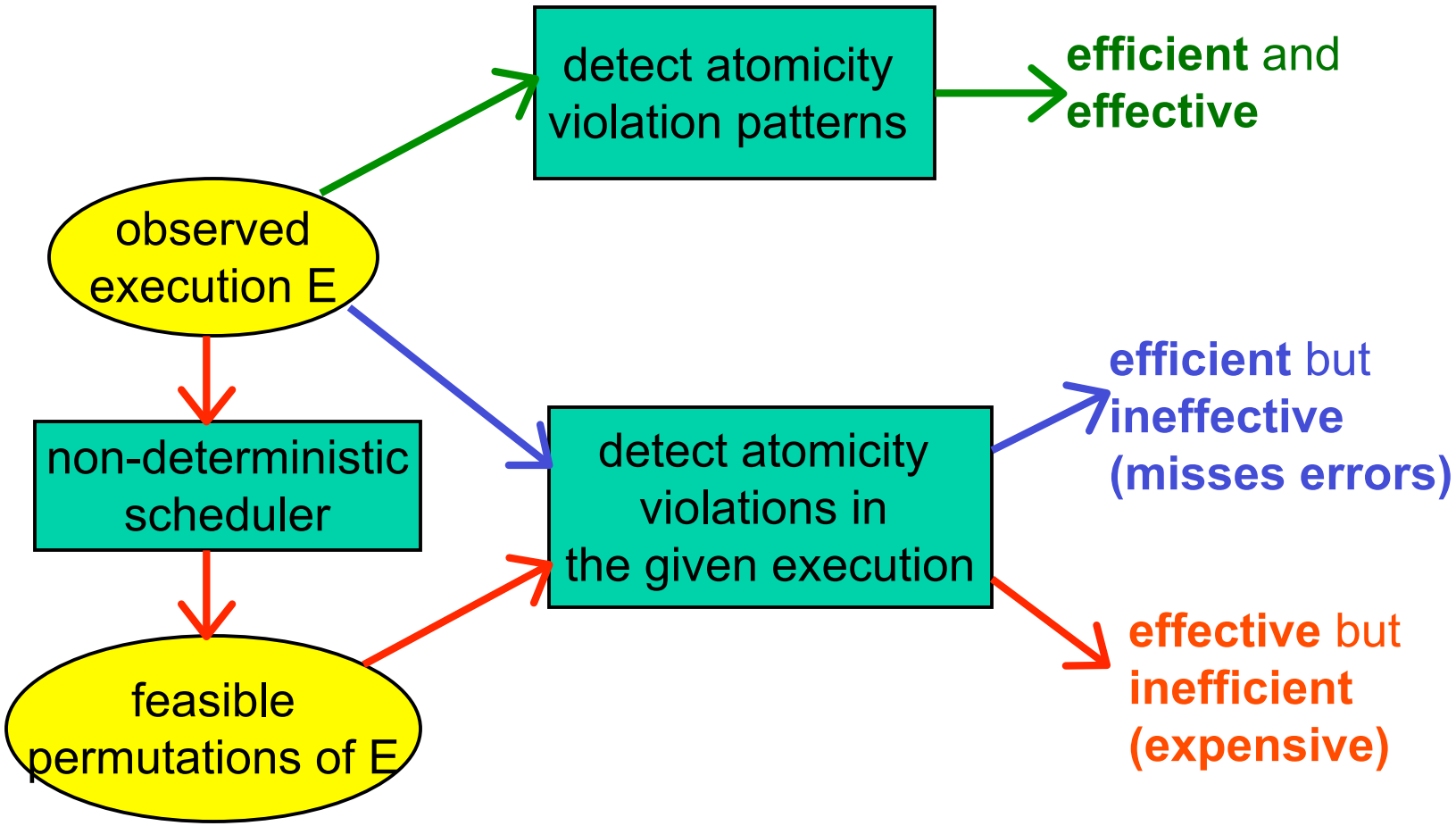
Outline

- Introduction to atomicity
- Conflict-atomicity and view-atomicity
- **Runtime analysis**
- Commit-node algorithm
- Experiments and conclusions

Runtime Analysis of Atomicity



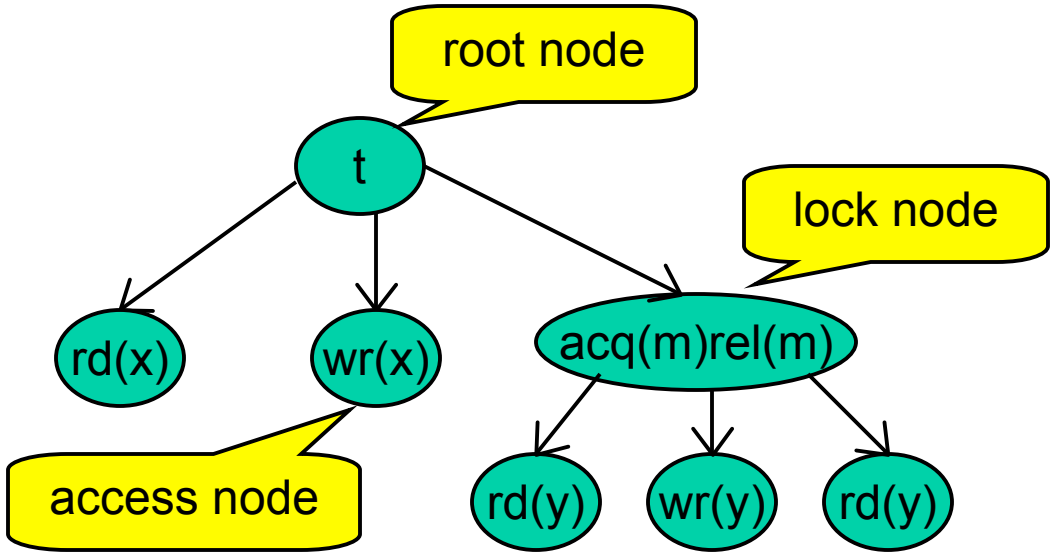
How to Analyze?



Outline

- Introduction to atomicity
- Conflict-atomicity and view-atomicity
- Runtime analysis
- **Commit-node algorithm**
- Experiments and conclusions

Step 1: Construct Access Tree



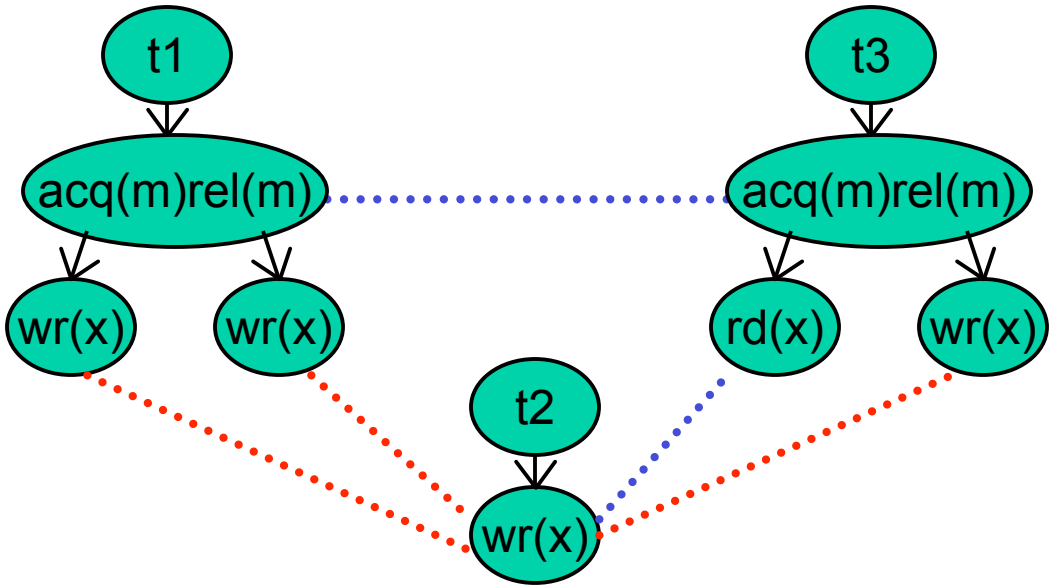
Access tree for transaction t.

```
rd(x)
wr(x)
acq(m)
    rd(y)
    wr(y)
    rd(y)
rel(m)
```

Transaction t

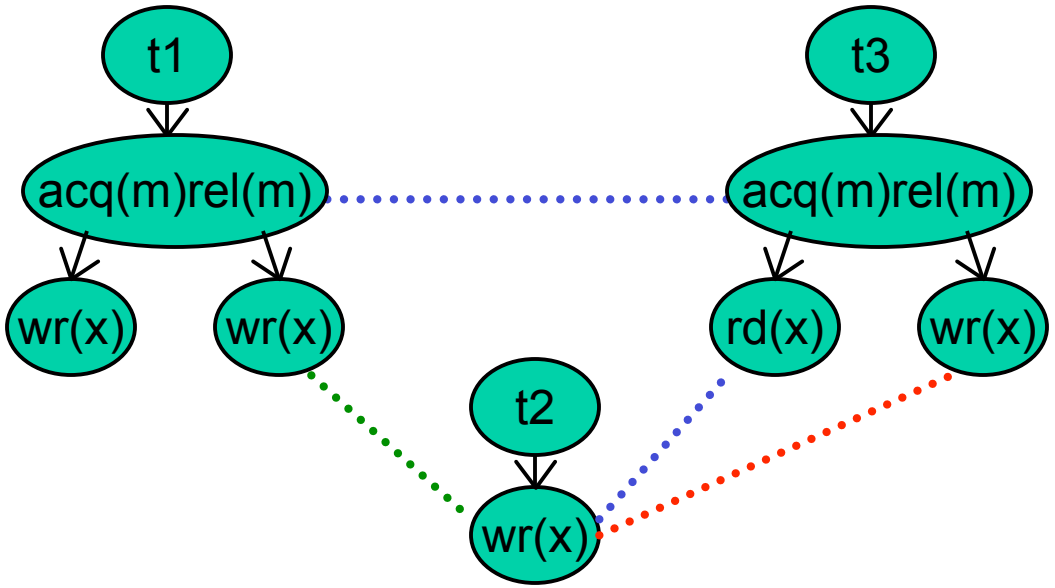
Step 2: Construct Conflict-Forest

- Add **inter-edges** between access trees:
  - wr(x) – rd(x): the read event can read the value written by the write.
    - "Can" means "in some feasible permutation of the execution".
  - wr(x) – wr(x): both write to the same variable.
- If a common lock is held at both events, connect the lock nodes instead of the access nodes.



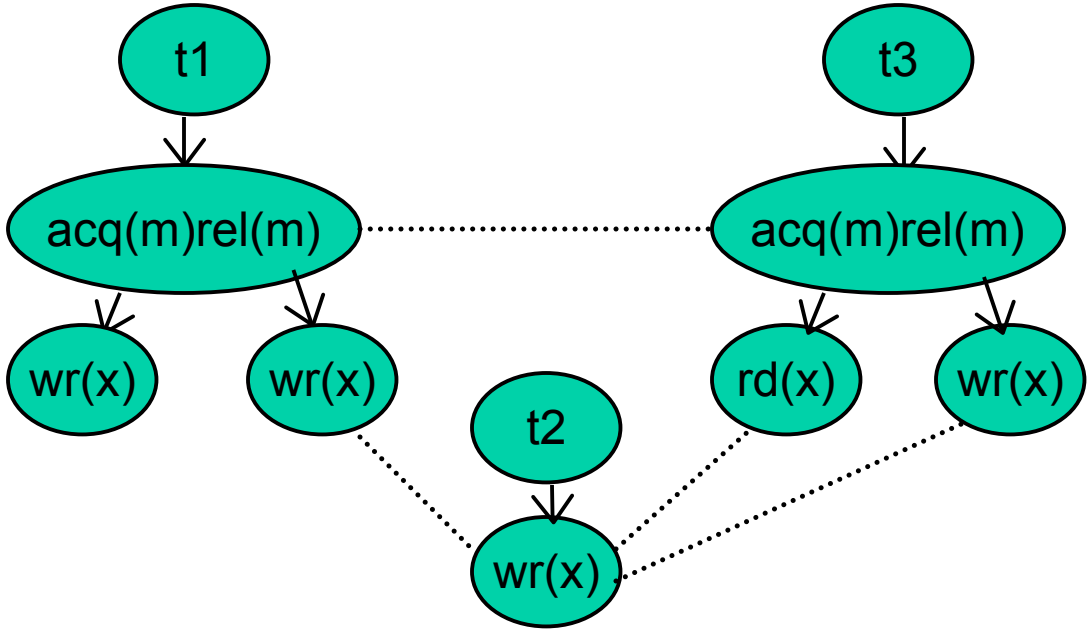
Step 2': Construct View-Forest

- Add **inter-edges** between access trees:
  - wr(x) – rd(x): the read event can read the value written by the write.
  - wr(x) – wr(x): both written values can be read by the same read event.
  - fnlWr(x) – fnlWr(x): both events are the final writes to the same variable in their transactions.
- If a common lock is held at both events, connect the lock nodes instead of the access nodes.



Commit-Nodes

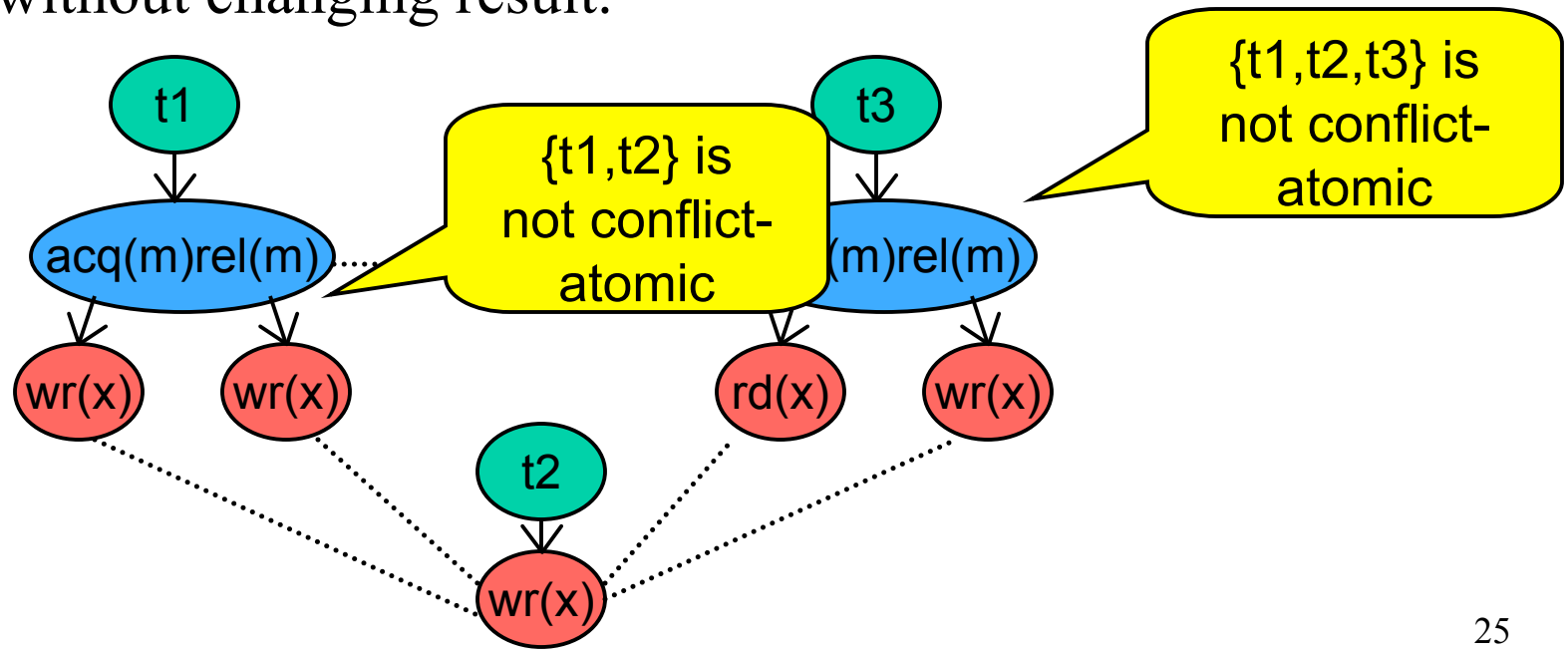
- **Communication node**: end-point of an inter-edge.
- **Commit node**: a communication nodes without communication node descendants.



A more sophisticated theorem appeared in Wang and Stoller PPoPP06

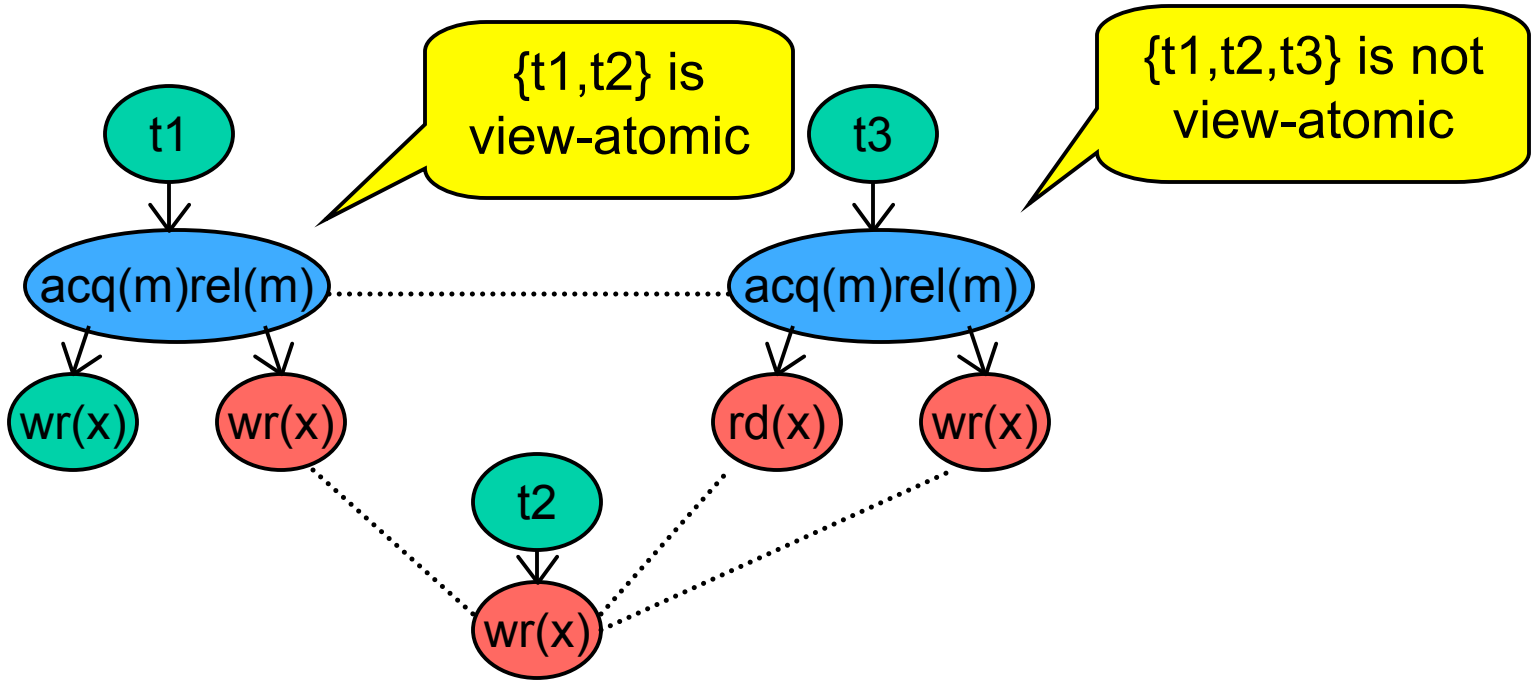
Step 3: Check Conflict-Atomicity by Counting Commit Nodes

- **Theorem**: For a set **T** of transactions without potential for deadlock, if every transaction in **T** has at most one commit node in the conflict-forest, then **T** is conflict-atomic.
- **Intuition**: If a transaction **t** has at most one commit node, it is like a commit point: all events of **t** can be moved there without changing result.

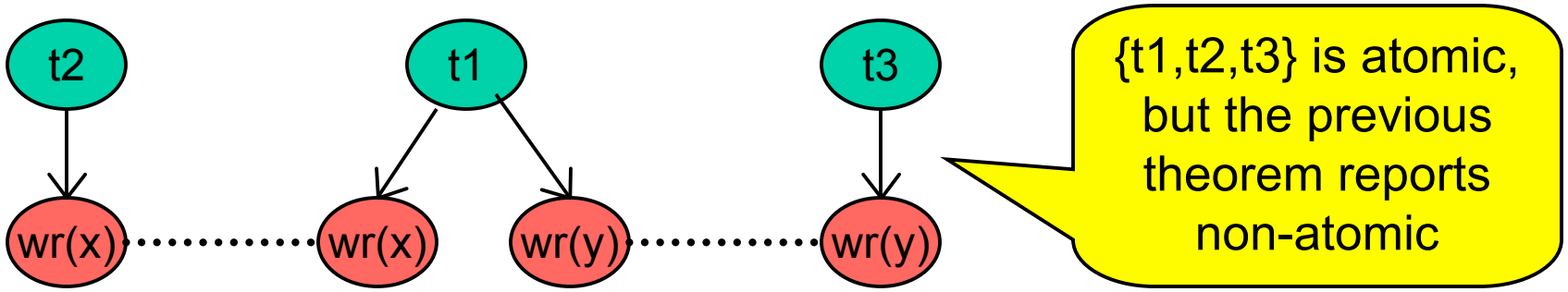


Step 3: Check View-Atomicity by Counting Commit Nodes

- **Theorem**: For a set **T** of transactions without potential for deadlock, if every transaction in **T** has at most one commit node in the view-forest, then **T** is view-atomic.
- **Intuition**: same as the previous theorem.



A More Sophisticated Theorem



- **Theorem**: For a set **T** of transactions without potential for deadlock, if ∀ **t**∈**T**, ∀ communication nodes **n1**,**n2** ∈ **t,** where **n1** and **n2** do not contain each other, **n1** and **n2** are not involved in any cycle of the **conflict-forest** (**view-forest**) , then **T** is **conflict-atomic** (**view-atomic**).

Related Work

- Runtime Analysis for Atomicity
  - Reduction-based algorithm [Wang and Stoller 2003,2006]
  - Block-based algorithm [Wang and Stoller 2003,2006]
  - Atomizer [Flanagan and Freund 2004]
- Static Analysis for Atomicity
  - Type system [Flanagan and Qadeer 2003] [Sasturkar et. al. 2005]
  - Method consistency [von Praun 2004]
- Hybrid (static + runtime) Analysis for Atomicity [Sasturkar et. al. 2005] [Agarwal et. al. 2005]
- Atomic sets [Vaziri 2006]

Outline

- Introduction to atomicity
- Conflict-atomicity and view-atomicity
- Runtime analysis
- Commit-node algorithm
- **Experiments and conclusions**

Summary of Experiments

- Evaluated on 12 benchmarks totaling 39 KLOC.
- **Heuristic:** public or synchronized methods are treated as transactions.

	Atomizer (Flanagan et. al.)	Reduction-based	Block-based	Commit-node
Accuracy	least	less	most (exact)	less than block-based in theory, but same in practice.
	9 bugs 14 benign alarms 38 false alarms	10 bugs 20 benign alarms 25 false alarms	10 bugs 20 benign alarms 0 false alarms	10 bugs 20 benign alarms 0 false alarms
Performance	fast	fast	slow	fast
	median slowdown 38x than original	same as Atomizer	227% slower than reduction-based	same as Atomizer
Diagnostic Information	less	less	more	medium

Conclusions and Future Work

- **Conclusions**
  - Checking conflict-atomicity $\overset{\text{polynomial reducible}}{\Longleftrightarrow}$ checking view-atomicity.
  - Speed: commit-node alg ≈ reduction-based alg.
  - Accuracy: commit-node alg = block-based alg in practice.
- **Future work**
  - Reduce the overhead of the runtime commit-node algorithm.
  - Consider other kinds of synchronization.
- **Questions?**