the complexity of predicting atomicity violations

Azadeh Farzan
Univ of Toronto

P. Madhusudan
Univ of Illinois at Urbana Champaign

.

UPCRC Illinois
Universal Parallel Computing Research Center

Motivation: Interleaving explosion problem

- Testing is the main technique for correctness in the industry
- Fundamentally challenged for concurrent programs:
  - Given even a ***single*** test input for a concurrent program, testing is hard!
  - Too many interleavings to check thoroughly

Idea: Select a small subset of interleavings to test that are likely to expose concurrency bugs

How to select schedules cleverly

– CHESS: Microsoft Research

  Explores all possible interleavings with at most k context-switches, for a small k.

  We believe atomicity errors will constitute a far smaller but more interesting class of runs to test.

– A bunch of tools that try to somehow come up with interleavings that may have errors

  - Eg. ConTest: IBM

– Our view:

  Don't look randomly for schedules!

  Look systematically for interesting patterns
  of thread interaction that are more likely to have errors.

In this talk: Atomicity

Atomicity :

One particular high-level pattern that gets violated in many concurrency bugs:

- A local piece of code needs to access shared data without (real) interference from other threads.
- Extremely common intention, the violation of which leads to many errors.
- In concurrency bug studies, we as well as others (Lu-Park-Seo-Zhou'08) have found that the majority of errors (~70%) are due to atomicity violations.
- Hence finding executions that violate atomicity and testing them is a good way to prune the interleavings to test!

Atomicity error: example

- https://bugzilla.mozilla.org/show_bug.cgi?id=290446

- Summary:
  Update of remote calendar does not use WebDAV locking (concurrency control)

- When updating/inserting a new event in a remote WebDAV calendar, the calendar file is not locked. In order to avoid losing data the concurrency control of WebDAV should be used (Locking).

- Steps to Reproduce:
  - 1. User A starts creating a new event in the remote calendar
  - 2. User B starts creating a new event in the remote calendar
  - 3. Users A and B read-modify-write operations are interleaved incorrectly

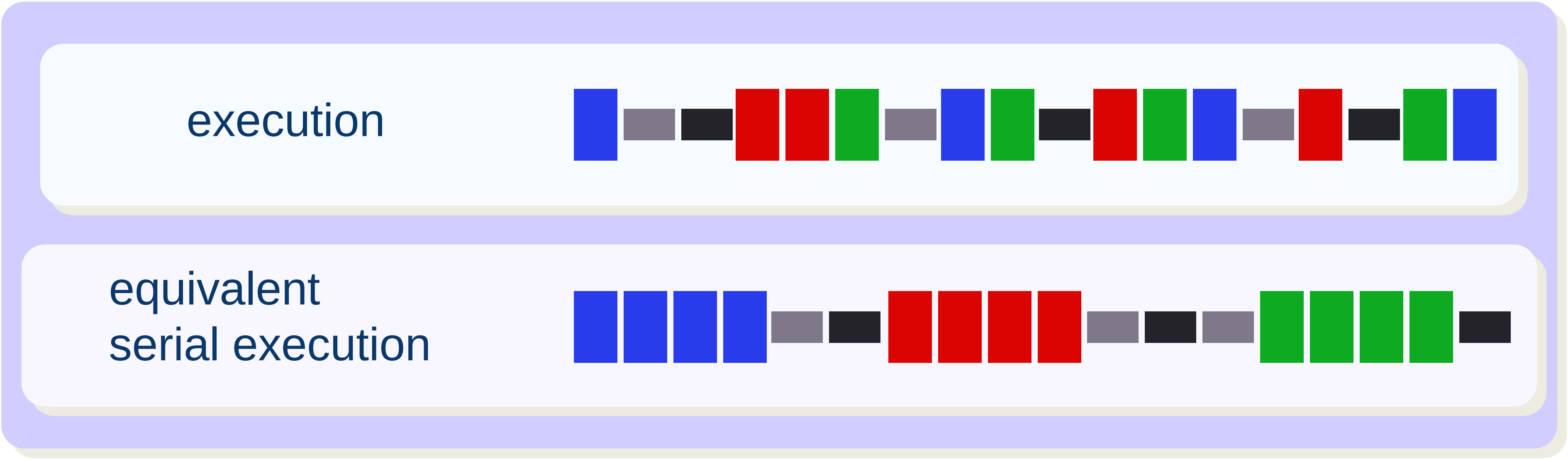
•Actual Results: The actual sequence could/would be: 1. User A - GET test.ics 2. User B - GET test.ics 3. User A - PUT test.ics 4. User B - PUT test.ics

- *In this case the new event posted by user A is lost.*

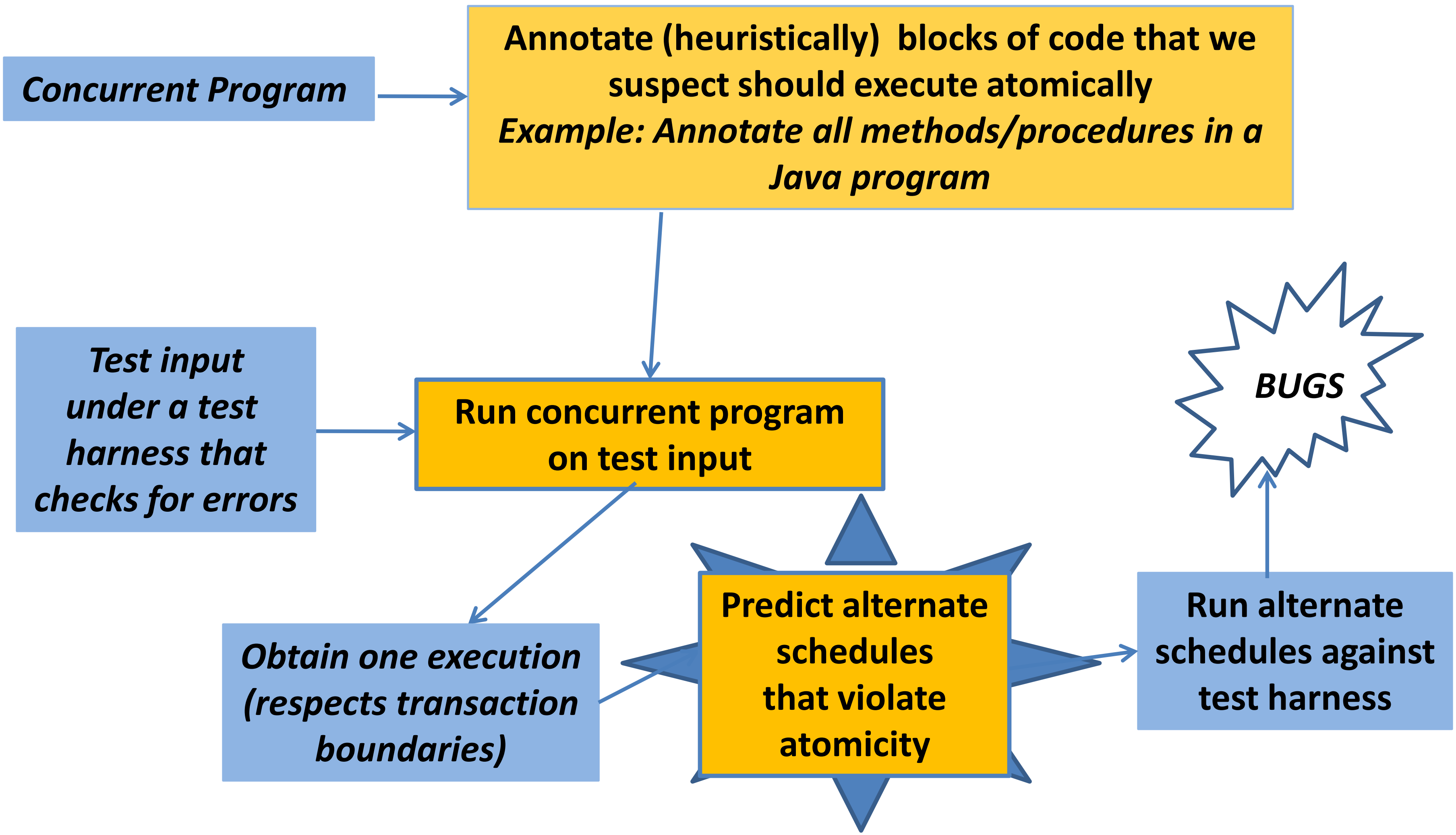
Atomicity

- Transaction: sequential logical unit of computation: syntactically identified: small methods, procedures, etc.

- An execution r of a concurrent program P is atomic if there exists an equivalent run of P in which every transaction is non-interleaved.



Application: Finding bugs while testing



Main problem

- Given programs $P_1 \,||\, P_2 \,||\ldots P_n$ where
  - Each P_i is be a straight-line program

    (useful when attacking the testing problem)
  - Each P_i is be a regular program

    (modeled as finite automata; useful in abstracted pgms)
  - Each P_i is a recursive program

    (modeled as PDS; for abstracted pgms with recursion)

Question:

Is there any interleaved run that violates atomicity?

Atomicity based on Serializability; When are two runs equivalent?

Events: { T:begin, T:end } U { T:read(x) , T:write(x) | x is a shared var }

Concurrent Run: sequence of events.

Dependence/ Conflicting events:

$$\mathcal{D} = \{(T_1 : a_1, T_2 : a_2) \mid T_1 = T_2 \;\vee\; \exists \mathbf{x} \exists i, j \in \{1, 2\} : a_i = \mathtt{write(x)} \wedge a_j \in \{\mathtt{read(x)}, \mathtt{write(x)}\}\}$$

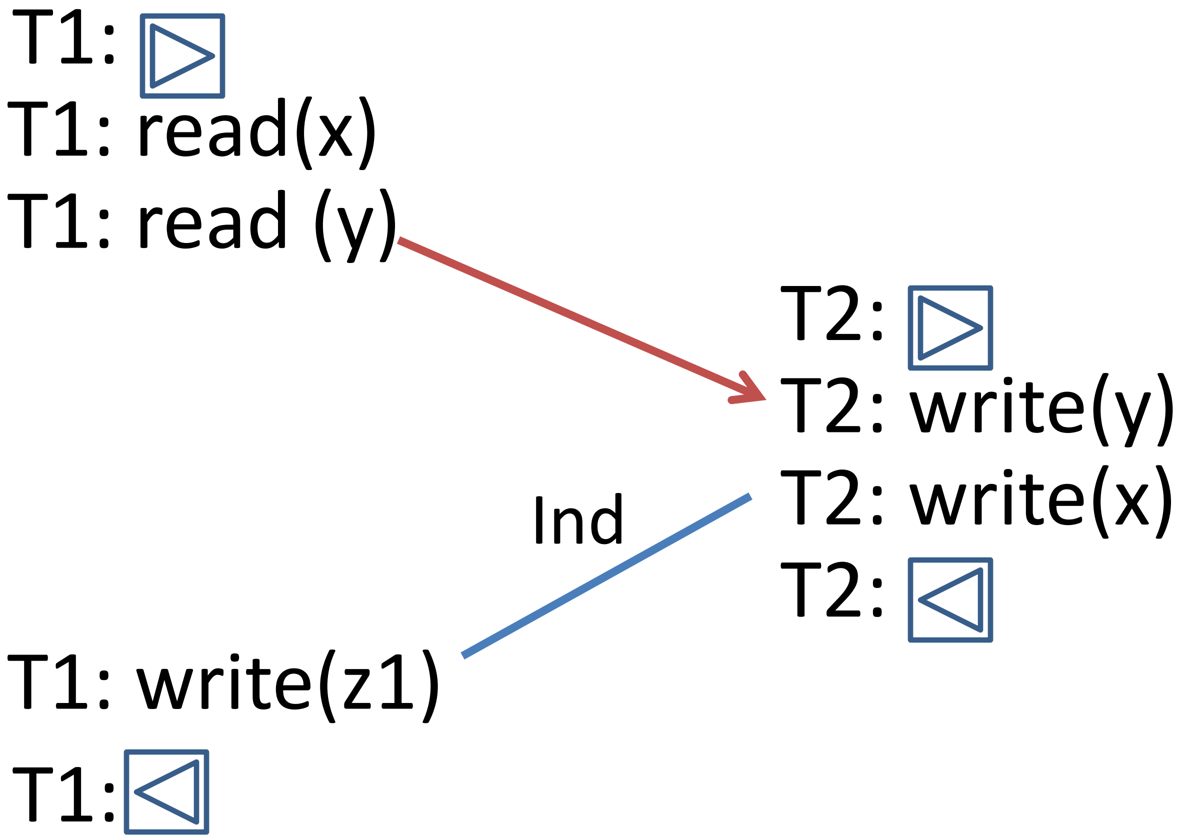
Equivalence of Runs: two runs are equivalent if conflicting events are not reordered

$r \sim r'$ iff for every $e_1 \, D \, e_2$, $r \downarrow \{e_1, e_2\} = r' \downarrow \{e_1, e_2\}$

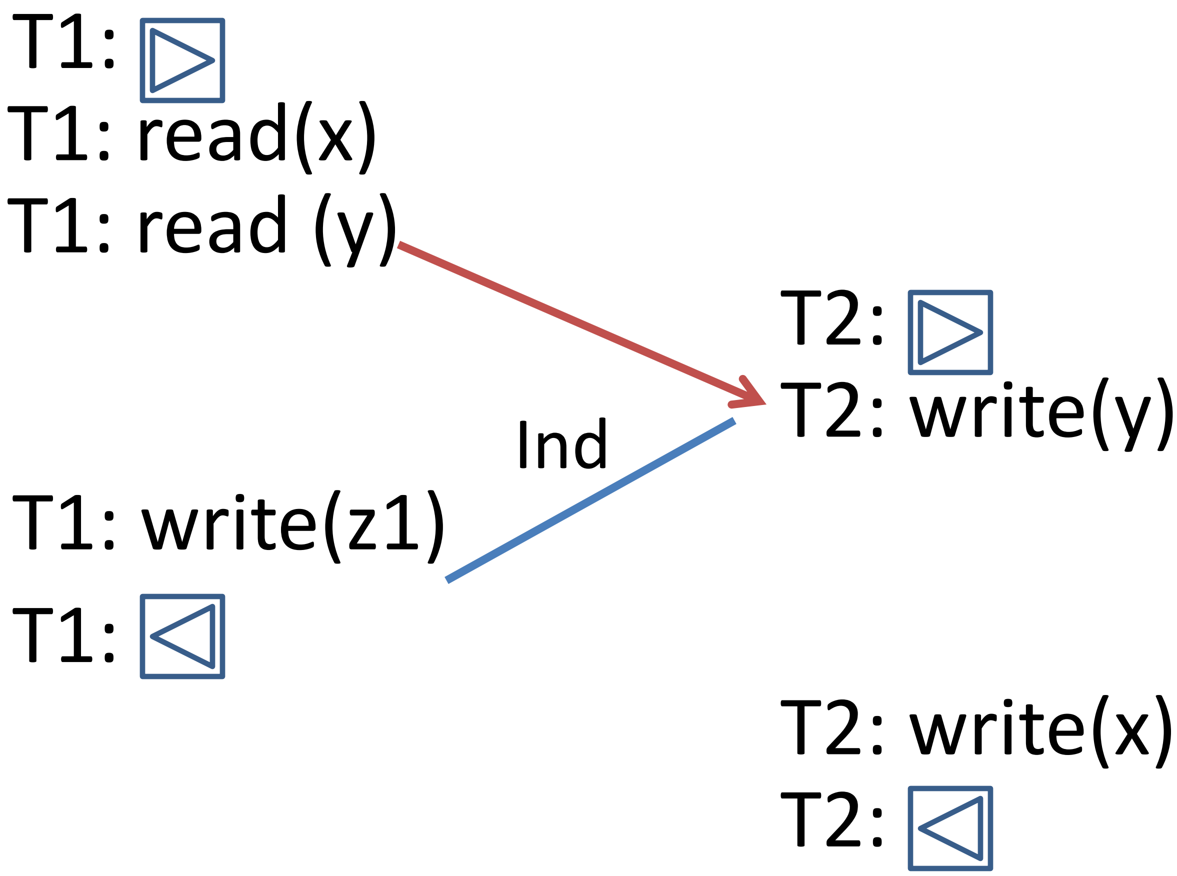
Serial Run: all transactions are executed non-interleaved.

Atomic (Serializable) Run: there exists an equivalent serial run.

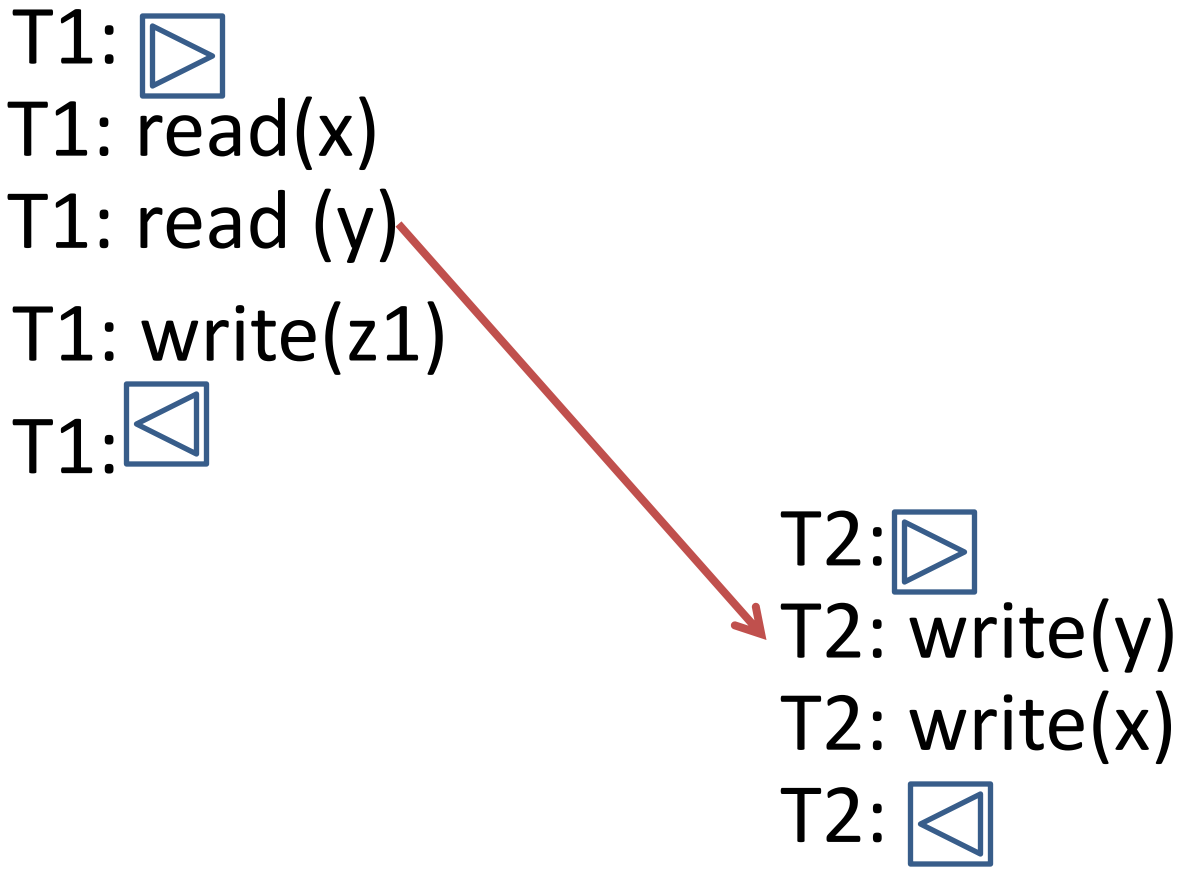
Atomicity based on Serializability



Atomicity based on Serializability



Atomicity based on Serializability

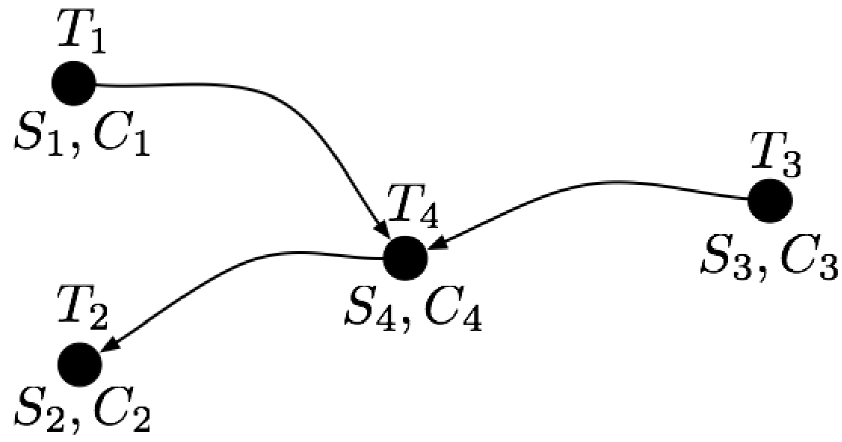


Before we predict, can we monitor atomicity efficiently?

- Monitoring: Given an execution r, is r atomic?

- An extremely satisfactory solution [Farzan-Madhusudan: CAV08]

  We can build sound and complete monitoring algorithms that keeps track of:
   - a set of vars for each thread
   - a graph with vertices as threads



- If #vars = V, # threads = n, then algorithm uses $O(n^2 + nV)$ space.

  Efficient streaming algorithm.

  Independent of length of run!

Predicting Atomicity Violations

Example:

Given programs P1 and P2 (here straight-line) check whether there is an interleaving that violates atomicity.

P1:

```
T1:  begin
T1:  acq (l)
T1:      read(Amount)
T1:  rel (l)
T1:  acq(l)
T1:      write(Amount)
T1:  rel(l)
T1:  end
```

P2:

```
T2:  begin
T2:  acq (l)
T2:      read(Amount)
T2:  rel (l)
T2:  acq(l)
T2:      write(Amount)
T2:  rel(l)
T2:  end
```

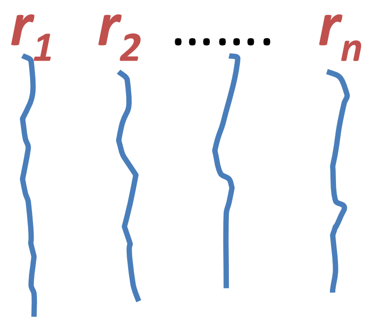
```
T1:  begin
T1:  acq (l)
T1:      read(Amount)
T1:  rel (l)

T2:  begin
T2:  acq (l)
T2:      read(Amount)
T2:  rel (l)
T2:  acq(l)
T2:      write(Amount)
T2:  rel(l)
T2:  end

T1:  acq(l)
T1:      write(Amount)
T1:  rel(l)
T1:  end
```

Interleaved execution of P1 and P2 that violates atomicity

Prediction Model

- Given an execution *r*, look at the local executions

  each thread executes $r_1, r_2, \ldots r_n$



- Can we find another execution *r'* that is obtained by recombining this set of local runs such that *r'* is non-atomic?

- Predicted runs could
  - respect no synchronization constraints (less accurate)
  - respect concurrency control constraints such as locking (more accurate)

- The run *r'* may not be actually feasible!
  - Conditionals in programs may lead the program to different code
  - Certain operations on datastructures may disable other operations ....

- ***Key requirement:***
  We should not enumerate all interleavings!
  Must be more efficient.

Predicting atomicity violations

How to predict atomicity violations for st-line or regular programs?

- **Naïve algorithm:**
  - Explore all interleavings and monitor each for atomicity violations
  - Runs in time $O(k^n)$ for n-length runs and k threads --- infeasible in practice!

- **Better algorithm: Dynamic programming using the monitoring algm**

  - *Predicting from a single run with a constant number of variables, can be done in time*

    $O(n^k\, 2^{k2})$ *---- better than* n^k*, the number of interleavings*

  But even n^k *is huge! Too large to work in practice even for k=2!*
  (m is 100 million events! k=2,..10,..)

  Also, exponential dependence in k is unavoidable (problem is NP-hard).

- *We want to avoid the k being on the exponent of m*
  Main question of the paper: Can we solve in time linear in m? (like $n+2^k$*)*
  i.e. can we remove the exponent k from n?

Main results - I

- **Good news:**

  If prediction need not respect any synchronization constraint (no locks)

  - *Predicting from a single run with a constant number of variables, can be done in time*

    $O(n + kc^k)$ *n=length of runs; k= # threads*

  - *Regular programs also can be solved in time* $O(n + kc^k)$

    where n=size of each local program, k = #threads

  - *Recursive programs are also (surprisingly) decidable.*

    $O(n^3 + kc^k)$

    where n=size of each local program, k = #threads

Main results - II

- **Bad news:**

  If prediction *needs to respect locking*,

  existence of prediction algorithm for regular programs running in time linear in m is **unlikely**.

  In fact, algorithms for regular programs that take time a fixed polynomial in ***n*** is unlikely.

  i.e. O(poly(m). f(k)) for ***any*** function f() is unlikely!

  The problem is **W[1]-hard**.

- Also, prediction for concurrent recursive programs in the presence of locks is undecidable.

Prediction without synchronization constraints

- Idea: Compositional reasoning
  - Extract from each local thread run a small amount of information (in time linear in the run)
  - Combine the information across threads to check for atomicity violations
  - Information needed from each local run is called a *profile*.

Profiles

- *Key idea:*

  If there is a serializability violation, then there are really only two events in each thread that are important!

  Also, we need to know if these events occur in the same transaction or not.

Let r be a local run of a thread T.

Profiles of r are:

- T:beg T:a T:end — event a occurs in r
- T:beg T:a T:b T:end — a occurs followed by b within the *same transaction*
- T:beg T:a T:end T:beg T:b T:end — a occurs followed by b but in *different transactions*

Reasoning atomicity using profiles

Key lemma:

A set of programs with no locks (straight-line, regular or recursive) has a non-serializable execution ***iff***

there is a profile of each local program such that the profiles, viewed as a program, have a non-serializable execution.

Proof idea: skeleton of a serializability violation:

Only two events per thread are needed to witness "cycle" for non-serializability

Prediction without synchronization constraints

- Straight-line and regular programs: $O(n+kc^k)$ time
  - Extract profiles from each local program

    $O(n)$ time --- constant number of profiles
  - Check if the profiles have a serializability violation

    $O(kc^k)$ time – check all possible interleavings of profiles for serializability violations

- Recursive programs: $O(n^3+kc^k)$ time
  - Extract profiles from each local thread using PDS reachability
    - *$O(n^3)$ time*
  - Check if profiles have a serializability violation $O(kc^k)$ time

Prediction with locking constraints

- Consider a set of regular programs $P_1 \mid\mid P_2 \mid\mid \ldots . P_n$
- Then it is *unlikely* that the atomicity prediction problem is solvable in time O(poly(n). f(k)) for any function f !

i.e. we cannot remove the exponent k from n

- How do we prove this?
  - Using parameterized complexity theory
  - The problem is W[1]-hard (with parameter k).

Prediction with locking constraints

- Parameterized complexity theory:
  - Consider an algorithmic problem where input is of length n, but every instance has a parameter k associated with it.
  - A problem is fixed-parameter tractable (FPT) over parameter k if it is solvable in time O(poly(n). f(k)) where f() is any function.
  - I.e. solved in a fixed-polynomial time in n, for any k.
- W[1]-hard problems
  - No fixed-parameter algorithms known
  - Believed not to be FPT.
- Example:
  - Vertex cover is FPT in parameter k=number of colors
  - Independent-set is W[1]-hard in parameter k = number of sets

Prediction with locking constraints

- Prediction of atomicity violations in regular programs is W[1]-hard
- Hence an algorithm that runs in time O(poly(n).f(k)) is unlikely
  (let alone an algorithm that runs linear in n).

- Proof is by a (parameterized) reduction from the finite-state automata intersection problem (where the parameter is the number of automata), which is known to be W[1]-hard.

- Note:
  - Prediction of atomicity violations in straight-line programs is still open!

- Prediction of atomicity violations in recursive programs is undecidable
  - not surprising as locks can be used to communicate (Kahlon et al)

Current and future directions

- Key project:
  - Testing tool that executes alternate schedules that violate atomicity in order to find bugs.
  - More recent work has shown that nested locking yields tractable algorithms! (using ideas from Kahlon et al)
  - For non-nested locking, in practice, one can do more coarse analysis simply using locksets, and this yields reasonably good prediction algorithms.
- Open problem:
  - Atomicity for straight-line programs with locks still open.