Static and Dynamic Verification of Concurrent Programs

Aarti Gupta
Systems Analysis & Verification
NEC Labs America, Princeton, USA

Third Summer School on Formal Techniques
May 20 – 24, 2013

NEC Laboratories America
Relentless passion for innovation

www.nec-labs.com

Acknowledgements

- Vineet Kahlon*, Chao Wang*, Nishant Sinha*, Pallavi Joshi (NEC Labs)

- Akash Lal (Microsoft Research, India)
- Madanlal Musuvathi (Microsoft Research)
- Kedar Namjoshi (Alcatel-Lucent)
- Chang-Seo Park (UC Berkeley, now at Google)
- Andrey Rybalchenko, Ashutosh Gupta, Corneliu Poppea (TU Munich), Alexander Malkis (Imdea)
- Arnab Sinha (Princeton University, now at Microsoft)
- Tayssir Touili (LIAFA)

Motivation

❑ Key Computing Trends



Mobile   Server   Gaming

Low Power, High Performance

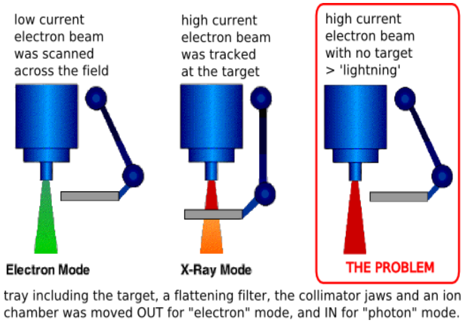
- **Multi-core platforms everywhere**
- ***Need parallel, multi-threaded programming***

Data centers, Cloud platforms

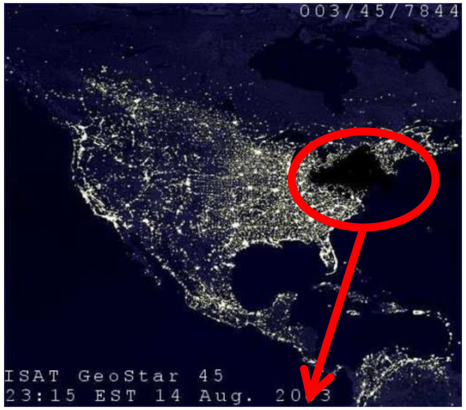
- **Distributed systems**

❑ Parallel/Multi-threaded Programming

- **Difficult to program**
  - **Dependencies due to shared data**
  - **Subtle effects of synchronizations**
- **Difficult to debug**
  - **too many interleavings of threads**
  - **hard to reproduce bugs**



Therac-25 medical radiation device (1985) malfunction due to SW race, at least 5 deaths



2003 Northeast Blackout
Cost: $4 billion

Nasdaq's Facebook glitch came from 'race conditions'
Nasdaq may pay out as much as $13 million due to a hard-to-find software bug

What will I (try to) cover?

- **Basic elements**
  - **Model of concurrency**
    - **Asynchronous interleaving model (unlike synchronous hardware)**
    - **Explosion in interleavings**
  - **Synchronization & Communication**
    - **Shared variables: between threads or shared memory for processes**
    - **Locks, semaphores: for critical sections, producer/consumer scenarios**
    - **Atomic blocks: for expressing atomicity (non-interference)**
    - **Pair-wise rendezvous**
    - **Asynchronous rendezvous**
    - **Broadcast: one-to-many communication**
  - **On top of other features of sequential programs**
    - **Recursive procedures, Loops, Heaps, Pointers, Objects, …**
    - **(Orthogonal concerns and techniques)**
- **Will cover Static and Dynamic verification techniques**
  - **Model checking, Abstract interpretation, Systematic testing, …**

What I will not be able to cover

- **Active topics of research**
  - Theorem-proving , type systems, runtime monitoring
  - Separation logic: pointers & heaps, local reasoning

  - Parallel programs: Message-passing (e.g. MPI libraries), HPC applications
  - Memory models: Relaxed memory models (e.g.TSO), Transactional memories

  - Synthesis/Optimization of locks/synchronizations
  - Concurrent data structures/libraries: Lock-free structures
  - Object-based verification: Linearizability checking

Models for Verifying Concurrent Programs

- **Finite state systems**
  - Asynchronous composition of processes, including buffers/channels for messages, no recursion
  - Usage: Inline procedures up to some bound to get finite models
  - Techniques: Bounded verification

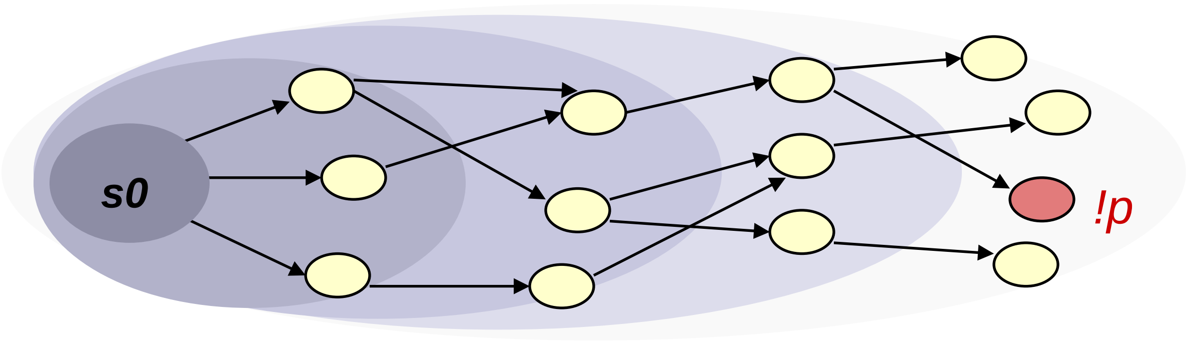
- **Sequential programs**
  - Recursive procedures and other features, no synchronization or communication, no interleavings
  - Usage: add synchr-comm, interleavings (thread interference)
  - Techniques: Bounded as well as unbounded verification

- **Pushdown system models**
  - Stack of a pushdown system (PDS) models recursion, finite control, data is finite or infinite (with abstractions)
  - Usage: System of interacting PDSs, interactions may be restricted
  - Techniques: PDS-based model checking

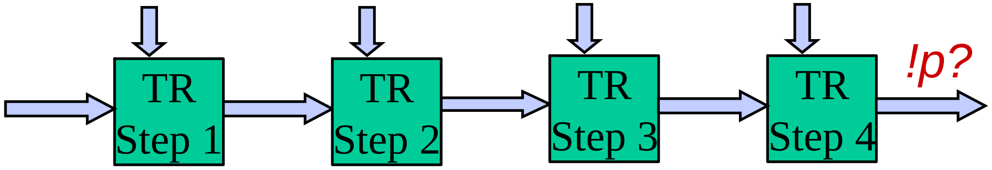
Model Checking

- **Model Checking**
  - Exhaustive state space exploration
  - Maintains a representation of visited states (explicit states, symbolic states, …)
  - Expensive, needs abstractions and approximations
- **Bounded Model Checking**
  - State space search for bugs (counterexamples) or inputs for test cases
  - Typically does not maintain representation of visited states
  - Less expensive, but needs good search heuristics

Model Checking AG p
Does the set of states reachable from s0 contain a bad state(s)?



Bounded Model Checking
Is there is a path from the initial state s0 to the bad state(s)?



Outline

- ✓ **Introduction**

- ❑ **PDS-based Model Checking**
  - Theoretical results

- ❑ **Static Verification**
  - Reduction: Partial order reduction
  - Abstraction and Composition: Static analysis, Thread-modular reasoning
  - Bounding: Context-bounded analysis, Memory Consistency-based analysis

- ❑ **Dynamic Verification**
  - Preemptive Context Bounding
  - Predictive Analysis
  - Active Testing
  - Coverage-guided Systematic Testing

- ❑ **Summary & Challenges**

Pushdown System (PDS) Model

- **Each thread is modeled as a PDS**
  - **Finite Control : models control flow in a thread (data is abstracted)**
  - **Stack : models recursion, i.e., function calls and returns**

- **PDS Example**

States: {s,t,u,v}

Stack Symbols: {A,B,C,D}

Transition Rules: <s,A> → < t, *e* >

<s,A> → < t, B >

<s,A> → < t, C B >

If the state is s, and A is the symbol at the top of the stack, then transit to state t, pop A, and push B, C on the stack

PDS-based Model Checking

- **Close relationship between Data Flow Analysis for sequential programs and the model checking problem for Pushdown Systems (PDS)**
  - **The set of configurations satisfying a given property is regular**
  - **Has been applied to verification of sequential Boolean programs**

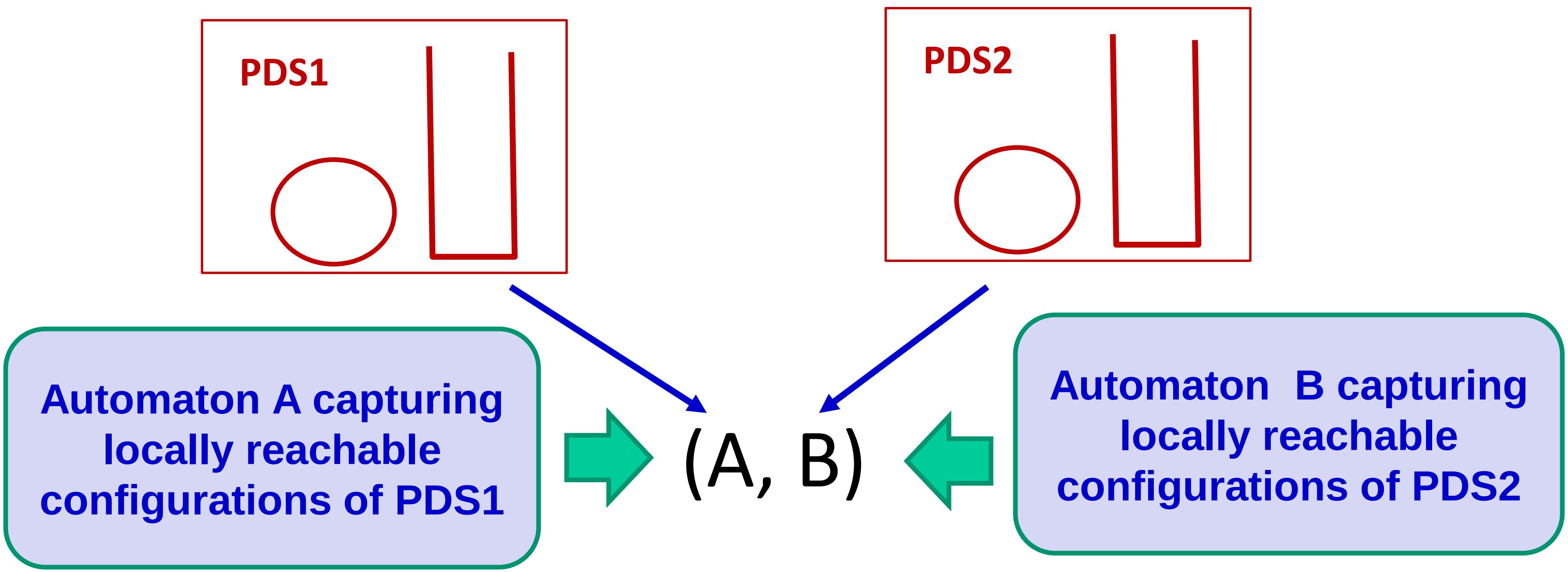
[Bouajjani *et al*., Walukeiwicz, Esparza *et al.*]

- **Analogous to the sequential case, dataflow analysis for concurrent program reduces to the model checking problem for interacting PDSs**

- **Problems of Interest: To study multi-PDSs interacting via the standard synchronization primitives**
  - **Locks**
  - **Pairwise and Asynchronous Rendezvous**
  - **Broadcasts**

Interacting PDSs

- Problem: For multi-PDS systems, the set of configurations satisfying a given property is not regular, in general
- Recall: Set of configurations is regular for individual PDS
- Strategy: Compute locally reachable configurations of individual PDS, and leverage cases of "loose coupling"



Key Challenge

Capture interaction based on synchronization patterns

Capturing Interaction in presence of Synchronizations

- **Key primitive: *Static Reachability***
  - **A global control state t is *statically reachable* from state s**

    if there exists a computation from s to t that respects the constraints imposed by synchronization primitives,

    e.g., locks, wait/notifies, …

- **However, static reachability is undecidable**
  - **for pairwise rendezvous** [Ramalingam 00]
  - **for arbitrary lock accesses** [Kahlon *et al.* 05]
  - **Undecidability hinges on a close interaction between synchronization and recursion**
  - **(Note: Even for finite data abstractions)**

- **How to get around this undecidability?**
  - **Special cases of programming patterns: Nested Locks, Bounded Lock Chains**
  - **Place restrictions on synchronization and communication**

Programming Pattern: Nested Locks

Nested Locks:
Along every computation, each thread can only release that lock which it acquired last, and that has not yet been released

- **Example:**

```
f( ) {
    acquire(b) ;
    g ( );
    // h ( );
    release(c);
}

g( ){
    acquire(a);
    release(a);
    release(b);
    acquire(c);
}

h( ){
    acquire(c);
    release(b);
}
```

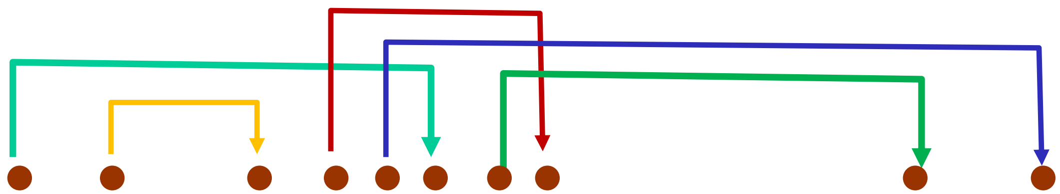
f calls g: nested locks
f calls h: non-nested locks

- **Programming guidelines typically recommend that programmers use locks in a nested fashion**

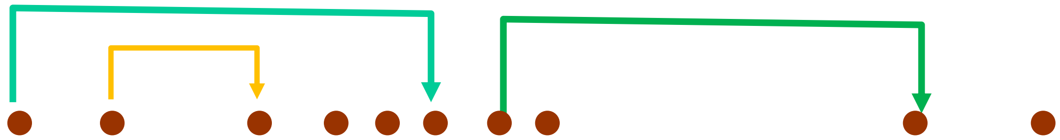
- **Multiple locks are enforced to be nested in Java$_{1.4}$ and C#**

Programming Pattern: Lock Chains

- **Lock Chains**

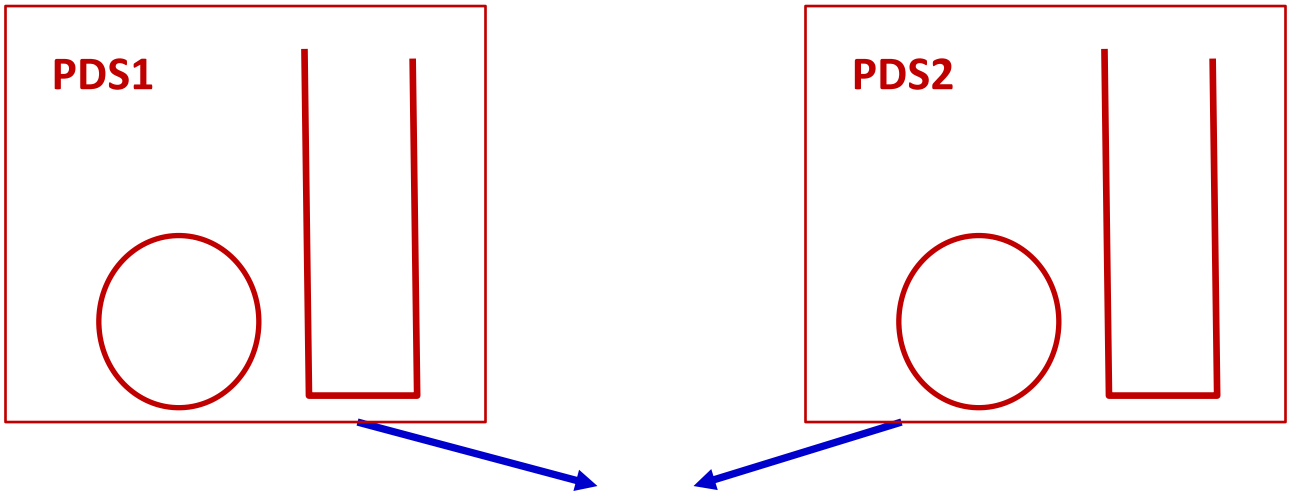


- **Nested Locks: Chains of length one**



- **Most lock usage is nested**
- **Non-nested usage occurs in niche applications, often bounded chains**
  - **Serialization, e.g. 2-phase commit protocol uses chains of length 2**
  - **Interaction of mutexes with synchronization primitives like wait/notify**
  - **Traversal of shared data structures, e.g. length of a statically-allocated array**

Interacting PDSs with Locks



(A, B)

Key Challenge: Capture interaction based on synchronization patterns

General Problem for arbitrary lock patterns: Undecidable [Kahlon *et al.* CAV 2005]

For nested locks and bounded lock chains: Decidable

[Kahlon *et al.* POPL 07,LICS 09,CONCUR 11]

- **Tracks lock access patterns thread-locally as regular automata**
- **Incorporates a consistency check in the acceptance condition**

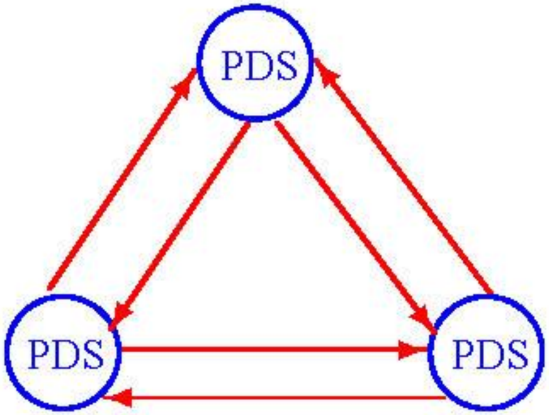
Restrict Synchronization & Communication: Example

Reachability is decidable for PDS Networks with: [Atig et al. 08]

- **acyclic communication graph**
- **lossy FIFO channels**

Boolean sequential programs ⇔ Pushdown system (PDS)

Parallel processes communicating via channels ⇔ Network of pushdown systems communicating via channels



PDS-based Model Checking: Summary

Reachability Problem

- ❑ **Undecidable for Pairwise Rendezvous** **[Ramalingam 00]**
- ❑ **Undecidable for PDSs interacting via Locks** **[Kahlon *et al.* CAV 05]**
- ❑ **Decidable for PDSs interacting via Nested Locks** **[Kahlon *et al.* CAV 05]**
- ❑ **Decidable for PDSs interacting via Bounded Lock Chains**
  [Kahlon LICS 09, CONCUR 11]

Reachability/Model Checking is Decidable under Other Restrictions

- **Constrained Dynamic Pushdown Networks** **[Bouajjani *et al*. TACAS 07]**
- **Asynchronous Dynamic Pushdown Network** **[Bouajjani *et al*. FSTTCS 05]**
- **Reachability of Acyclic Networks of Pushdown Systems**
  [Atig *et al.* CONCUR 08]
- **Context-bounded analysis for concurrent programs with dynamic creation of threads** **[Atig *et al.* TACAS 09]**

Practical Verification of Concurrent Programs

- **Hard to apply PDS-based methods directly**
  - **Huge gap between model and modern programming languages**

- **In addition to state space explosion due to data (as in finite state systems and sequential programs)**

  the complexity bottleneck is exhaustive exploration of interleavings

- **The next section describes various strategies to tackle this in practice**
  - **Reduce number of interleavings to consider**
    - **Partial Order Reduction (POR)**
  - **Use program abstractions and compositional techniques**
    - **Static analysis**
    - **Thread-modular reasoning**
  - **Bound the problem**
    - **Context-bounded analysis**
    - **Memory Consistency-based analysis**

Some Preliminaries

- **What is checked in practice?**

- **Common concurrency bugs**
  - Dataraces, deadlocks, atomicity violations

- **Standard runtime bugs**
  - Null pointer dereferences
  - Memory safety bugs

- **Properties**
  - Safety, e.g. mutual exclusion
  - Liveness, e.g. absence of starvation

Common Concurrency Bugs

- **Race Condition**: **simultaneous memory access (at least one write)**

```
/*--- Thread 1 ----*/
 . . .
Write (globalVar);
 . . .
```

```
/*--- Thread 2 ----*/
. . .
Read (globalVar);
. . .
```

- **Deadlock: hold-and-wait cycles**

```
/*--- Thread 1 ---*/
 lock(A);
 . . .
 lock(B);
```

```
/*--- Thread 2 ---*/
 lock(B);
 . . .
 lock(A);
```

- **Atomicity violation: interference from other threads/processes**

```
/*--- Thread 1 ----*/
 if (account_ptr != NULL) {
   ...
   account_ptr -> amount -= debit;
 }
```

```
/*--- Thread 2 ---*/
if (account_ptr != NULL) {
  free(account_ptr);
  account_ptr = NULL;
}
```

Data Race Detection

- **Data Race: If two *conflicting* memory accesses happen *concurrently***

- **Two memory accesses *conflict* if**
  - **They target the same location**
  - **They are not both read operations**

- **Data races may reveal synchronization errors**
  - **Typically caused because programmer forgot to take a lock**
  - **Many programmers tolerate "benign" races**
  - **Racy programs risk obscure failures caused by memory model relaxations in the hardware and the compiler**

Data Race Detection: Basics (1)

- **Two popular approaches for datarace detection**

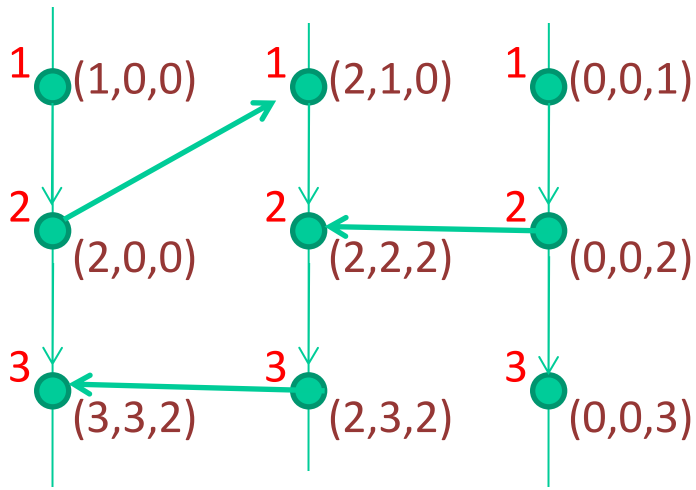
- **Lockset analysis** [Savage *et al*. 97, ERASER]
  - **Definition**
    - ***Lockset(l): The set of locks held at program location l***
  - **Method**
    - **Compute locksets for all locations in a program (statically or dynamically)**
    - **Race: When there are conflicting accesses from program locations with disjoint locksets**
  - **Gives too many false warnings, since program locations may not be reachable concurrently**

  - **Opportunity for more precise analysis (discussed in static analysis)**

Happens-Before Order

[Lamport]

- **Use logical clocks and timestamps to define a partial order called *happens-before* on events in a concurrent system**
- **States *precisely* when two events are *logically* concurrent (abstracts away real time)**



- Cross-edges from send events to receive events
- (a_1, a_2, a_3) happens before (b_1, b_2, b_3) iff $a_1 \leq b_1$ and $a_2 \leq b_2$ and $a_3 \leq b_3$

- **Distributed Systems: Cross-edges from send to receive events**
- **Shared Memory Systems: Cross-edges represent *ordering effects of synchronization***
  - **Edges from lock release to subsequent lock acquire**
  - **Long list of primitives that may create edges: Semaphores, Waithandles, Rendezvous, System calls (asynchronous IO)**

Data Race Detection: Basics (2)

❑ Happens-Before (HB) analysis

- ***Happens-Before order: a partial order over synchronization events***

  [Lamport 77]

- **Method:**
  - **Observe HB order during dynamic execution**
  - **Race: If conflicting accesses are not ordered by HB**

- **This is precise, but dynamic executions have limited coverage**

➢ **Opportunity for improving coverage over alternate schedules (discussed later in predictive analysis)**

Outline

- ✓ **Introduction**
- ✓ **PDS-based Model Checking**
  - ✓ **Theoretical results**
- ➢ **Static Verification**
  - **Reduction: Partial order reduction**
  - **Abstraction and Composition: Static analysis, Thread-modular reasoning**
  - **Bounding: Context-bounded analysis, Memory Consistency-based analysis**
- ❑ **Dynamic Verification**
  - **Preemptive Context Bounding**
  - **Predictive Analysis**
  - **Active Testing**
  - **Coverage-guided Systematic Testing**
- ❑ **Summary & Challenges**

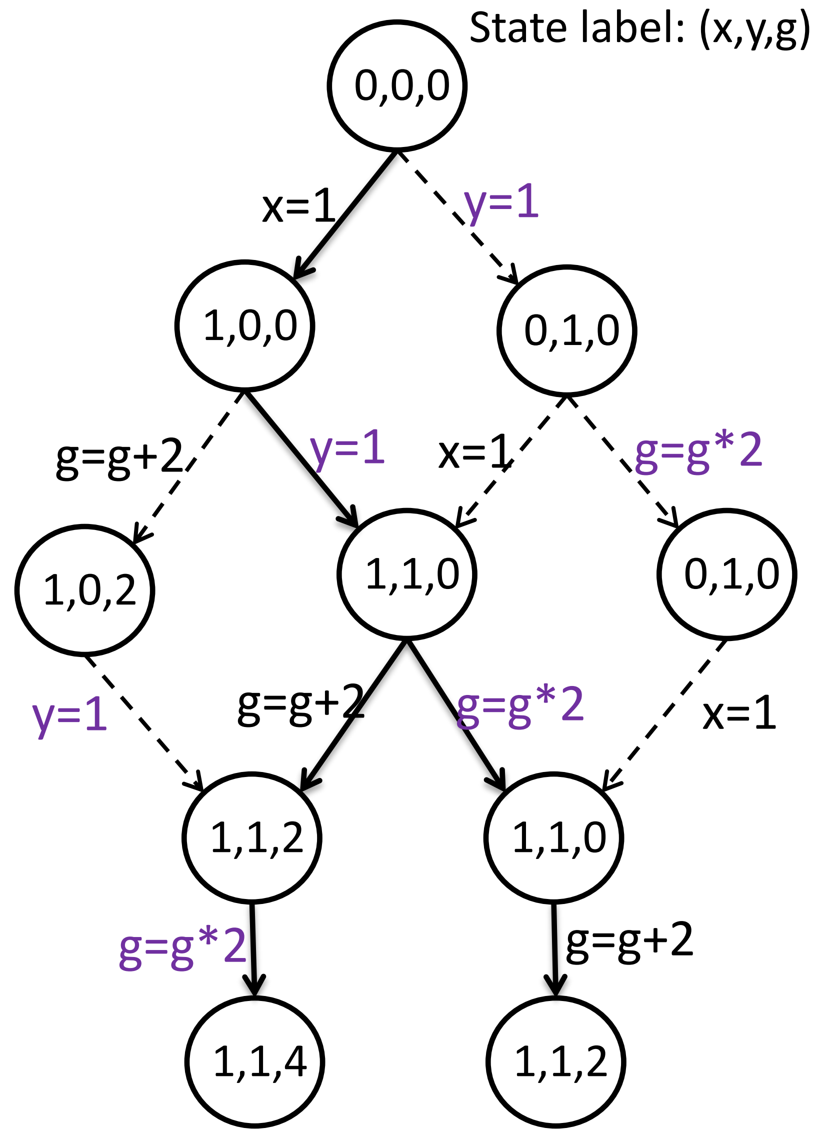
Partial Order Reduction (POR)

Consider the following thread executions.

Thread 1	**Thread 2**
x=1	y=1
g=g+2	g=g*2

The full-blown state-space can be large.

Good news: *the order of independent events does not affect the state that is reached.*



Partial Order Reduction (POR)

Consider the following thread executions.

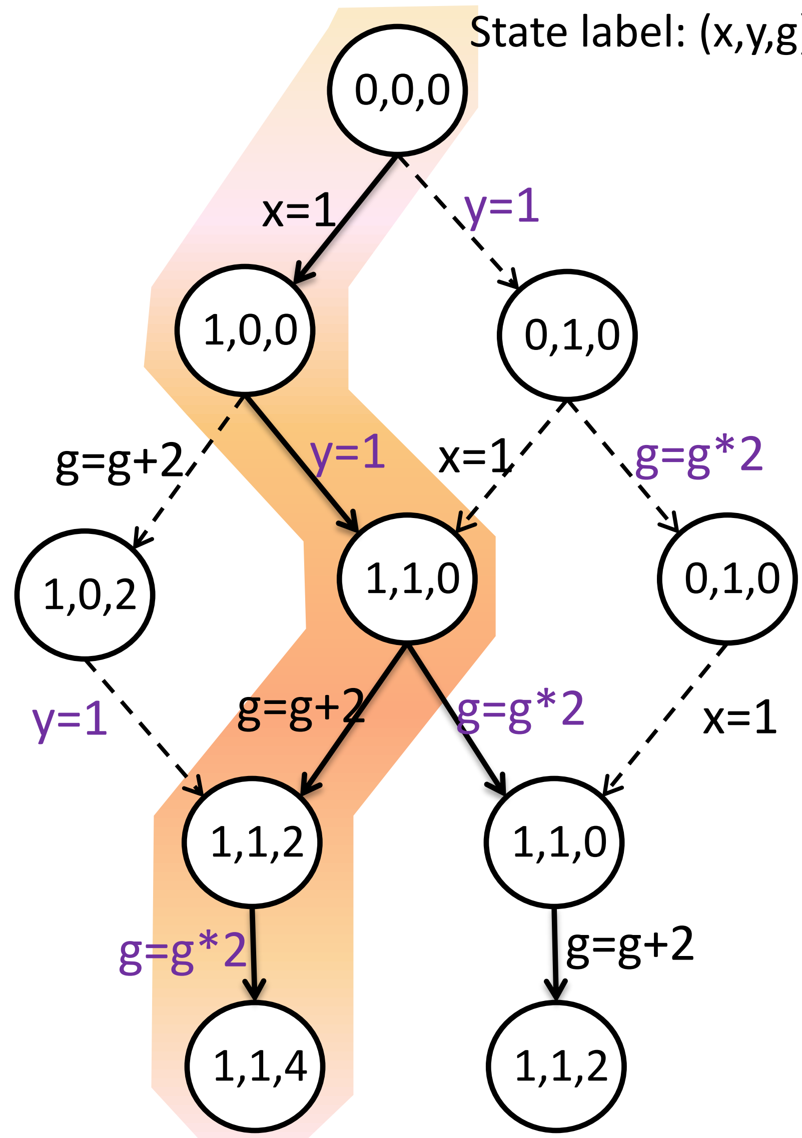
Thread 1	Thread 2
x=1	y=1
g=g+2	g=g*2

The full-blown state-space can be large.

Good news: *the order of independent events does not affect the state that is reached.*

Different orders of independent events constitute an equivalence class (Mazurkiewicz trace equivalence).

It suffices to explore only one representative from each equivalence class.



Partial Order Reduction (POR)

Consider the following thread executions.

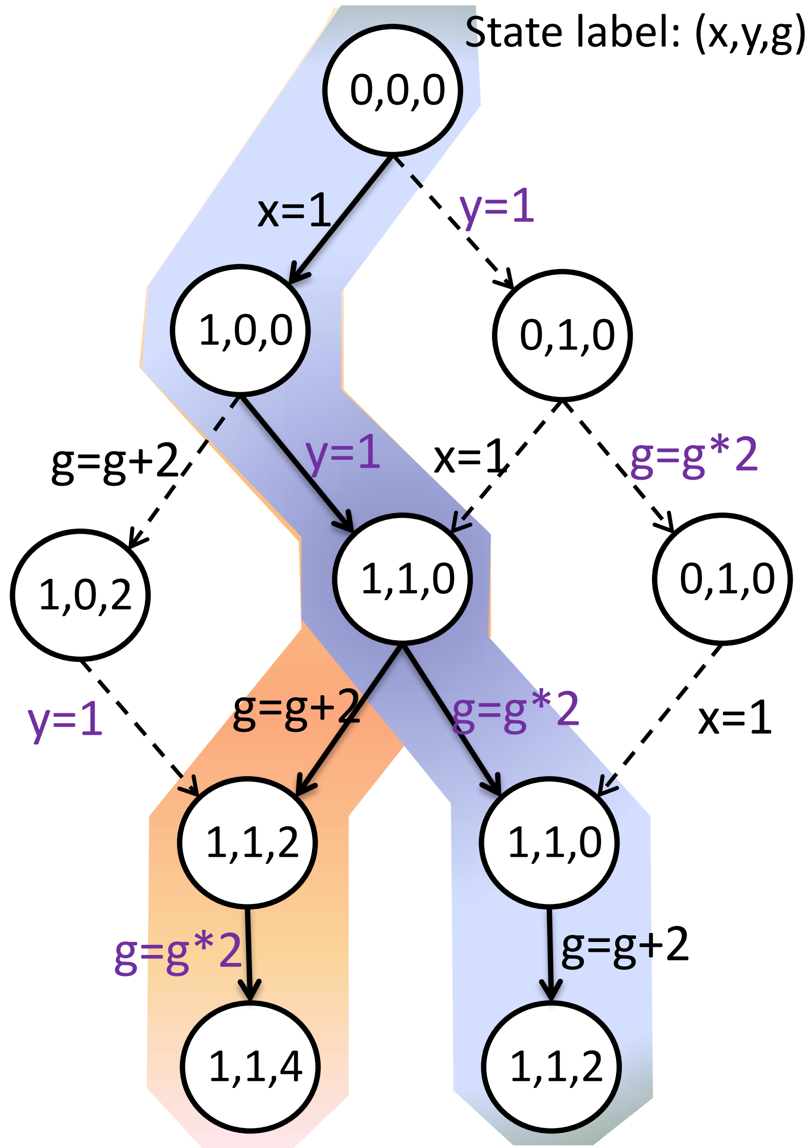
Thread 1	**Thread 2**
x=1	y=1
g=g+2	g=g*2

The full-blown state-space can be large.

Good news: *the order of independent events does not affect the state that is reached.*

Different orders of independent events constitute an equivalence class (Mazurkiewicz trace equivalence).

It suffices to explore only one representative from each equivalence class.



POR in Model Checking

- POR in explicit-state model checking / stateless search
  - Persistent sets, stubborn sets, sleep sets
    - [Godefroid 1996], [Peled 1993], [Valmari 1990], …
  - Dynamic POR (uses HB to derive precise conflict sets), Cartesian POR
    - [Flanagan & Godefroid, POPL 2005], [Gueta et al, SPIN 2007]

- POR in Software Model Checkers
  - **SPIN [Holzmann], VeriSoft [Godefroid], JPF [Visser et al., Stoller et al.]**
    - **Pioneering efforts on model checking concurrent programs**

- POR in symbolic model checking / bounded model checking
  - In BDD based model checking
    - [Alur et al, 2001], [Theobald et al, 2003],…
  - In SAT/SMT based BMC
    - [Cook, Kroening, Sharygina, 2005],
    - [Grumberg, Lerda, Strichman, Theobald, 2005],
    - [Kahlon et al. 2006], [Wang et al. 2008], [Kahlon et al. 2009]

Classic Notion of Independence

- Independence relation [Katz & Peled, 1992] [Godefroid and Pirottin, 1993]

Definition 1 (Independence Relation [14, 8]). *$R \subseteq trans \times trans$ is an independence relation iff for each $\langle t_1, t_2 \rangle \in R$ the following two properties hold for all $s \in S$:*

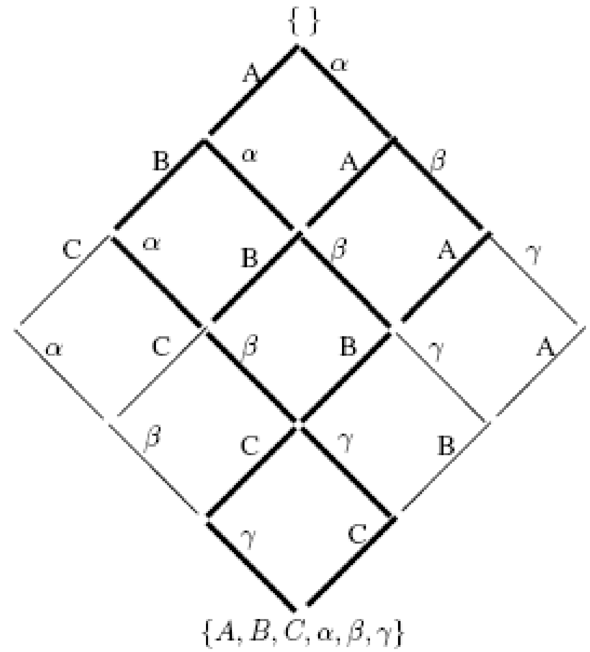
1. *if t_1 is enabled in s and $s \xrightarrow{t_1} s'$, then t_2 is enabled in s iff t_2 is enabled in s'; and*
2. *if t_1, t_2 are enabled in s, there is a unique state s' such that $s \stackrel{t_1 t_2}{\Rightarrow} s'$ and $s \stackrel{t_2 t_1}{\Rightarrow} s'$.*

- Mainly of semantic use (not practical to check)

- Extended to “conditional dependence relation”
  - With respect to “a single state s”, rather than “for all s in S”
  - Well suited for explicit-state algorithms (Adaptive Search), but not for symbolic algorithms

Motivating Example

T_1		T_2	
	i = foo() ;		j = bar() ;
	...		...
A	a[i] = 10 ;	α	a[j] = 50 ;
B	a[i] = a[i]+20;	β	a[j] = a[j]+100;
C	*p = a[j] ;	γ	*q = a[i] ;



Combining classic POR methods with symbolic algorithms is non-trivial

- dependence needs to be defined respect to a set of states (vs. a state)
- need an efficient symbolic encoding

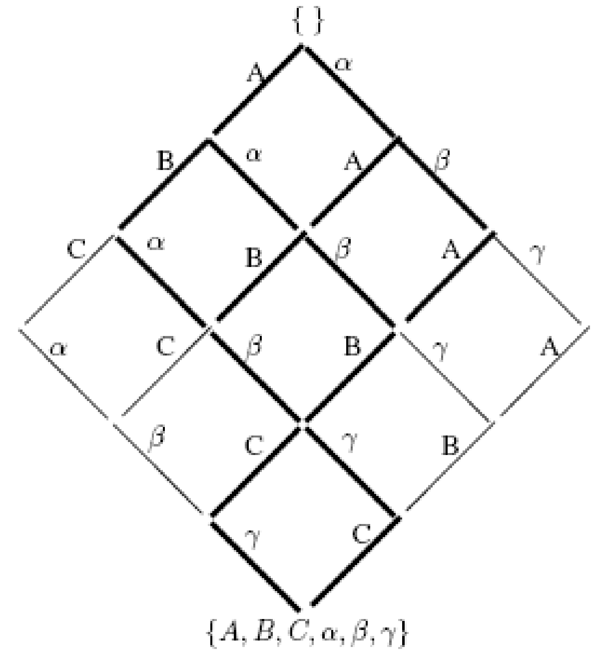
Motivating Example (cont'd)

T_1

	i = foo() ;
	...
A	a[i] = 10 ;
B	a[i] = a[i]+20;
C	*p = a[j] ;

T_2

	j = bar() ;
	...
α	a[j] = 50 ;
β	a[j] = a[j]+100;
γ	*q = a[i] ;



How to exploit this type of PO reductions symbolically?

Guarded Independence Relation

- Independence relation [Katz & Peled, 1992] [Godefroid and Pirottin, 1993]

Definition 1 (Independence Relation [14, 8]). *$R \subseteq trans \times trans$ is an independence relation iff for each $\langle t_1, t_2 \rangle \in R$ the following two properties hold for all $s \in S$:*

1. *if t_1 is enabled in s and $s \xrightarrow{t_1} s'$, then t_2 is enabled in s iff t_2 is enabled in s'; and*
2. *if t_1, t_2 are enabled in s, there is a unique state s' such that $s \stackrel{t_1 t_2}{\Rightarrow} s'$ and $s \stackrel{t_2 t_1}{\Rightarrow} s'$.*

- Guarded by predicates (representing sets of states) [Wang et al. TACAS 08]

Definition 2. *Two transitions t_1, t_2 are guarded independent with respect to a condition c_G iff c_G implies that the following properties hold:*

1. *if t_1 is enabled in s and $s \xrightarrow{t_1} s'$, then t_2 is enabled in s iff t_2 is enabled in s'; and*
2. *if t_1, t_2 are enabled in s, there is a unique state s' such that $s \stackrel{t_1 t_2}{\Rightarrow} s'$ and $s \stackrel{t_2 t_1}{\Rightarrow} s'$.*

Guarded Independence Relation (GIR) for POR

- Notation

For a transition t, we use $V_{RD}(t)$ to denote the set of variables read by t,
$V_{WR}(t)$ to denote the set of variables written by t.

the *potential conflict set* between t_1 and t_2 from different threads

$$\mathcal{C}_{t_1,t_2} = V_{RD}(t_1) \cap V_{WR}(t_2) \cup V_{RD}(t_2) \cap V_{WR}(t_1) \cup V_{WR}(t_1) \cap V_{WR}(t_2)$$

- Collect GIR with a simple traversal of the program structure

1. when $\mathcal{C}_{t_1,t_2} = \emptyset$, add $\langle t_1, t_2, true\rangle$ to R_G;
2. when $\mathcal{C}_{t_1,t_2} = \{a[i], a[j]\}$, add $\langle t_1, t_2, i \neq j\rangle$ to R_G;
3. when $\mathcal{C}_{t_1,t_2} = \{*p_i, *p_j\}$, add $\langle t_1, t_2, p_i \neq p_j\rangle$ to R_G;
4. when $\mathcal{C}_{t_1,t_2} = \{x\}$, consider the following cases:
   a. **RD-WR:** if $x \in V_{RD}(t_1)$ and the assignment $x := e$ appears in t_2, add $\langle t_1, t_2, x = e\rangle$ to R_G;
   b. **WR-WR:** if $x := e_1$ appears in t_1 and $x := e_2$ appears in t_2, add $\langle t_1, t_2, e_1 = e_2\rangle$ to R_G;
   c. **WR-C:** if x appears in the condition $cond$ of a branching statement t_1, such as `if(cond)`, and $x := e$ appears in t_2, add $\langle t_1, t_2, cond = cond[x \rightarrow e]\rangle$ to R_G, in which $cond[x \rightarrow e]$ denotes the replacement of x with e.

Outline

- ✓ **Introduction**
- ✓ **PDS-based Model Checking**
  - ✓ **Theoretical results**
- ✓ **Static Verification**
  - ✓ **Reduction: Partial order reduction**
  - ➢ **Abstraction and Composition: Static analysis, Thread-modular reasoning**
  - – **Bounding: Context-bounded analysis, Memory Consistency-based analysis**
- ❑ **Dynamic Verification**
  - – **Preemptive Context Bounding**
  - – **Predictive Analysis**
  - – **Active Testing**
  - – **Coverage-guided Systematic Testing**
- ❑ **Summary & Challenges**

Motivating Example for Static Analysis

```
void Alloc_Page ( ) {
 a = c;
 pt_lock(&plk);
 if (pg_count  >= LIMIT) {
    pt_wait (&pg_lim, &plk);
    incr (pg_count);
    pt_unlock(&plk);
    sh1 = sh;
 } else {
    pt_lock (&count_lock);
    pt_unlock (&plk);
    page = alloc_page();
    sh = 5;
    if (page)
       incr (pg_count);
    pt_unlock(&count_lock);
  end-if
  b = a+1;
}
```

```
void Dealloc_Page ( )
 pt_lock(&plk);
 if (pg_count  == LIMIT) {
    sh = 2;
    decr (pg_count);
    sh1 = sh;
    pt_notify (&pg_lim, &plk);
    pt_unlock(&plk);
 } else {
    pt_lock (&count_lock);
    pt_unlock (&plk);
    decr (pg_count);
    sh = 4;
    pt_unlock(&count_lock);
  end-if
}
```

Consider all possible pairs of locations where shared variables are accessed (e.g. for checking data races)

Motivating Example: Lockset Analysis

```
void Alloc_Page ( ) {
 a = c;
 pt_lock(&plk);
 if (pg_count  >= LIMIT) {
    pt_wait (&pg_lim, &plk);
    incr (pg_count);
    pt_unlock(&plk);
    sh1 = sh;
 } else {
    pt_lock (&count_lock);
    pt_unlock (&plk);
    page = alloc_page();
    sh = 5;
    if (page)
       incr (pg_count);
    pt_unlock(&count_lock);
  end-if
  b = a+1;
}
```

```
void Dealloc_Page ( )
  pt_lock(&plk);
  if (pg_count  == LIMIT) {
     sh = 2;
     decr (pg_count);
     sh1 = sh;
     pt_notify (&pg_lim, &plk);
     pt_unlock(&plk);
  } else {
     pt_lock (&count_lock);
     pt_unlock (&plk);
     decr (pg_count);
     sh = 4;
     pt_unlock(&count_lock);
   end-if
 }
```

Lockset Analysis: Compute the set of locks at location *l*
Here, lock *plk* is held in both locations.
Hence, these locations are simultaneously unreachable.
Therefore, there is no datarace.

Motivating Example: Synchronization Constraints

```
void Alloc_Page ( ) {
  a = c;
  pt_lock(&plk);
  if (pg_count  >= LIMIT) {
     pt_wait (&pg_lim, &plk);
     incr (pg_count);
     pt_unlock(&plk);
     sh1 = sh;
  } else {
     pt_lock (&count_lock);
     pt_unlock (&plk);
     page = alloc_page();
     sh = 5;
     if (page)
        incr (pg_count);
     pt_unlock(&count_lock);
   end-if
   b = a+1;
}
```

```
void Dealloc_Page ( )
  pt_lock(&plk);
  if (pg_count  == LIMIT) {
     sh = 2;
     decr (pg_count);
     sh1 = sh;
     pt_notify (&pg_lim, &plk);
     pt_unlock(&plk);
  } else {
     pt_lock (&count_lock);
     pt_unlock (&plk);
     decr (pg_count);
     sh = 4;
     pt_unlock(&count_lock);
   end-if
}
```

These locations are simultaneously unreachable due to wait-notify ordering constraint. Therefore, no datarace.

Motivating Example

```
void Alloc_Page ( ) {
  a = c;
  pt_lock(&plk);
  if (pg_count  >= LIMIT) {
     pt_wait (&pg_lim, &plk);
     incr (pg_count);
     pt_unlock(&plk);
     sh1 = sh;
  } else {
     pt_lock (&count_lock);
     pt_unlock (&plk);
     page = alloc_page();
     sh = 5;
     if (page)
        incr (pg_count);
     pt_unlock(&count_lock);
   end-if
   b = a+1;
}
```

```
void Dealloc_Page ( )
  pt_lock(&plk);
  if (pg_count  == LIMIT) {
     sh = 2;
     decr (pg_count);
     sh1 = sh;
     pt_notify (&pg_lim, &plk);
     pt_unlock(&plk);
  } else {
     pt_lock (&count_lock);
     pt_unlock (&plk);
     decr (pg_count);
     sh = 4;
     pt_unlock(&count_lock);
   end-if
```

Data race?

How do we get these invariants?
By using abstract interpretation, model checking, …

NO, due to invariants at these locations
pg_count is in (-inf, LIMIT) in T1
pg_count is in [LIMIT, +inf) in T2
Therefore, these locations are not simultaneously reachable

Symbolic Verification of Programs

❑ Abstract Interpretation

[Cousot & Cousot 77]

– **State sets are not exact, but over-approximations (for sound analysis)**

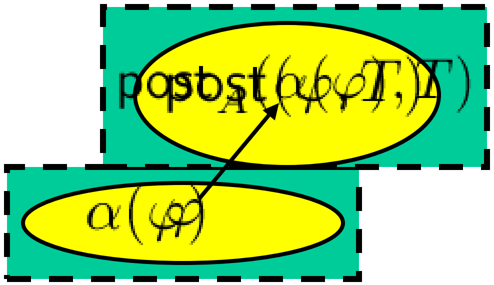
– **Abstract post operation**

$$\mathsf{post}_A(\psi, T) : \boxed{(\exists X_0)}\, (\psi[X_0] \wedge T[X_0, X])$$

Abstract post

Abstract description

Set of states



– **Over-approximate fixpoint computation**

$$\psi_{i+1} : \psi_i \boxed{\sqcup} \mathsf{post}_A(\psi_i, T))$$

$$\psi_{i+1} \boxed{\sqsubseteq} \psi_i$$

Join operation

❑ Popular for generating inductive invariants for Sequential Programs

– **Abstract domains: intervals, octagons, polyhedra, …**

Concurrent Programs: Static Analysis

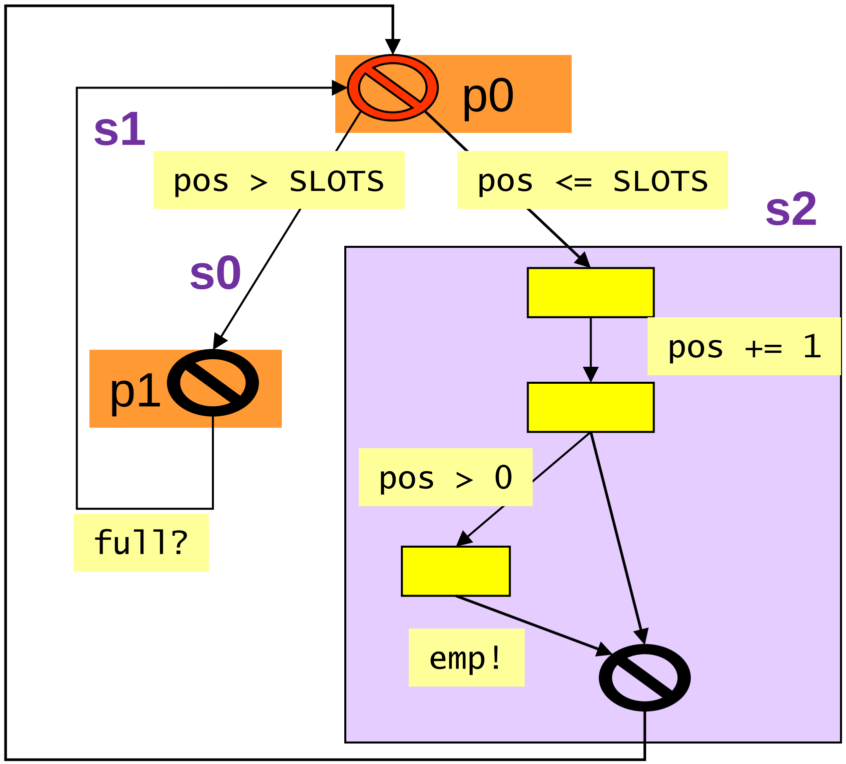
- **Intuitively, one can reason similarly for concurrent programs**
  - **Not all product (global) control states, but only the *statically reachable* states**
  - **Transaction Graph:**
    - **Each node is a *statically reachable* global control state,**
    - **Each edge is a *transaction*, i.e. an uninterruptible sequence of actions by a single thread**

- **Two main (inter-related) problems**
  - **How to find which global control states (nodes) are reachable?**
  - **How to find (large) transactions?**
    - **Larger the transactions, smaller the number of interleavings to consider**

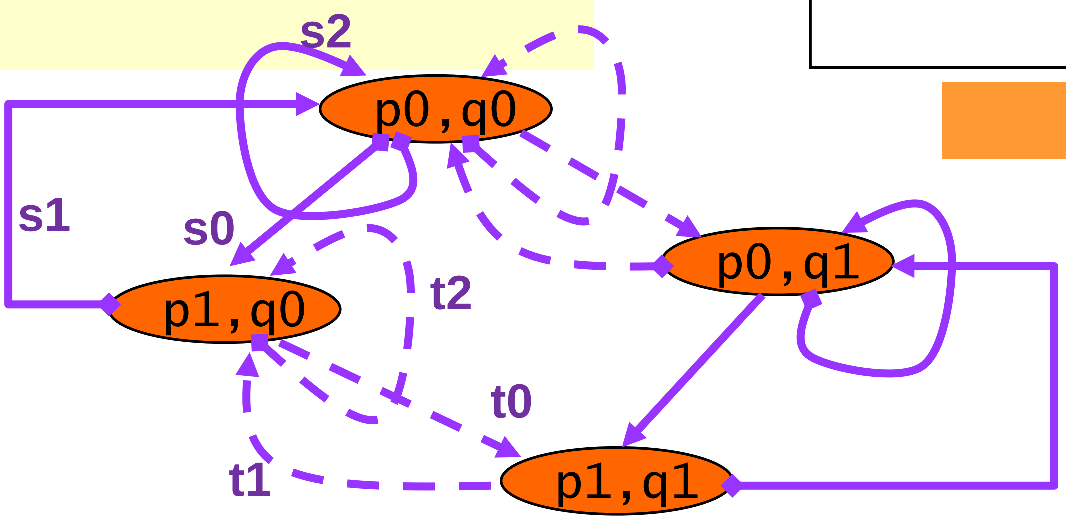
- **Refinement Approach** [Kahlon *et al.* TACAS 09]
  - **At any stage, the transaction graph *over-approximates* the set of thread interleavings for sound static analysis or model checking**
  - ***Iteratively refine* the transaction graph by computing *invariants***

Transaction Graph Example

```
repeat (forever){
  lock(posLock);
  while ( pos > SLOTS){
      unlock(posLock);
      wait(full);
      lock(posLock);
  }
  data[pos++] := ...;
  if (pos > 0){
     signal(emp);
  }
   unlock(posLock);
}
```



Nodes where context switches to be considered



Refining Transactions

[Kahlon *et al.* TACAS 09]

- **Initial Transaction Graph**
  - **Make this as small as possible**
  - **Use static partial order reduction (POR) to consider non-redundant interleavings**
    - **Over control states only, but need to consider CFL-reachability**
  - **Use synchronization constraints to eliminate statically unreachable nodes**
    - **Recall: Static reachability wrt synchronization operations**
    - Precise analysis for nested locks, bounded lock chains, locks with wait-notify

      [Kahlon *et al.* 05, Kahlon 08, Kahlon & Wang 10]

- **Iterative Refinement of Transaction Graph**

  Repeat
  - **Compute invariants over the transaction graph using abstract interpretation**
    - Abstract domains: range, octagons, polyhedra [Cousot & Cousot, Miné. …]
  - **Use invariants to prove nodes unreachable, and simplify graph**
  - **Re-compute transactions (POR, synchronization analysis)**

  Until **transactions cannot be refined further.**

Application: Detection of Data Races

- **Implemented in NEC's CoBe (Concurrency Bench) tool**

- **Phase 1: Static Warning Generation**
  - **Shared variable detection, Lockset analysis**
  - **Generate warnings at global control states (c1, c2) when**
    - The same shared variable is accessed, at least one access is a write, and
    - Locksets at c1 and c2 are disjoint

- **Phase 2: Static Warning Reduction (for improved precision)**
  - **Create a Transaction Graph, and generate sound invariants**
    - POR reductions, synchronization analysis, abstract interpretation
  - **If (c1, c2) is proved unreachable, then eliminate the warning**

- **Phase 3: Model Checking**
  - **Otherwise, create a model for model checking reachability of (c1, c2)**
    - Slicing, constant propagation, enforcing invariants: lead to smaller models
    - Makes bounded model checking viable
    - Provides a concrete error trace

CoBe: Experiments

- Linux device drivers with known data race bugs

Linux Driver	KLOC	#Sh Vars	#Warnings	Time (sec)	# After Invariants	Time (sec)	#Witness MC	#Unknown
pci_gart	0.6	1	1	1	1	4	0	1
jfs_dmap	0.9	6	13	2	1	52	1	0
hugetlb	1.2	5	1	4	1	1	1	0
ctrace	1.4	19	58	7	3	143	3	0
autofs_expire	8.3	7	3	6	2	12	2	0
ptrace	15.4	3	1	15	1	2	1	0
raid	17.2	6	13	2	6	75	6	0
tty_io	17.8	1	3	4	3	11	3	0
ipoib_multicast	26.1	10	6	7	6	16	4	2
TOTAL			99		24		21	3
decoder	2.9	4	256	5min	15	22min		
bzip2smp	6.4	25	15	18	12	35		

After Phase 1 (Warning Generation)

After Phase 2 (Warning Reduction)

After Phase 3 (Model Checking)

CoBe: Experiments

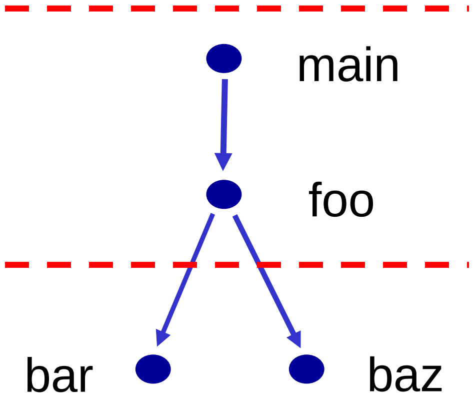
- Linux device drivers with known data race bugs

Linux Driver	KLOC	#Sh Vars	#Warnings	Time (sec)	# After Invariants	Time (sec)	#Witness MC	#Unknown
pci_gart	0.6	1	1	1	1	4	0	1
jfs_dmap	0.9	6	13	2	1	52	1	0
hugetlb	1.2	5	1	4	1	1	1	0
ctrace	1.4	19	58	7	3	143	3	0
autofs_expire	8.3	7	3	6	2	12	2	0
ptrace	15.4	3	1	15	1	2	1	0
raid	17.2	6	13	2	6	75	6	0
tty_io	17.8	1	3	4	3	11	3	0
ipoib_multicast	26.1	10	6	7	6	16	4	2
TOTAL			99		24		21	3
decoder	2.9	4	256	5min	15	22min		
bzip2smp	6.4	25	15	18	12	35		

- Successfully applied to medium-sized Linux device drivers
- How about scalability on industry projects?
  - On large code (> 100 kLOC – 1 MLOC), could not create CFG for entry function

CoBe: Layered Analysis

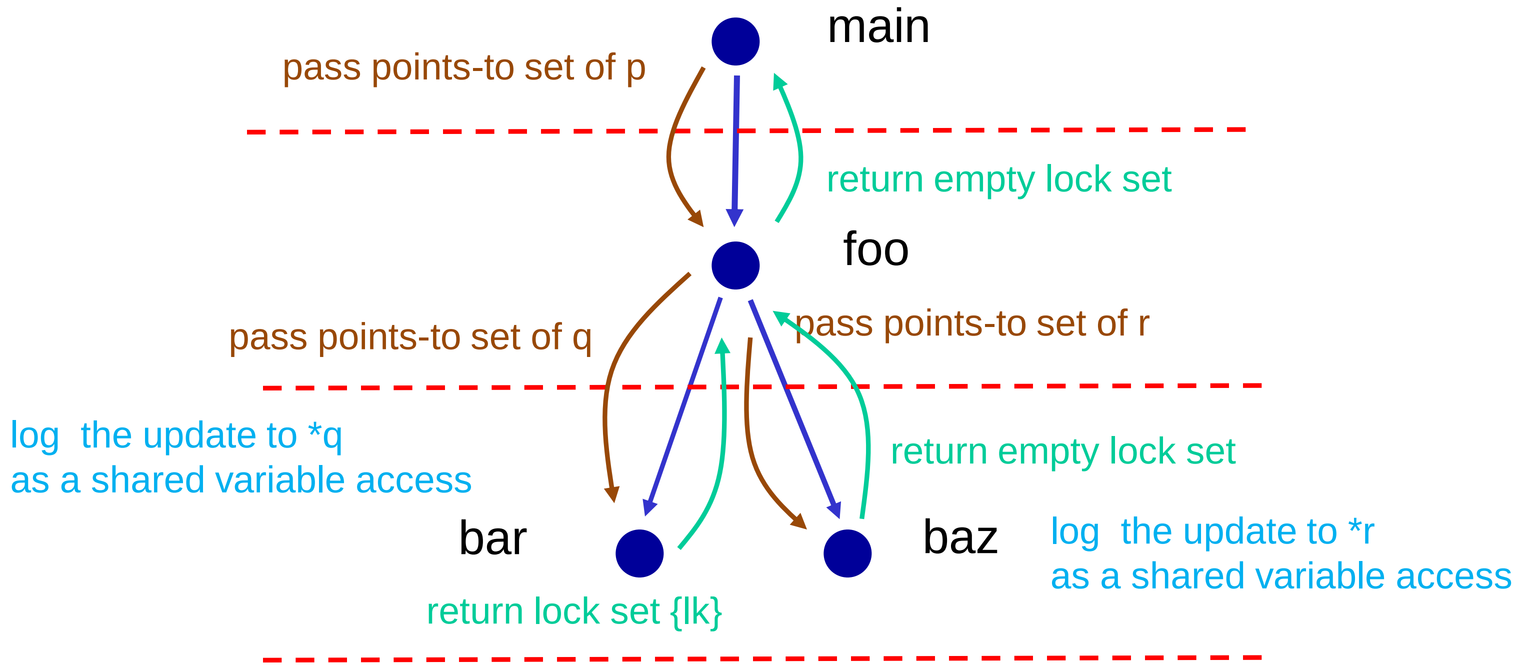
- **Issue**
  - For threads executing functions with large CFGs (control flow graphs), the CFG construction itself may run out of memory
- **Strategy**
  - Trade-off time for space, via Call Graph layering
  - Work with few layers in memory at a time, using files to transfer information between layers



- **Implementation**
  - First the CFG of the entry function in each thread is created up to some small depth cutoff (DC), e.g. DC = 2 includes main and foo above
  - The CFGs of functions called at depth greater than DC (e.g. bar, baz) are built on-the-fly, in depth-first order according to call graph of entry function (main)
  - Aliasing and lockset information is passed across layers.

CoBe: Layering Example

```
int sh;

main(){            foo(int *p){        bar(int *q){      baz(int *r){
  foo(&sh);          bar(p);             lock(lk);         *r = 0;
}                    unlock(lk);         *q = 1;         }
                     baz(p);           }
                   }
```



Thread-Modular Reasoning

- As we just saw, invariants play a key role in static analysis
- **Compositional verification**
  - ***Proofs rules*** **typically use** ***inductive invariants***
  - **Advantage: Avoids explicit reasoning over interleavings**

- **Some Basics**
  - **An** ***assertion*** **is a set of states**
  - **Assertion φ is** ***invariant*** **if it includes all reachable states**
  - **Invariance is proved using an auxiliary** ***inductive*** **invariant θ**
    - **(initiality)** $[\, I \Rightarrow \theta \,]$
    - **(inductiveness)** $[\, next\,(T, \theta) \Rightarrow \theta \,]$
    - **(adequacy)** $[\, \theta \Rightarrow \varphi \,]$
  - $next(T, \psi)$ **is the set of successors of states in ψ by T**
  - R **is the strongest inductive invariant**
    - **but may not need strongest**

Localized Inductive Invariants

Idea: build an inductive invariant out of "little" pieces

- Restrict θ to the shape $\theta_1(X, L_1) \wedge \theta_2(X, L_2) \wedge \ldots \wedge \theta_N(X, L_N)$
- X is the set of (globally) shared program variables (e.g., locks)
- L_i is the set of variables local to process P_i (e.g., program counter, stack, temporary variables)
- The shape inherently limits correlations between local variables of different components (e.g., $(x > l_1)$ is OK but not $(l_1 + l_2 > l_3)$)

Localized Inductive Invariants = Compositional Proof

Inductiveness for a localized assertion turns into the rely-guarantee form

- (initiality) For all i: $[I_i \Rightarrow \theta_i]$
- (inductiveness) For all i: $[\mathrm{next}(T_i, \theta_i) \Rightarrow \theta_i]$
- (non-interference) For all $i, j : j \neq i$:
  $[\mathrm{next}(\mathit{intf}_j^\theta \wedge \mathit{unchanged}(L_i), \theta_i) \Rightarrow \theta_i]$
- The effect of process P_j on the shared state is called *interference*, represented by $\mathit{intf}_j^\theta(X, X') = (\exists L_j, L_j' : T_j \wedge \theta_j)$

(We'll call a localized inductive invariant a "split invariant".)

Computing the Strongest Split Invariant

The Knaster-Tarski Theorem also gives a simple iterative scheme to compute the fixpoint.

1. Set the initial vector $\theta^0 = (\mathit{false}, \mathit{false}, \ldots, \mathit{false})$
2. At stage i, compute $\theta^{i+1} = F(\theta^i)$
3. Stop when a fixpoint is reached (no change in any component)

Theorem: Complexity

This algorithm takes time polynomial in N and in the size L (the number of states) of each component. The complexity is (roughly) $O(N^3L^3)$.

Local Proofs [Cohen & Namjoshi CAV 07, CAV 08, CAV 10]
Can handle safety and liveness properties
Works well on many examples (Bakery, Peterson's, Szymanski, …)

[Gupta *et al.* POPL 11, CAV 11]

- Automation and experimental comparison of "Owicki-Gries" and "Rely-Guarantee" reasoning
- Applicable to arbitrary (ad-hoc) synchronization patterns (not only nested locking or datarace-free code)
- Analyze implicitly an unbounded number of context switches (not restricted to context-bounded switching)
- Handles non-thread-modular proofs (not restricted to thread-modular, global-only, assumptions)

`http://www.model.in.tum.de/~popeea/research/threader.html`

Uses well-known techniques from software model checking (predicate abstraction refinement, CEGAR) for automating the proof rules

Outline

- ✓ Introduction

- ✓ PDS-based Model Checking
  - ✓ Theoretical results

- ✓ Static Verification
  - ✓ Reduction: Partial order reduction
  - ✓ Abstraction and Composition: Static analysis, Thread-modular reasoning
  - ➢ Bounding: Context-bounded analysis, Memory Consistency-based analysis

- ❑ Dynamic Verification
  - Preemptive Context Bounding
  - Predictive Analysis
  - Active Testing
  - Coverage-guided Systematic Testing

- ❑ Summary & Challenges

Context-Bounded Analysis

- **Recall**
  - **The general problem of verifying a concurrent program (recursive procedures with synchronization) is undecidable.**
  - **We have seen various strategies to get around undecidability**
    - Exploiting patterns of synchronization
    - Restricting synchronization & communication
    - Ignoring recursion by (bounded) function inlining

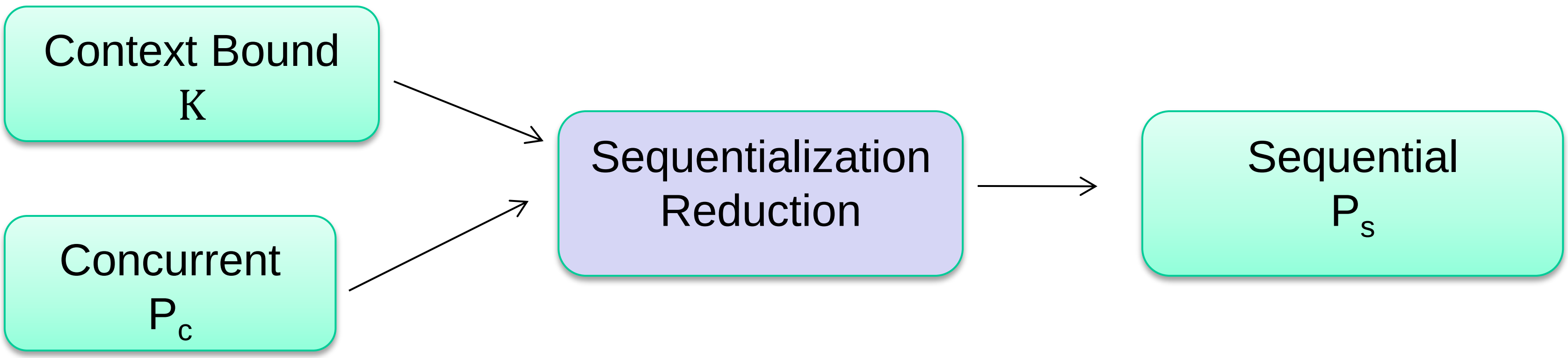
- **Another key idea: Bound number of context switches**
  - **Context-bounded analysis of PDSs is decidable [Qadeer & Rehof, TACAS 05]**
  - **Note: There can be recursion within each segment between context switches**

  - **In practice, many bugs are found within a small number of context switches**
  - **Implemented in tools: KISS, CHESS (Microsoft), …**

Context-Bounded Analysis using Sequentialization

[Lal & Reps, CAV 08]

❑ **Sequentialization: Reduce CBA to sequential program analysis**



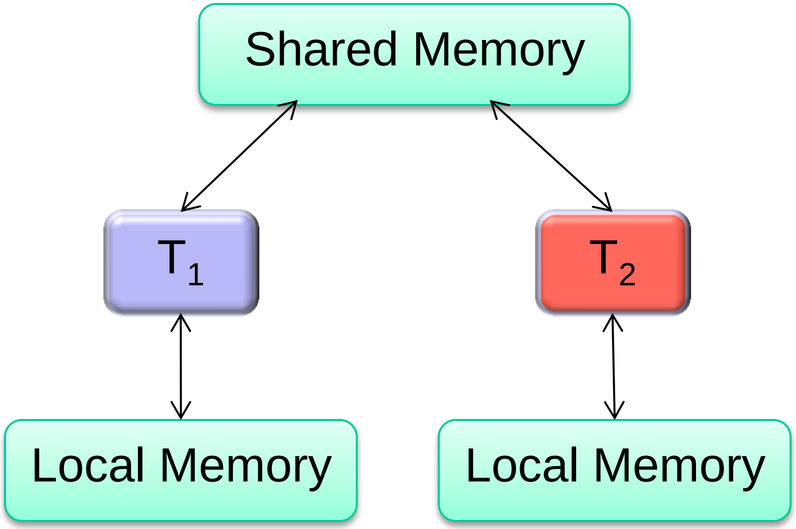
- Efficient reduction:
  - P_S has K times more global variables
  - No increase in local variables

- Can borrow all the cool stuff from the sequential world

From Concurrent to Sequential

❑ Model

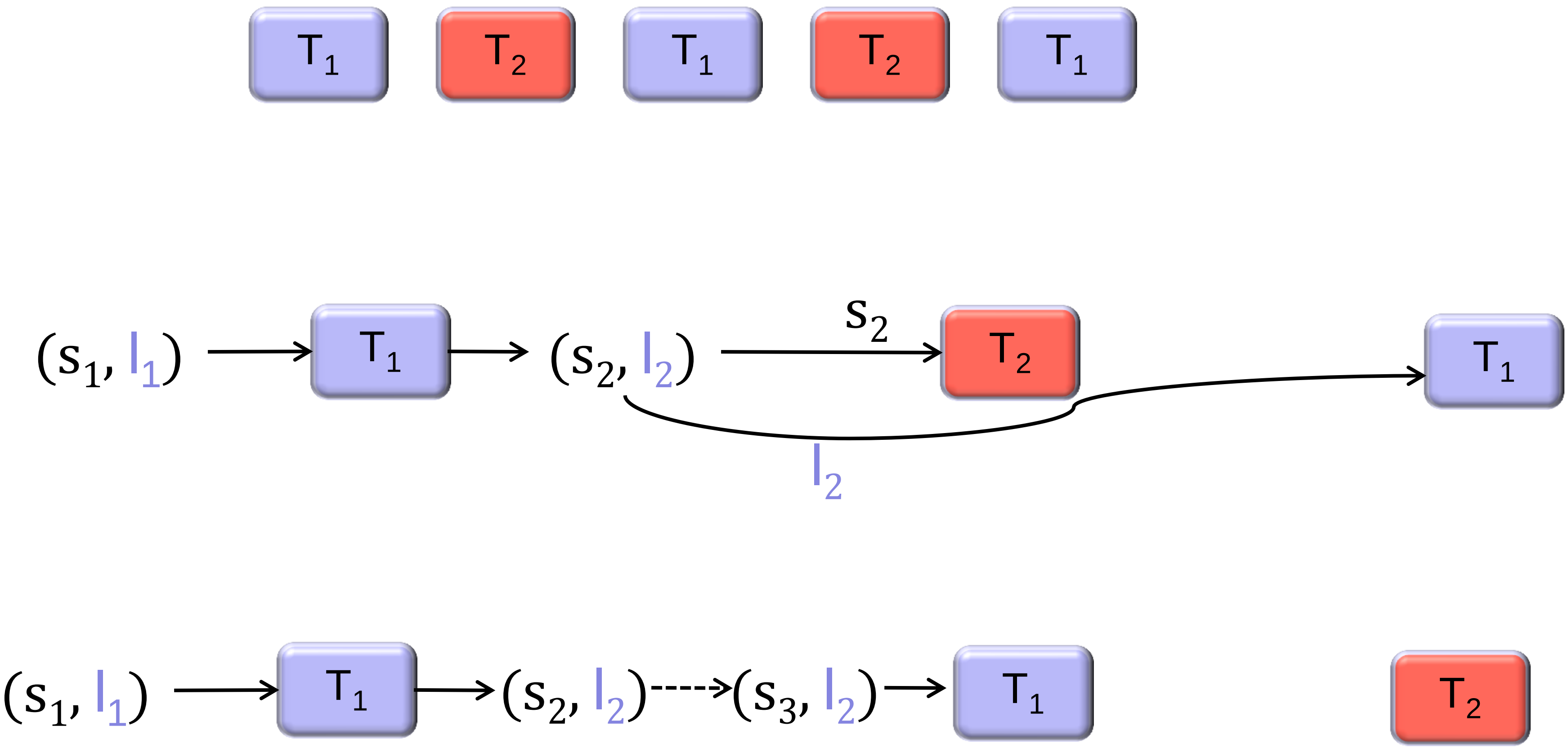
Two threads, shared memory, K execution contexts per thread



$$(s_1, l_1) \longrightarrow \boxed{T_i} \longrightarrow (s_2, l_2)$$

From Concurrent to Sequential

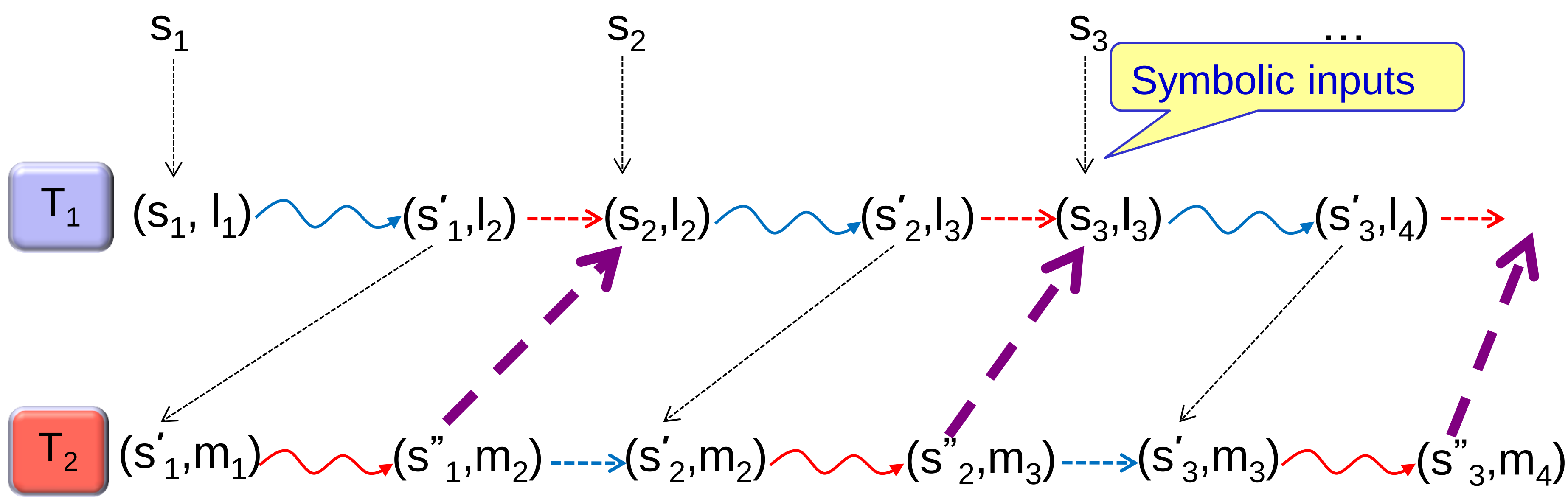
Execution proceeds as:



__Guess__ the effect of T_2 *__Verify__* the guess

Sequentialization: Idea

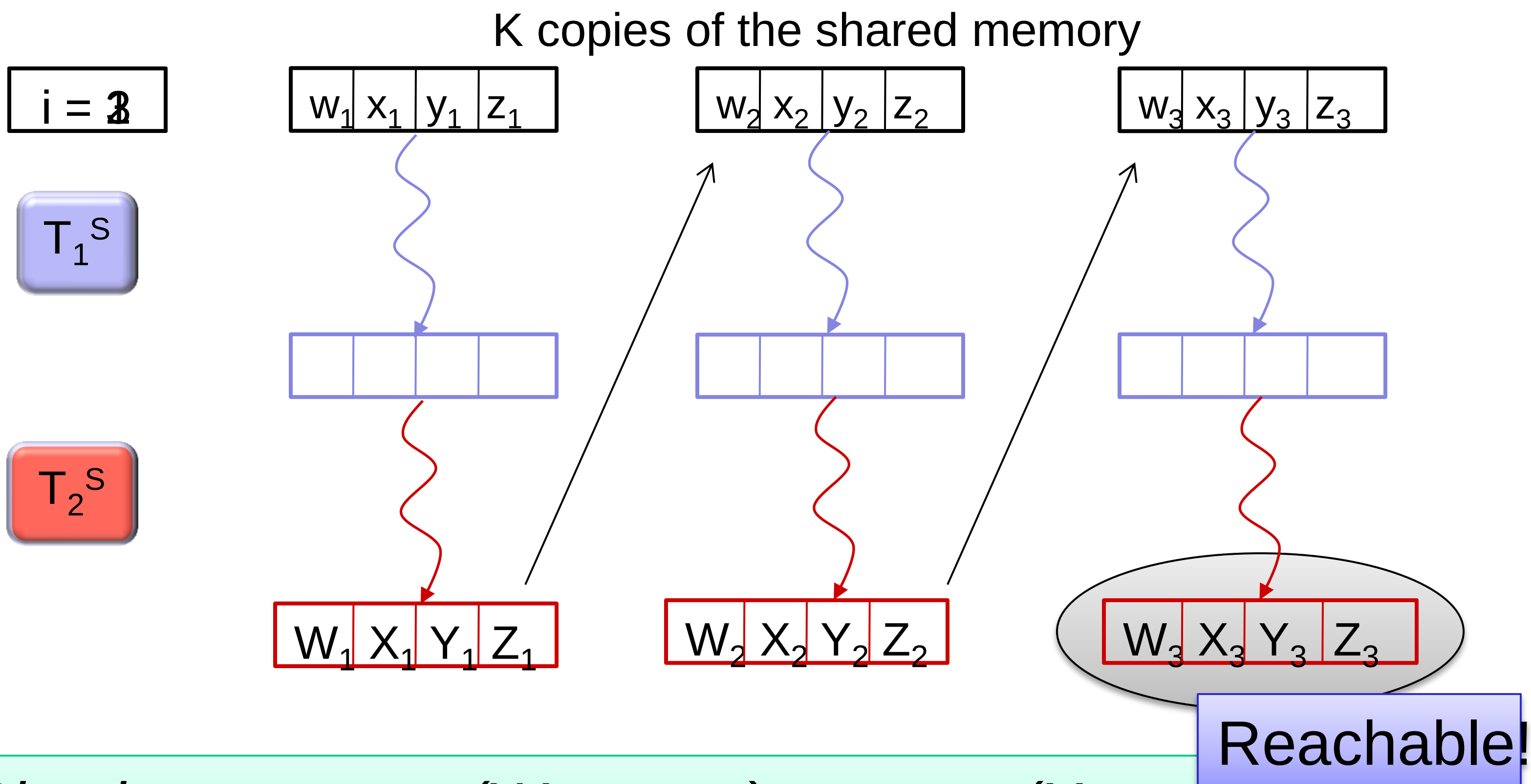
- K = number of chances that each thread gets
- Guess (K-1) global states: s_1 = init, s_2, …, s_K



T_1 processes all contexts first, guesses states of T_2
T_2 goes next, using states of T_1
At the end: Check the guesses, i.e. $s''_1 = s_2$ and $s''_2 = s_3$, …

Sequentialization Transformation

- $T_1 \longrightarrow T_1^s$ and $T_2 \longrightarrow T_2^s$
- $(T_1 \,||\, T_2) \longrightarrow (T_1^s;\; T_2^s\,;$ Checker; assert(no_error))



Checker: assume($W_1 == w_2$); assume($X_1 == x_2$); …
assume($Y_2 == y_3$); assume($Z_2 == z_3$)

Sequentialization: Summary

- Pushes “guesses” about interleaved states into inputs

- $T_1 \longrightarrow T_1^s$ and $T_2 \longrightarrow T_2^s$
- $(T_1 \,||\, T_2) \longrightarrow (T_1^s;\ T_2^s$; Checker; assert(no_error))

Main idea:
Reduce *control* non-determinism to *data* non-determinism

Memory Consistency-based Analysis

- Interleaving model
  - Partially ordered traces
  - Context-switching, interleaved traces
  - Is control-centric: Control induces data-flow

- Instead, consider a Memory Consistency (MC) model
  - e.g. Sequential Consistency (SC), Total Store Order (TSO), ….
  - MC model specifies rules under which a read may observe some write

- Data Nondeterminism in MC model
  - Reason about read-write interference directly
  - No need to have a scheduler
  - Is data-centric : data-flow induces control-flow
  - Examples: Nemos, Checkfence, x86-TSO, Memsat, Staged Analysis
  - Symbolic exploration using SAT/SMT solvers avoids explicit enumeration of interleavings

Sequential Consistency (SC) based Verification

[Sinha & Wang POPL 11]

Bounded

- Three steps
  1. Obtain an Interference Skeleton (IS) from (unrolled) Program
     - Global read and write events and their program order
     - Encoded as Φ_{IS}
  2. SC axioms for reads/writes in IS
     - Quantified first-order logic formula Π
  3. Encode Property as a formula Φ_P
     - data race, assertion violation, …

- Check $\Phi_{IS} \wedge \Pi \wedge \Phi_P$ for satisfiability (using an SMT solver)

Sequential Consistency Axioms

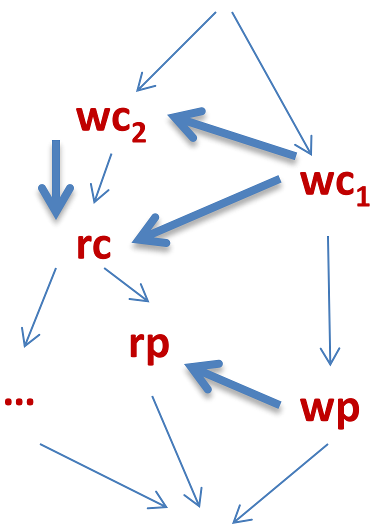
- Axioms of Sequential Consistency (SC)
  - each read must observe (**link with**) some write
  - read must link with most recent write in execution order

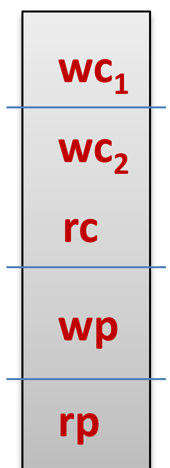
- Specified in typed first-order logic
  - read **r**, write **w**: Access type
- **Link** Predicate: **link (r,w)**
  - holds if read **r** observes write **w** in an execution
  - Exclusive : link (r,w) => ∀ w'. ¬ link (r,w')
- **Must-Happen-before** Predicate : **hb (w,r)**
  - **w** must happen before **r** in the execution
  - strict partial order

- These axioms are added to the Program precisely encoded using reads/writes and program order

Example

	Thread 1	Thread 2	
wc_2	c = true;	c = false;	wc_1
rc	if (c) {	…….	
rp	*p = 0;	p = 0;	wp
	}		
	else …		





Goal: Detect NULL pointer access violation

- so **rp** must be enabled
- en (rp) = (en (rc) $\wedge$ val(rc) = true)

en(rp) $\Rightarrow$ en (rc) (Path conditions)

and, en(rp) $\Rightarrow$ val (rc) = true (*)

Because en(rp), so **link(rp,wp)** (Π)

So, hb (wp,rp) (Π)

link (rc, wc_1) $\vee$ link (rc, wc_2) (Π)

Try link (rc, wc_1)

so, val (rc) = val(wc_1) = false (Π)

Contradicts with (*)

so, **link (rc, wc_2)**

so, hb (wc_2, rc) (Π)

Check (Π) for **rc**: intruding write **wc_1**

so, Add **hb(wc_1, wc_2)**

linearize to obtain a feasible trace

Outline

✓ **Introduction**

✓

✓

PDS-based model checking, Static Verification
- **May not scale to large programs**
- **Too many false warnings**
- **Difficult to apply in multi-process or distributed settings**

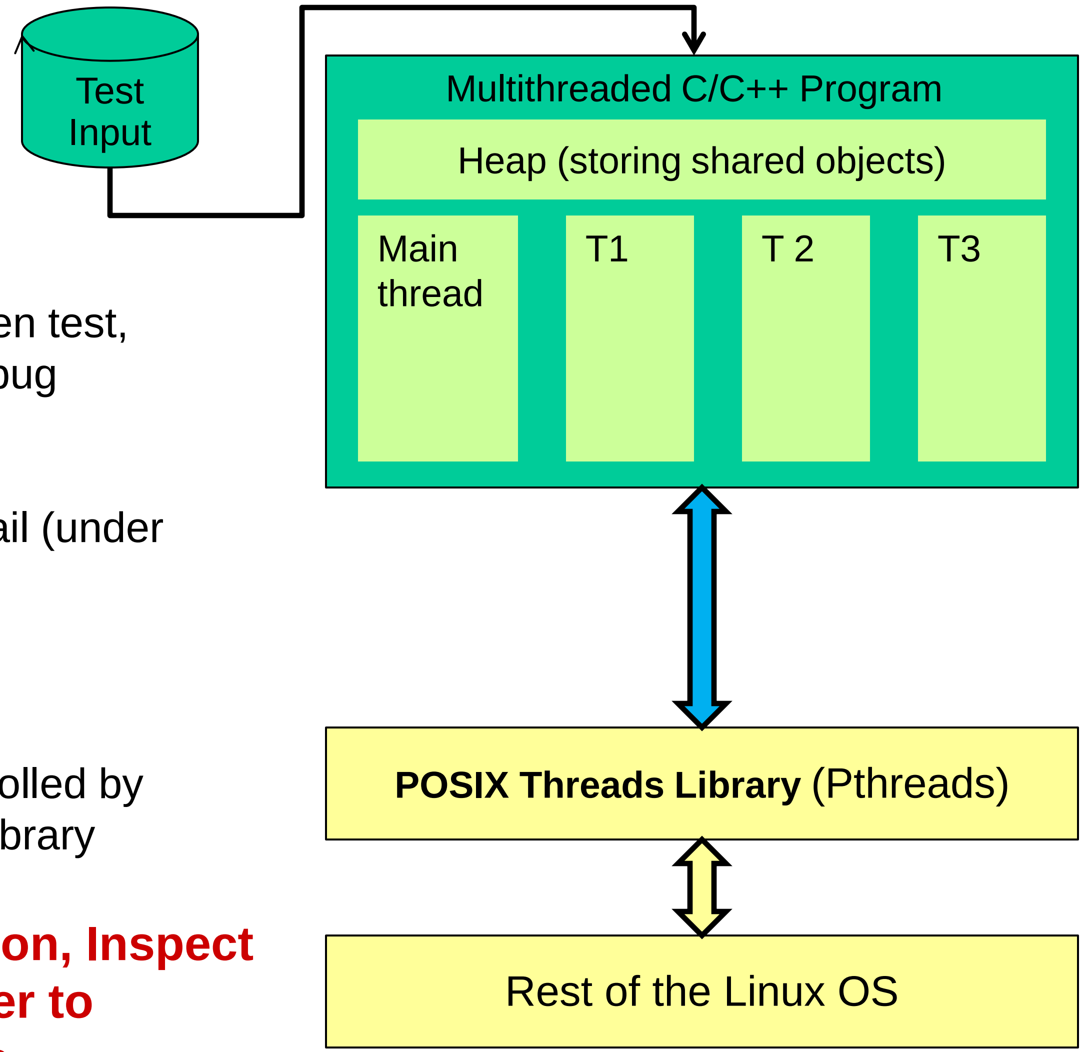
Interest in Dynamic Verification based on executions

- **Bounding: Context bounded analysis , Memory Consistency based analysis**

➢ **Dynamic Verification**
- **Preemptive Context Bounding**
- **Predictive Analysis**
- **Active Testing**
- **Coverage-guided Systematic Testing**

❑ **Summary & Challenges**

Testing Multi-threaded Programs



User expectation:

If the program fails the given test, the user wants to see the bug

The reality:

Even if the program may fail (under a certain schedule),
the user likely won't see it

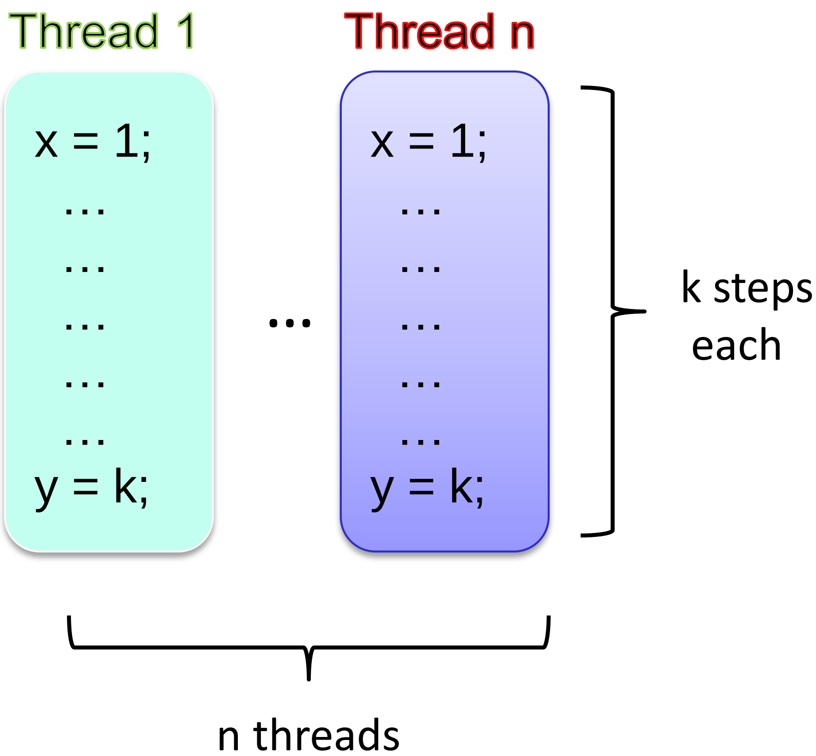
Why?

Thread scheduling is controlled by the OS and the Pthreads library

Tools: VeriSoft, Chess, Fusion, Inspect
Take control of the scheduler to execute alternate schedules

CHESS: Heisenbugs and State space explosion

[Musuvathi *et al.* PLDI 07, OSDI 08]



- **Number of executions = $O(n^{nk})$**
- **Exponential in both n and k**
  - **Typically: n < 10, k > 100**
- **Limits scalability to large programs**

Goal: Scale CHESS to large programs (large k)

CHESS: Preemptive Context Bounding (PCB)

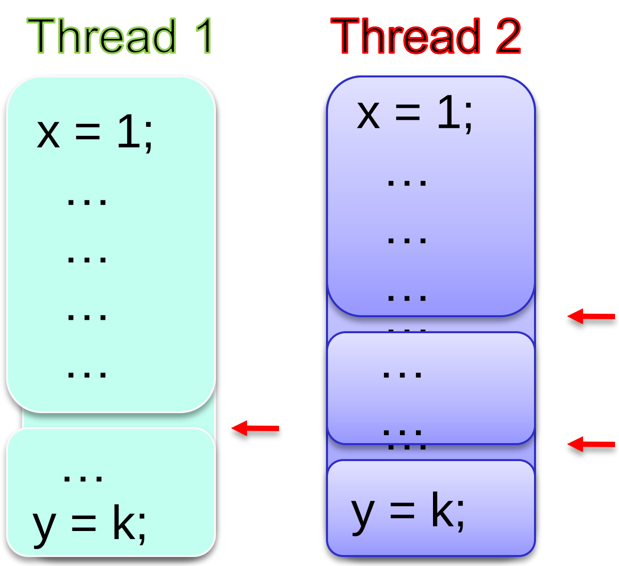
[Musuvathi *et al.* PLDI 07, OSDI 08]

- **Terminating program with fixed inputs and deterministic threads**
  - **n threads, k steps each, c preemptions**
  - **Preemptions are context switches forced by the scheduler**

- **Number of executions <= $_{nk}C_c \,.\, (n+c)!$**

  $= O((n^2k)^c .\, n!)$

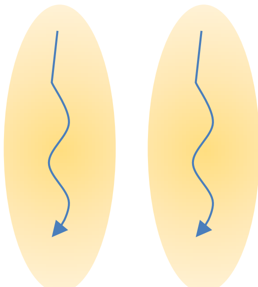
Exponential in n and c, but not in k



- **Choose c preemption points**
- **Permute n+c atomic blocks**

Many bugs found in a small number of preemptions

Trace Based Verification



Concurrent program

Formal

Verification

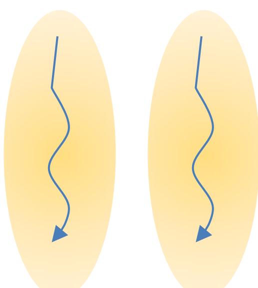
e.g. model checking

Large state-space

Full formal verification is often intractable

Will not talk about this

<u>Alternate approaches</u>



Concurrent program

Collect shared

access footprint

Trace

Online/offline monitoring of trace

Monitoring problem

Tractable and no false alarms.

Predict errors in alternate interleavings

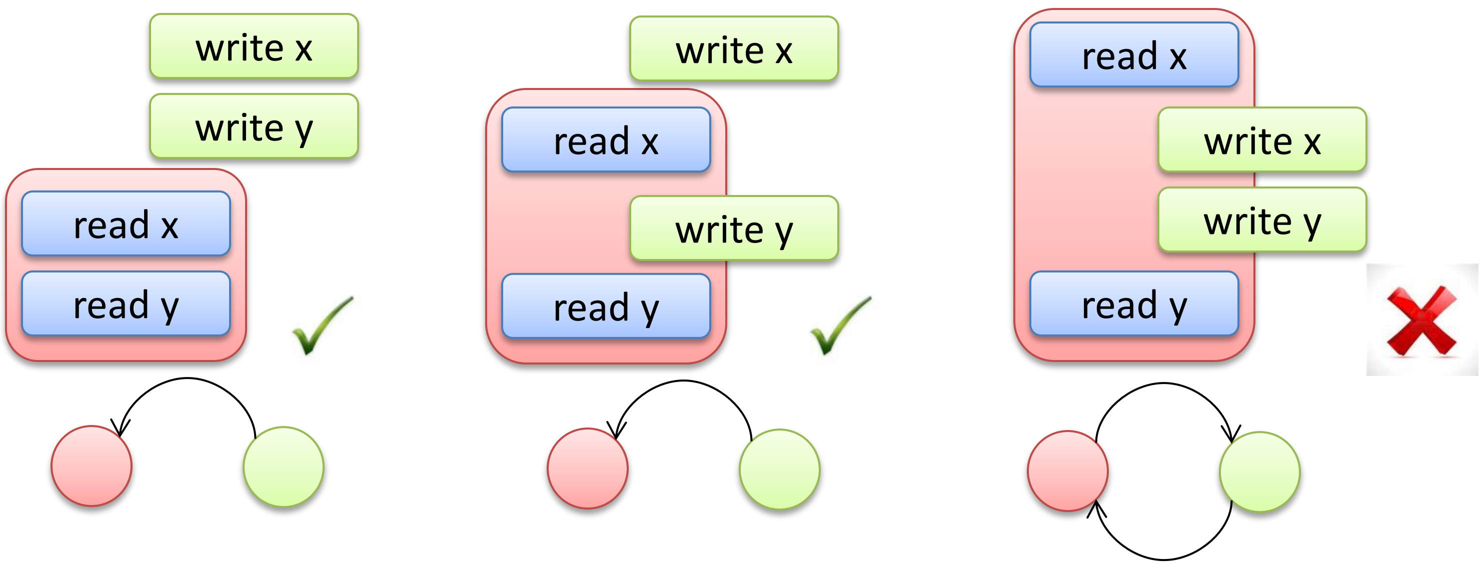
Next

Predictive Analysis problem

Larger set of interleavings is explored.

Recall: Atomicity Violations

- Atomicity is a desired correctness criterion for concurrent programs.
  - Non-interference on shared accesses from code residing outside and inside an atomic region.
  - Serializability is a notion that checks atomicity.



- A recent study shows 69% of concurrency bugs due to atomicity violations [Lu et al. ASPLOS'08]

Predictive Analysis: Motivating Example

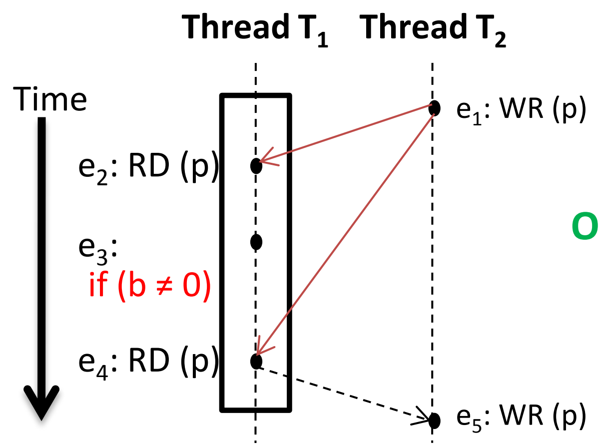
We are checking for potential serializability violations.

Thread T_1 | **Thread T_2**

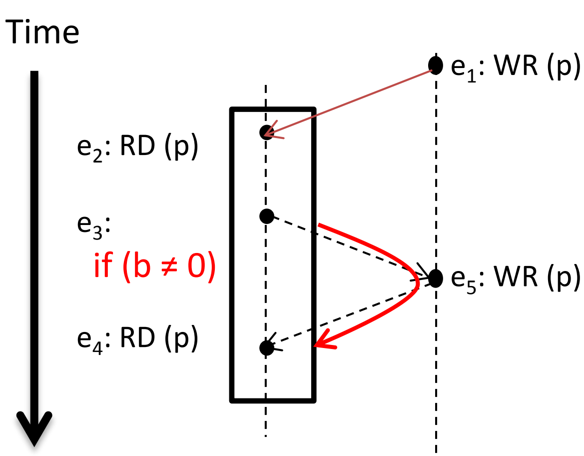
e_1: p := &a;

atomic{
e_2: b := p;
e_3: if (b ≠ 0)
e_4: *(p) := 10;
}

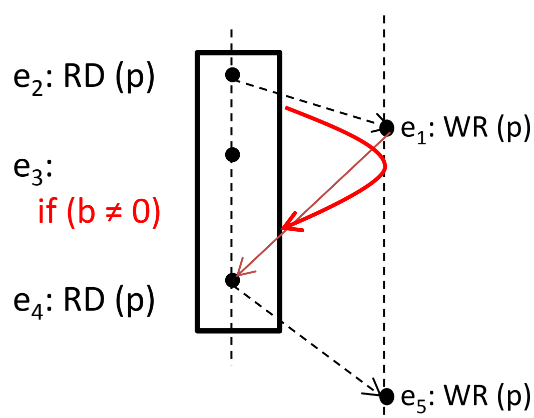
e_5: p := 0;



Original trace is bug-free.



(a) Unserializable and feasible.



(b) Unserializable and infeasible.

An alternate interleaving can be buggy.

However, if a read is *mismatched* (not reading from the original write), the alternate trace might be *infeasible* (since control flow could be altered).

→ Violation path

Predictive Analysis: Idea

- **Predictive analysis** [Rosu *et al*. CAV 07, Farzan *et al*. TACAS 09, …]

  – **Run a test execution and log information about *events* of interest**

  – **Generate a *predictive model* over the events, by relaxing some ordering constraints**

  – **Analyze the predictive model to check *alternate interleavings* of these events**

  – **Note: Does not cover events not observed in the trace**

- **Examples of predictive models**
  – **Control State Reachable (CSR) model: simple**
  – **Maximal Causal Model (MCM): good coverage**

Control State Reachable (CSR) Predictive Model

[Farzan & Parthasarathy, 2009]

shared variables: $x{=}0$ initially

Thread T_1	Thread T_2
atomic{	
t_1 : $a := x$	
t_2 : $x := a + 1$	
}	
	$t_3 : b := x$
	t_4 : if$(b > 0)$
	t_5 : $x := 5;$

observe "events" instead of "statements"
Ignore read-write values, log lock/unlock ops

t1: RD(x)	
t2: WR(x)	
	t3: RD(x)
	t4: nop
	t5: WR(x)

CSR reports a bogus bug

This interleaving is "not feasible"

But CSR would report it as an error

t1: RD(x)	
	t3: RD(x)
	t4: nop
	t5: WR(x)
t2: WR(x)	

Maximal Causal Model (MCM) for Predictive Analysis

[Serbanuta, Chen & Rosu, 2008]

shared variables: $x=0$ initially

Thread T_1	Thread T_2
atomic{ **$a := x + 1$**	
t_1 : $\cancel{a := x}$	
$\boxed{t_2 : \ x := a + 1}$	
}	
	$\boxed{t_3 : b := x}$
	t_4 : if$(b > 0)$
	t_5 : $x := 5;$

observe "events" instead of "statements"
Values of read and writes must be consistent

t1: RD(x) = 1	
t2: WR(x) = 2	
	t3: RD(x)=2
	t4: nop
	t5: WR(x)=5

MCM misses the real bug

This interleaving is actually "feasible" →

(but it would be missed by MCM)

t1: RD(x) = 1	
	t3: RD(x)=2
	t4: nop
	t5: WR(x)=5
t2: WR(x) = 1	

Symbolic Predictive Analysis

- **Symbolic Predictive Analysis** [Wang *et al.* FM 09, TACAS 10]
  - **Generate a precise predictive model by considering constraints due to synchronization and dataflow**
    - **Motivation: No false bugs, no missed bugs**
  - **Symbolically explore all possible thread interleavings of events in that trace, using an SMT solver**
    - **Motivation: Performs better than explicit enumeration**

Predictive Model

T_0

```
      int x = 0;
      int y = 0;
      pthread_t t1, t2;
      main() {
t1        pthread_create(t1,foo);
t2        pthread_create(t2,bar);
t3        pthread_join(t2);
t4        pthread_join(t1);
t5        assert( x != y);
      }
```

T_1

```
          foo() {
              int a;
t11           a=y;
t12           if (a==0) {
t13               x=1;
t14               a=x+1;
t15               x=a;
t16           }else
t17               x=0;
t18 }
```

T_2

```
          bar() {
              int b;
t21           b=x;
t22           if (b==0) {
t23               y=1;
t24               b=y+1;
t25               y=b;
t26           }else
t27               y=0;
t28 }
```

C program: multi-threaded, using *Pthreads*

```
t0: x=0,y=0;
t1: fork(1)
t2: fork(2)
```

Execution trace

```
t11: a=y;
t12: assume(a=0)
t13: x=1;
t14: a=x+1;
t15: x=a;
t18:
```

```
t21: b=x;
t26: assume(b≠0)
t27: y=0;
t28:
```

Concurrent Trace Program (CTP)

```
t3: join(2)
t4: join(1)
t5: assert(x ≠ y);
```

"assume(c)" means the (c)-branch is taken

Symbolic Predictive Analysis using CTP

[Wang *et al.* FM 09, TACAS 10]

- **Build an SMT formula (e.g. linear arithmetic)**
  - **F_program : encodes all feasible thread interleavings of CTP**
  - **F_property : encodes the property, e.g. an assertion violation**

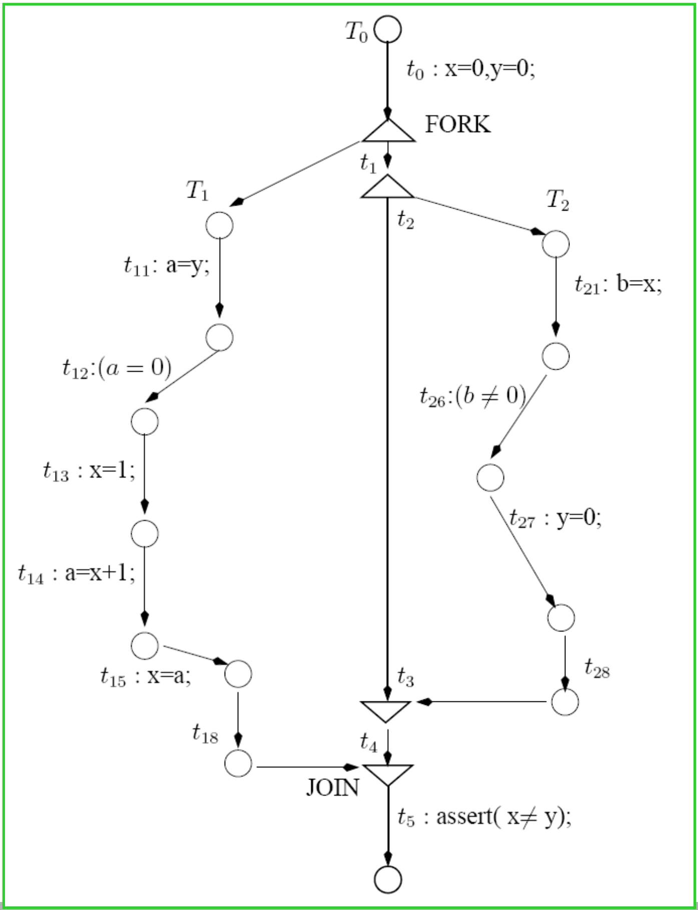
- **Solve using an SMT solver**

  (F_program $\wedge$ F_property)

  Sat → found a real error

  Unsat → no error in any interleaving

- **Improves**
  - **Precision over other predictive techniques**
  - **Covers all possible interleavings of the observed events.**



CTP Model: HB (Happens-Before) Constraints

- **Use a uniform HB (happens-before) model to capture constraints**
  - **Program order constraints**
    - e.g. sequential consistency (or weaker memory models …)
  - **Synchronization constraints**
    - e.g. fork-joins, wait-notify, mutual exclusion using locks, …
  - **Correctness violations**
    - e.g. assertion violations, data races, serializability violations

- **How is HB(t1, t2) implemented?**
  - **Event t1 happens strictly before t2**
  - **HB(t1, t2) := t1 < t2** where t1, t2 are integer variables

- **Use an SMT solver to solve the formula**
  - **Fusion used Yices** [Dutertre & de Moura 06]

Concurrent Static Single Assignment (CSSA) Encoding

Program order (within thread)

F_po: HB(t0,t1) & HB(t1,t2) & HB(t2,t3) & HB(t3,t4) & HB(t4,t5)
HB(t0,t11) & HB(t11,t12) & HB(t12,t13) & HB(t13,t14) & HB(t14,15) & HB(t15,t18) & HB(t18,t5)
HB(t1,t21) & HB(t21,t26) & HB(t26,t27) & HB(t27,t28) & HB(t28,t4)

Variable Definitions (with guards)

F_vd: (x0=0) & (y0=0) &
(a1=Y_1) & & (b1 = X_1)
(a1=0) & & (b1 != 0)
(x1=1) & & (y1 = 0)
(a2=X_2) &
(x2=a2)

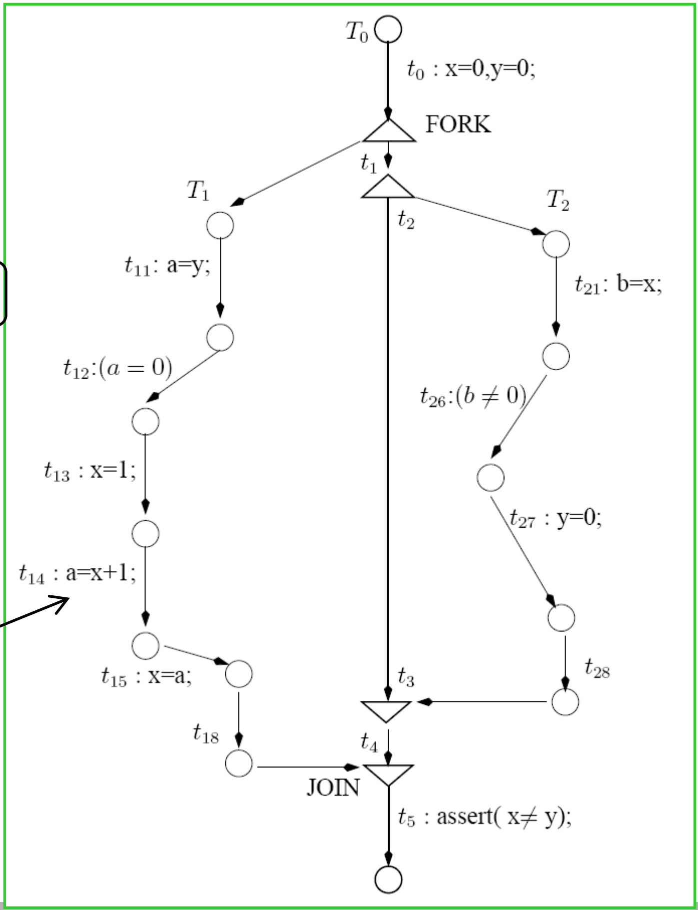
Assertion

F_property: (X_3 == Y_2)

Shared Variable Uses

F_pi: &((Y_1=y0) & HB(t11,t27) OR
(Y_1=y1) & HB(t27,t11))
& ((X_1=x0) & HB(t21,t13) OR
(X_1=x1) & HB(t13,t21) & HB(t21,t15) OR
(X_1=x2) & HB(t15,t21))
& (X_2=x1)
& (X_3=x2)
& (Y_2=y1)

π Functions for shared variable "uses"



CTP: Modeling Atomicity/Serializability Violations

Thread1

```
tc: if (buf_index + len < BUFFSIZE) {
        …
        …
tc’:  memcpy(buf[buf_index],log,len);
    }
```

Thread2

```
tr: buf_index += len;
```

- In practice, unserializable patterns cause a large number of concurrency errors

  [Lu *et al.* 2006]
- **Three-access pattern: involves three events (tc, tr, tc’) on a shared variable**
  - Two consecutive accesses in the current thread: **tc…tc’**
  - In between, one access from a remote thread: **tr**
- **Eight possible cases**
  - **Serializable: (R-R-R), (R-R-W), (W-R-R)**
  - **Un-serializable: (R-W-R), (R-W-W), (W-W-R), (W-W-W), (W-R-W)**

- **F_property: HB (tc, tr) && HB (tr, tc’)**

CTP: Modeling Context Bounding

- Let ***t_first*** be the start event of the CTP
- Let ***t_last*** be the end event of the CTP
- Let ***k*** be the max number of context switches allowed

$$(F\_program \;\&\&\; F\_property) \;\boxed{\&\& \; (t\_last - t\_first < k)}$$

Predictive Analysis using CTP: Summary

[Wang *et al.* FSE 09, FM 09, TACAS 10]

- **CTP (Concurrent Trace Program) model with HB constraints**
  - Precise symbolic model derived from a concurrent program trace
  - Models concurrency and synchronization primitives, dataflow, properties

- **SMT based symbolic search**
  - Based on CSSA (Concurrent Static Single Assignment) encoding
  - Can encode context bounding (e.g. like [CHESS])

- Can tune the level of precision in modeling and analysis
  - Use control state reachability to prune warnings [Kahlon & Wang, CAV 2010]
  - Modular analysis [Sinha & Wang, FSE 2010, POPL 2011]

Active Testing: CalFuzzer Tool

[Joshi, Naik, Park & Sen, CAV 09]

- Phase 1: Use imprecise static or dynamic program analysis to find "abstract" states where a potential violation can happen (e.g. datarace, deadlock, atomicity violation)

- Phase 2: "Direct" testing (by controlling the scheduler) based on the "abstract" states obtained from phase 1

More details in the Lab Session later today …

[Joshi, Park, Sen & Naik, PLDI 09]

Deadlock Detection: Example

Thread1

foo(o1,o2,true)

Thread2

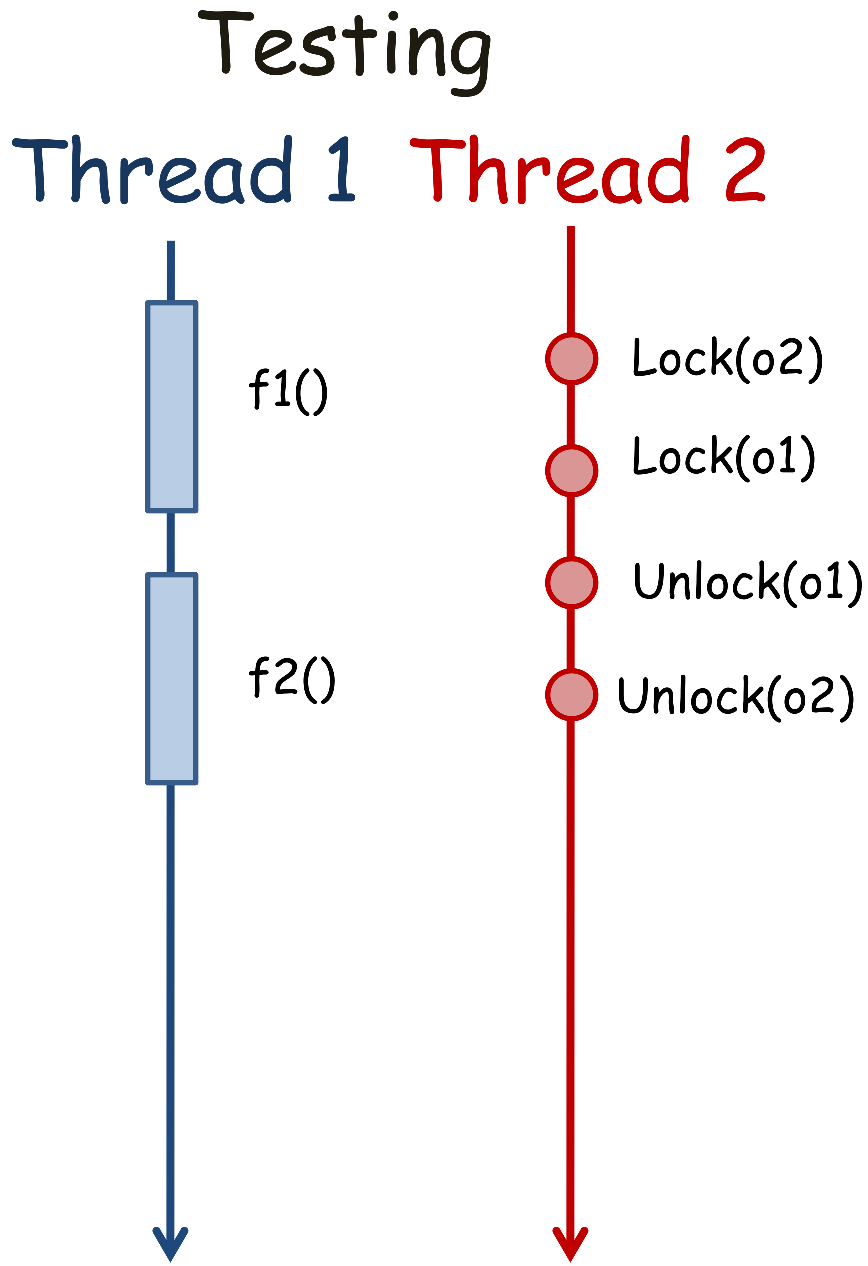
foo(o2,o1,false)

```
void foo(Object l1, Object l2, boolean flag) {

    if(flag) {
      // Long running computations
        s1: f1();
        s2: f2();
     }
    s3: synchronized(l1){
       s4: synchronized(l2){
       }
    }

}
```

Thread1 foo(o1,o2,true)

Thread2 foo(o2,o1,false)

```
void foo(Object l1, Object l2, boolean flag) {

    if(flag) {
     // Long running computations
       s1: f1();
       s2: f2();
    }
    s3: synchronized(l1){
      s4: synchronized(l2){
      }
    }

}
```

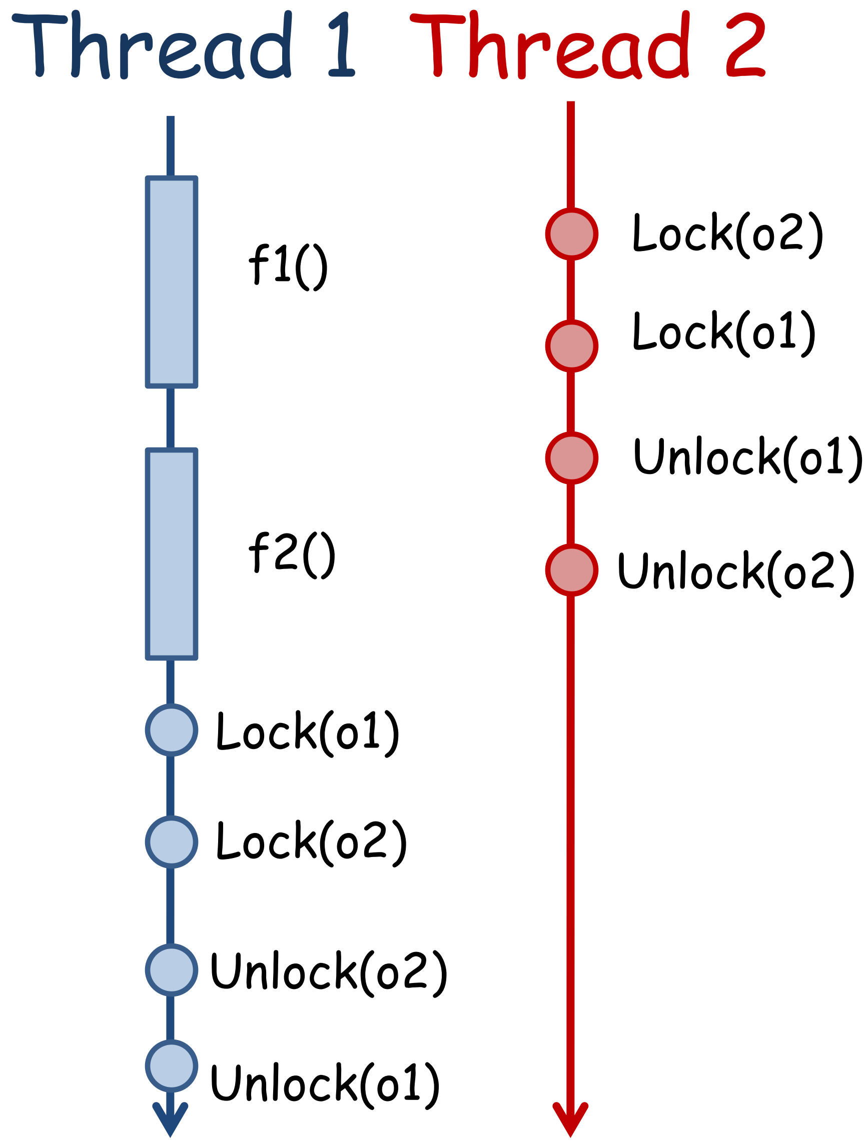


Testing

<u>Thread1</u> foo(o1,o2,true)

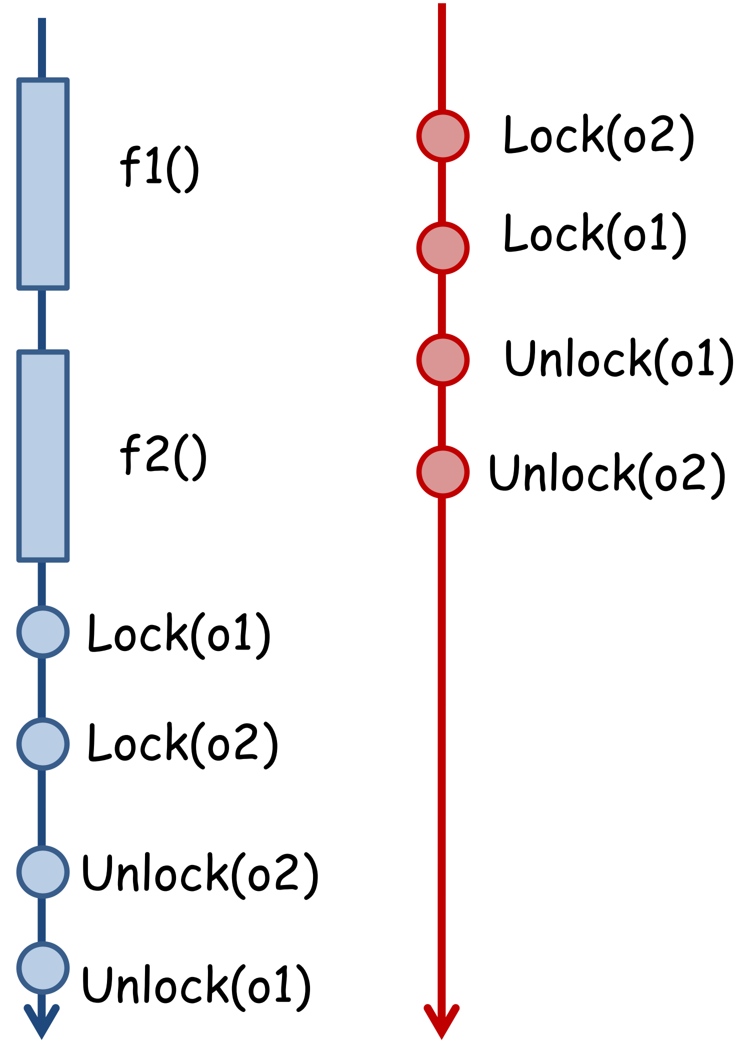
<u>Thread2</u> foo(o2,o1,false)

```
void foo(Object l1, Object l2, boolean flag) {

    if(flag) {
      // Long running computations
        s1: f1();
        s2: f2();
    }
    s3: synchronized(l1){
       s4: synchronized(l2){
       }
    }

}
```



Testing

Thread 1 Thread 2



<u>Thread1</u> foo(o1,o2,true)

<u>Thread2</u> foo(o2,o1,false)

```
void foo(Object l1, Object l2, boolean flag) {

    if(flag) {
      // Long running computations
        s1: f1();
        s2: f2();
    }
    s3: synchronized(l1){
      s4: synchronized(l2){
      }
    }

}
```

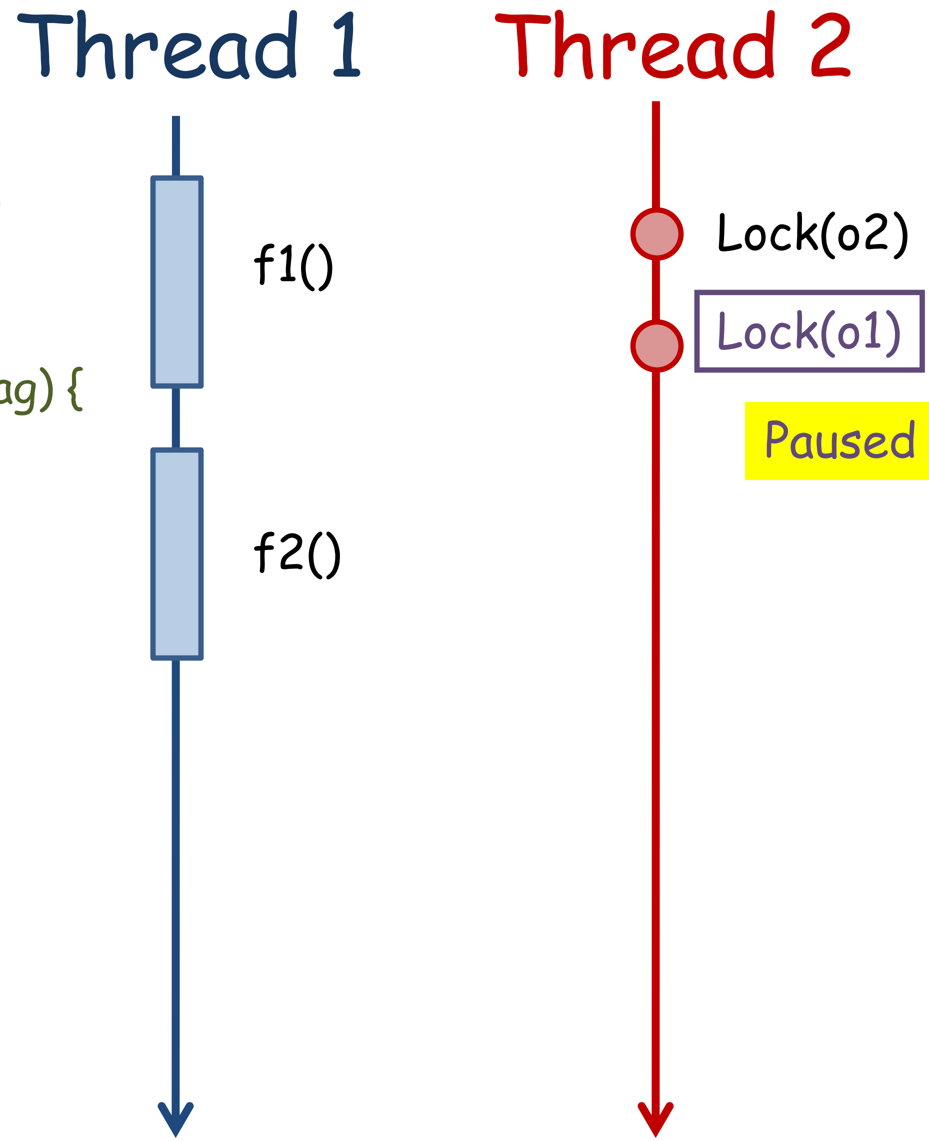
No deadlock detected

Deadlock Directed Testing

<u>Thread1</u> foo(o1,o2,true)

<u>Thread2</u> foo(o2,o1,false)

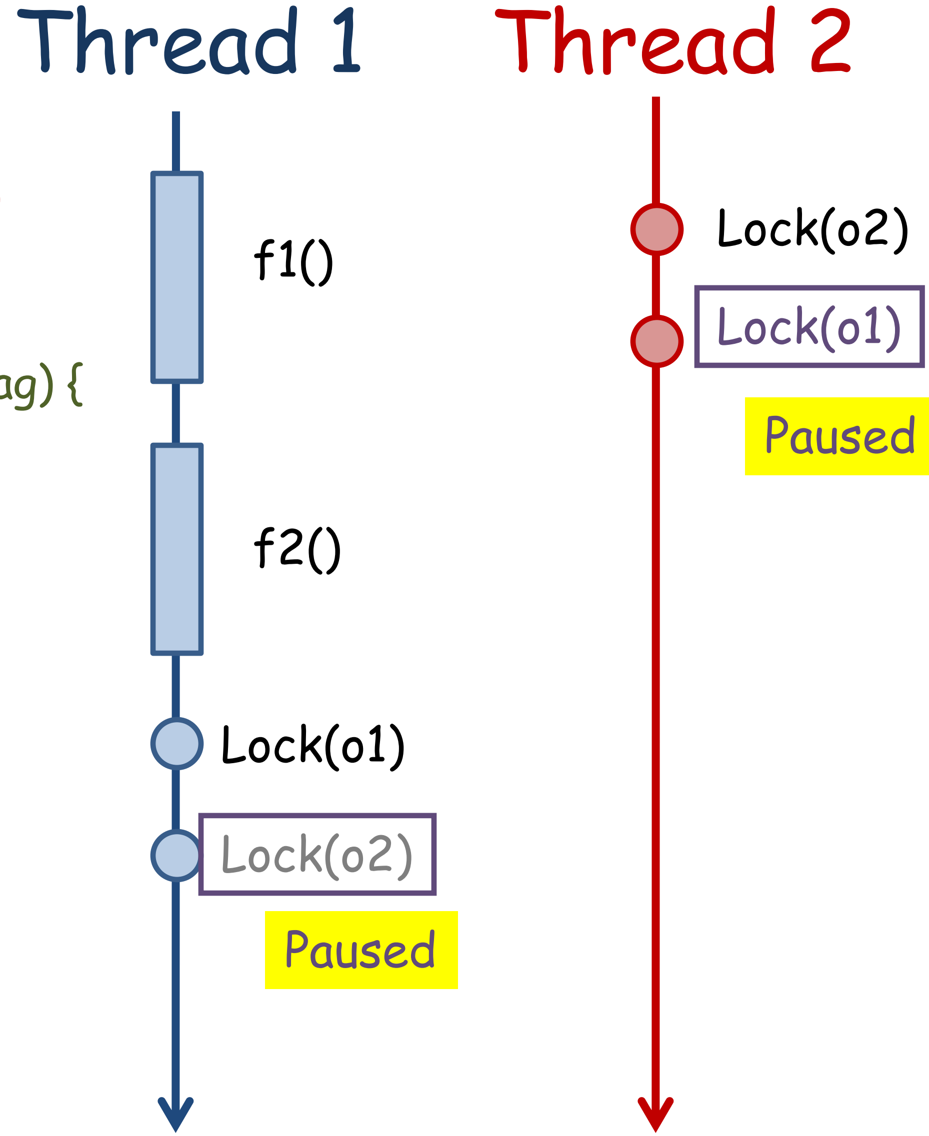
```
void foo(Object l1, Object l2, boolean flag) {

    if(flag) {
      // Long running computations
        f1();
        f2();
    }
    synchronized(l1){
      synchronized(l2){
      }
    }

}
```



Deadlock Directed Testing

Thread1	Thread2
foo(o1,o2,true)	foo(o2,o1,false)

```
void foo(Object l1, Object l2, boolean flag) {

    if(flag) {
      // Long running computations
        f1();
        f2();
    }
    synchronized(l1){
       synchronized(l2){
       }
    }

}
```

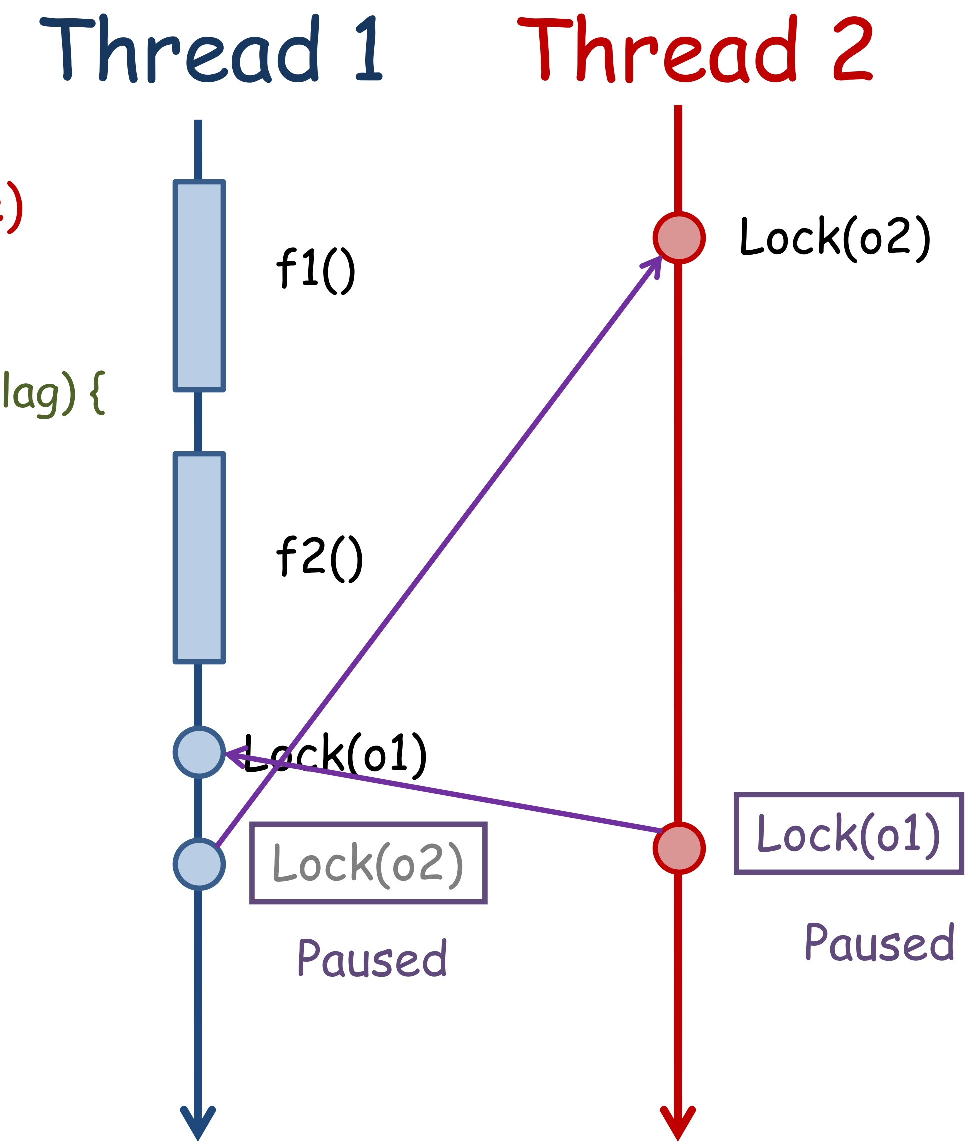


Deadlock Directed Testing

Thread1 Thread2

foo(o1,o2,true) foo(o2,o1,false)

```
void foo(Object l1, Object l2, boolean flag) {

    if(flag) {
      // Long running computations
        f1();
        f2();
    }
    synchronized(l1){
       synchronized(l2){
       }
    }

}
```



Deadlock Directed Testing

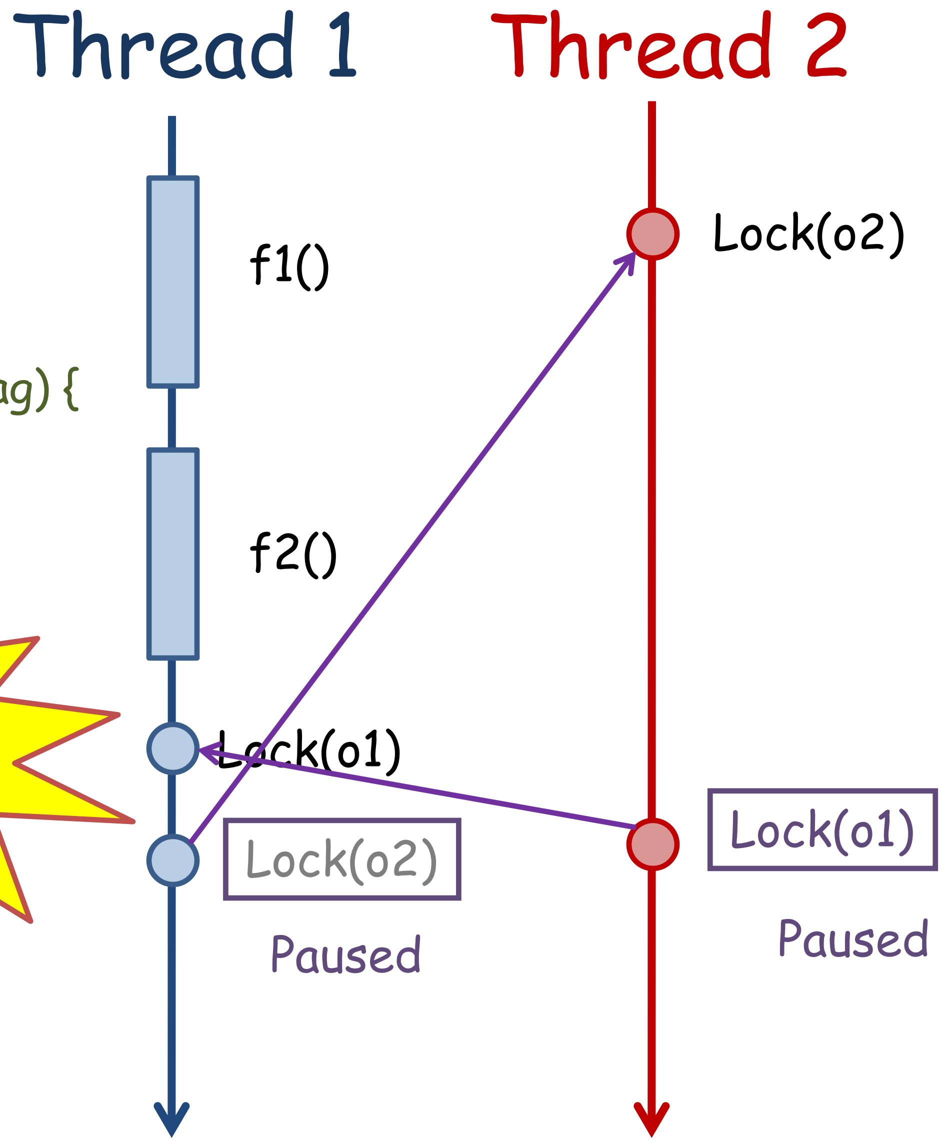
<u>Thread1</u>

foo(o1,o2,true)

<u>Thread2</u>

foo(o2,o1,false)

```
void foo(Object l1, Object l2, boolean flag) {

    if(flag) {
      // Long running computations
        f1();
        f2();
    }
    synchronized(l1){
      synchronized(l
      }
    }

}
```



Preempting threads

- How do we know where to pause a thread ?
  - Use existing static or dynamic analyses to find potential deadlock cycles
    - Note that these analyses may report false deadlock cycles
  - Use "information" recorded for a deadlock cycle to decide where to pause a thread
  - CalFuzzer uses a modified version of the Goodlock algorithm (iGoodlock)

[Havelund et al, Agarwal et al]

Take a Step Back …

- **What is the root cause of a "concurrency bug"?**
  - Programmers often make, but fail to enforce, some implicit assumptions regarding the concurrency control of the program
    - Certain blocks should be mutually exclusive → data race
    - Certain blocks should be executed atomically → atomicity violation
    - Certain operations should be executed in a fixed order → order violation

- **To chase "concurrency bugs", we would like to go after the "broken assumptions"…**
  - Exhaustively test all **concurrency control scenarios**
  - *But not all possible thread interleavings*

Coverage-Guided Systematic Testing

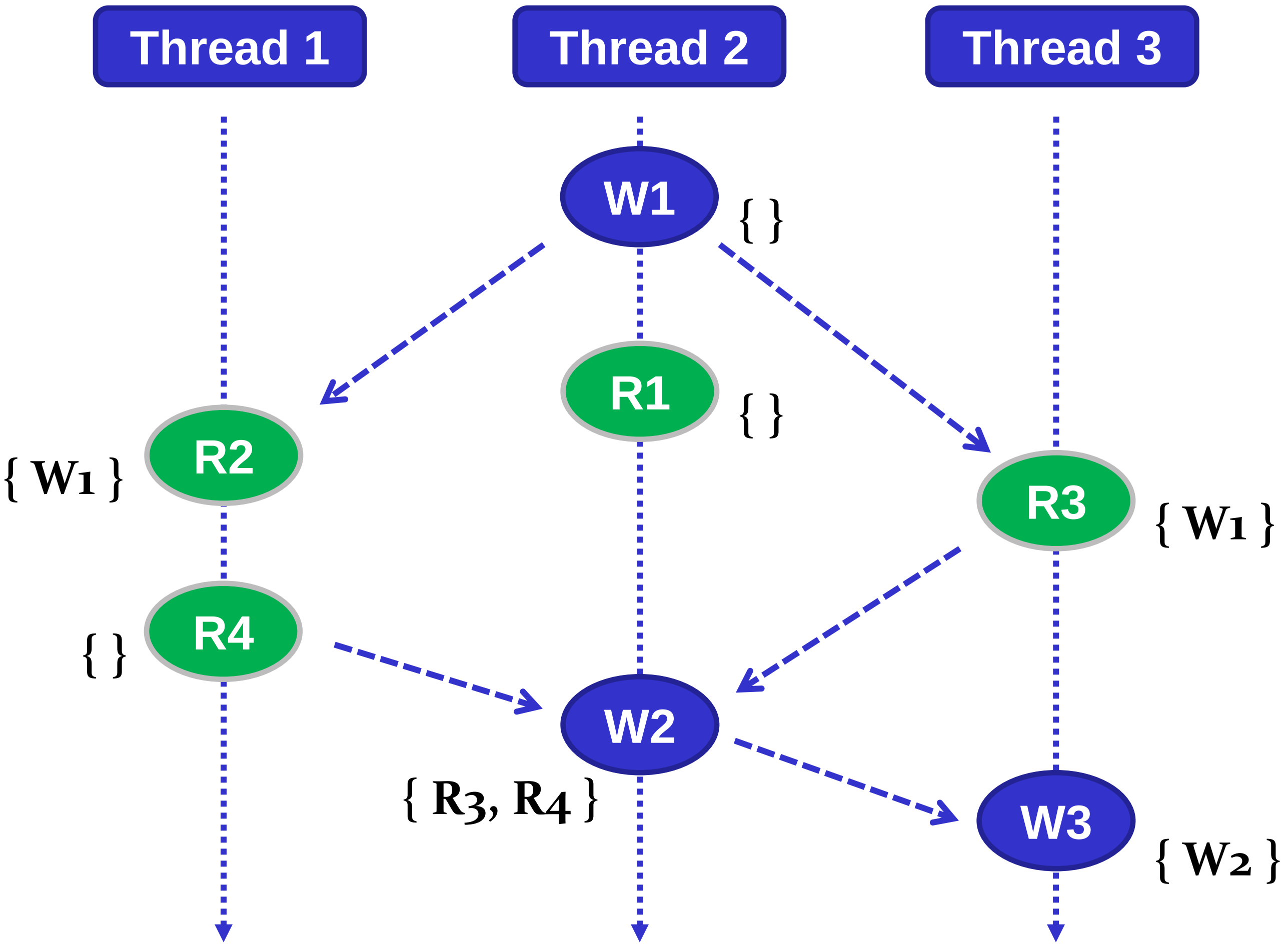
[Wang *et al*. ICSE 2011]

- **Coverage metric: "concurrency control scenario"**
  - **HaPSet (History-aware Predecessor Set)**
- **How do we use this metric?**
  - **Use a framework for systematically generating interleavings**
    - **e.g. stateless model checking**
  - **Keep track of HaPSets covered so far**
  - **Instead of DPOR/PCB, use HaPSet to prune away interleavings**
  - **Idea: Don't generate an interleaving to test if the "concurrency control scenario" (HaPSet) has already been covered**
- **Based on PSet (Predecessor Set)**
  - **Psets were used for enforcing safe executions**

Jie Yu, Satish Narayanasamy

A case for an interleaving constrained shared-memory multi-processor, International Symposium on Computer Architecture, 2009.

PSet (Predecessor Set) [Yu & Narayanasamy ISCA 09]



Psets are tracked for statements in code, not for events

PSet (statement):
the set of
<u>immediately dependent "remote" statements</u>

```
PSet(W1)  = {}
PSet(R1)  = {}
PSet(R2)  = {W1}
PSet(R3)  = {W1}
PSet(R4)  = {}
PSet(W2)  = {R3,R4}
PSet(W3)  = {W2}
```

HaPSet (extension)

[Wang *et al*. ICSE 2011]

1. Synchronization statements

- **PSet ignored synchronizations, e.g. lock/unlock, wait/notify**
- **HaPSet considers synchronizations – essential for concurrency**

2. Context & thread sensitivity

- **PSet (effectively) treats a statement as a (file,line) pair**
- **HaPSet treats a “statement” as a tuple (file,line,thr,ctx), where**
  - **thr = {local_thread, remote_thread} (exploits symmetry)**
  - **ctx = the truncated calling context**

Intuition: Why are HaPSets Useful?

```
    Thread T1
      …




    {
e2   if (p != 0)

e3     *(p) = 10;
    }
```

```
    Thread T2
     …
    {
e1    p = &a;
    }

    …



    {
e4    p = 0;
    }
```

From the given run

```
HaPSet(e1) = {}
HaPSet(e2) = {e1}
HaPSet(e3) = {}
HaPSet(e4) = {e3}
```

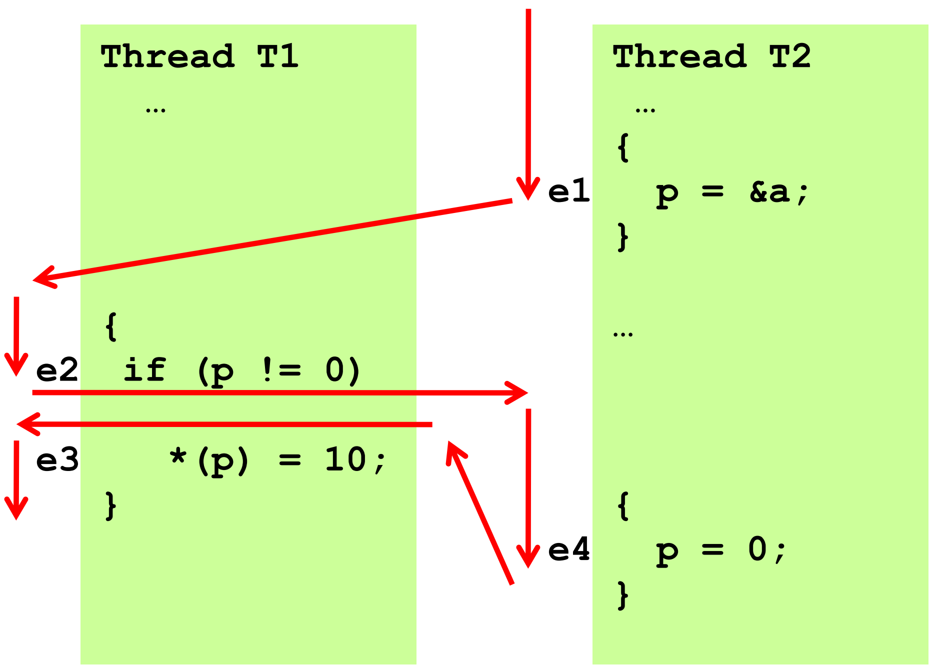
From all good runs

```
HaPSet(e1) = {e2}
HaPSet(e2) = {e1,e4}
HaPSet(e3) = {}
HaPSet(e4) = {e3}
```

Observations:
#1. In all good runs, HaPSet[e3] = { }
#2. In all good runs, e2 is not in HaPSet[e4]

Need only 2 test runs to capture all “good” runs

Why are HaPSets Useful?



From the given run

```
HaPSet(e1) = {}
HaPSet(e2) = {e1}
HaPSet(e3) = {}
HaPSet(e4) = {e3}
```

From all good runs

```
HaPSet(e1) = {e2}
HaPSet(e2) = {e1,e4}
HaPSet(e3) = {}
HaPSet(e4) = {e3}
```

From all (good and bad) runs

```
HaPSet(e1) = {e2}
HaPSet(e2) = {e1,e4}
HaPSet(e3) = {e4}
HaPSet(e4) = {e3,e2}
```

Observations:
#1. In all good runs, HaPSet[e3] = { }
#2. In all good runs, e2 is not in HaPSet[e4]

Steer search directly to a "bad" run

Does HaPSet Guided Search Work?

Thrift is a software framework by **Facebook**, for scalable cross-language services development.

The C++ library has **18.5K lines of C++ code**. It had a known **deadlock**.

HaPSet guided search

Much faster than Dynamic POR, PCB
Did not miss bugs in practice
(many other examples in paper)

Test Program				HaPSet		DPOR		PCB0		PCB1		PCB2	
	LoC	thrds	bug type	runs	time(s)	runs	time(s)	runs	time(s)	runs	time(s)	runs	time(s)
lib-w2-5t	18.5k	3	deadlk	14	27.8	23	18.6	512(no)	247.2	26	29.2	215	146.9
lib-w3-5t	18.5k	4	deadlk	18	27.5	733	TO	1301	TO	399	229.7	876	TO
lib-w4-5t	18.5k	5	deadlk	22	33.7	665	TO	1111	TO	980	TO	677	TO
lib-w5-5t	18.5k	6	deadlk	25	38.1	572	TO	899	TO	670	TO	582	TO

DPOR

PCB

Summary and Challenges

- **Verifying Concurrent Programs**
  - **Concurrent programs are difficult to get right**
  - **Active area of verification research**
    - **Model checking, Static analysis, Testing/dynamic verification, …**
    - **Precise analysis requires reasoning about synchronization**
      - Exploit programming patterns that are amenable for precise analysis
    - **Efficient analysis requires controlling complexity of interleavings**
      - Reductions, Implicit search, Abstractions, Compositional proofs

  - **Precision AND efficiency of analysis are needed for practical impact**
    - **Applications guided by practical concerns**
      - Context-bounding, Coverage-directed testing
    - **Advancements in Decision Procedures (SAT/SMT) offer hope**

- **Related Challenges**
  - **Multi-core systems, Many-core systems: Bug replay, debugging**
  - **Distributed systems: Systematic testing**
  - **Great opportunity due to proliferation of distributed networked services/systems**