Verifying Concurrent Programs
(Tutorial)

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

NEC Laboratories
America
Relentless passion for innovation

www.nec-labs.com

Acknowledgements

- Malay Ganai, Vineet Kahlon, Nishant Sinha*, Chao Wang* (NEC Labs)
- Akash Lal (MSR, India)
- Madanlal Musuvathi (MSR)
- Antoine Miné (CNRS)
- Kedar Namjoshi (Alcatel-Lucent)
- Andrey Rybalchenko, Ashutosh Gupta, Corneliu Poppea (TU Munich), Alexander Malkis (Imdea)
- Arnab Sinha (Princeton University)
- Tayssir Touili (LIAFA)
- Thomas Wahl (Northeastern University)

Motivation

❑ Key Computing Trends

Mobile Server Gaming

High Performance, Low Power

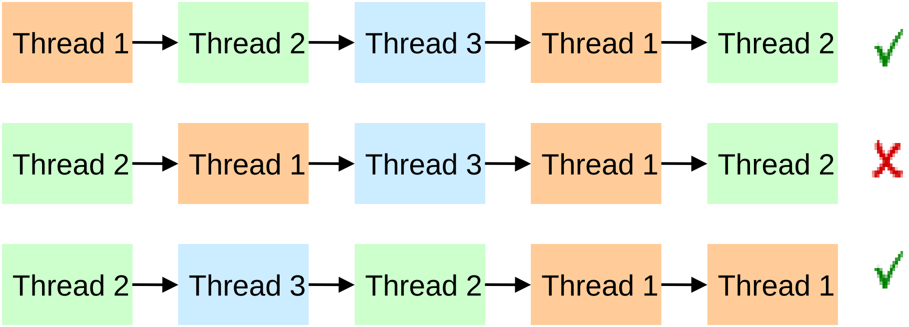
- **Single core solutions don't work**
- **Multi-core platforms**
- ***Need parallel, multi-threaded programming***

Data centers, Cloud platforms

- **Distributed, networked systems**

❑ Parallel/Multi-threaded Programming

- **Difficult to get right**
  - **Dependencies due to shared data**
  - **Subtle effects of synchronizations**
  - **Often manually parallelized**
- **Difficult to debug**
  - **too many interleavings of threads**
  - **hard to reproduce bugs**



Thread 1 → Thread 2 → Thread 3 → Thread 1 → Thread 2 ✓

Thread 2 → Thread 1 → Thread 3 → Thread 1 → Thread 2 ✗

Thread 2 → Thread 3 → Thread 2 → Thread 1 → Thread 1 ✓

What will I (try to) cover?

- ❑ **Basic elements**
  - – Model of concurrency
    - Asynchronous interleaving model (unlike synchronous hardware)
    - Explosion in interleavings
  - – Synchronization & Communication (S&C)
    - 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**

What I will not be able to cover

- **Active topics of research (but out of scope here)**
  - Parallel programs: Message-passing (e.g. MPI libraries), HPC applications
  - Synthesis/Optimization of locks/synchronizations for performance
  - Memory models: Relaxed memory models (e.g.TSO), Transactional memories
  - Object-based verification: Linearizability checking
  - Concurrent data structures/libraries: Lock-free structures
  - Separation logic: pointers & heaps, local reasoning
  - Theorem-proving , type analysis, runtime monitoring …

Models for Verifying Concurrent Programs

- **Finite state systems**
  - Asynchronous composition, S&C (including buffers/channels for messages), but no recursion
  - Setting: Inline procedures up to some bound to get finite models
  - Techniques: Bounded analysis (e.g. dynamic analysis, BMC)

- **Sequential programs**
  - Recursive procedures and other features, but no S&C and no interleavings
  - Setting: Add support for S&C and interleavings (thread interference)
  - Techniques: Bounded as well as unbounded analysis

- **Pushdown system models**
  - Stack of a pushdown system (PDS) models recursion, finite control, data is finite or infinite (with abstractions)
  - Setting: Consider interacting PDSs with various S&C
  - Techniques: PDS-based model checking

Outline

- ✓ **Introduction**

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

- ❑ **Static Verification Methods**
  - – Reduction: Partial order reduction, Symmetry
  - – Bounding: Context-bounded analysis, Memory Consistency-based analysis
  - – Program Abstraction: Static analysis, Thread-modular reasoning

- ❑ **Dynamic Verification Methods**
  - – Preemptive context bounding
  - – Predictive analysis
  - – Coverage-guided systematic testing

- ❑ **Conclusions**

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**

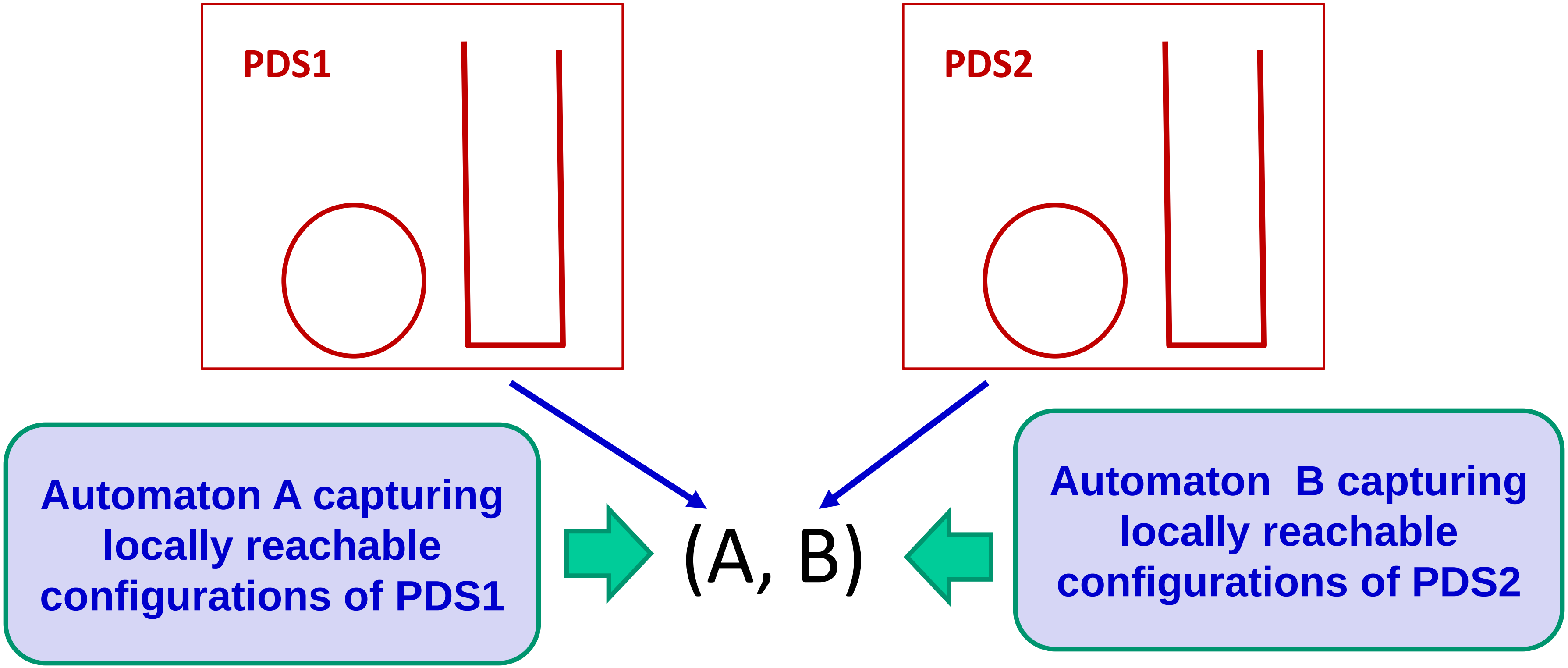
[Bouajjani *et al*., Walukeiwicz, Esparza *et al.*]

- ❑ **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
- ❑ Strategy: exploit the situations where PDSs are loosely coupled



Automaton A capturing locally reachable configurations of PDS1

(A, B)

Automaton B capturing locally reachable configurations of PDS2

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)**

- **Strategies to get around this undecidability**
  - **Special cases of programming patterns: Nested Locks, Bounded Lock Chains**
  - **Place restrictions on synchronization and communication (S&C)**

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);
 }
```

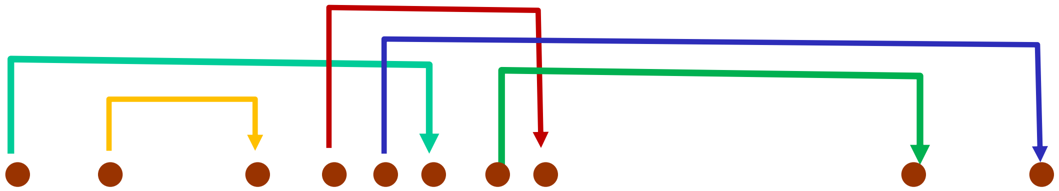
f calls g: nested locks
f calls h: non-nested locks

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

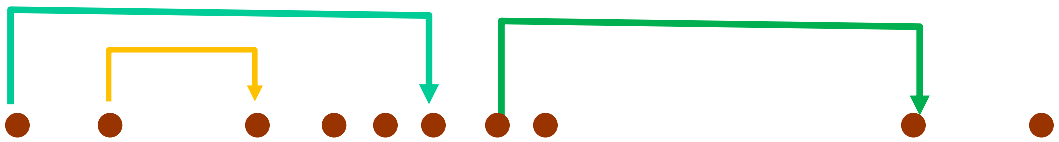
- **Multiple locks are enforced to be nested in Java$_{1.4}$ and C#**

Programming Pattern: Lock Chains

- Lock Chains

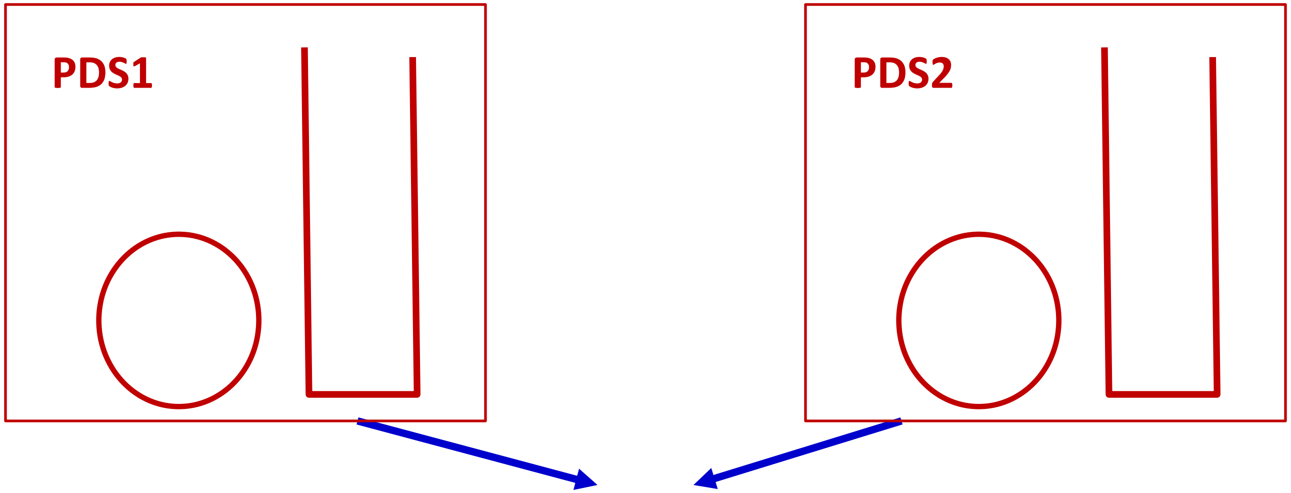


- Nested Locks: Chains of length one



- 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



PDS1

PDS2

(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 [POPL 07,LICS 09,CONCUR 11]

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

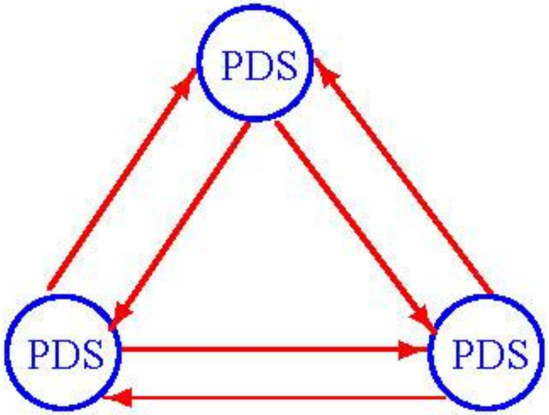
Restrict Synchronization & Communication: Example

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
    1. Partial Order Reduction (POR)
    2. Exploit symmetry
  - – Bound the problem
    3. Context-bounded analysis
    4. Memory Consistency-based analysis
  - – Use program abstractions and compositional techniques
    5. Static analysis
    6. Thread-modular reasoning

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

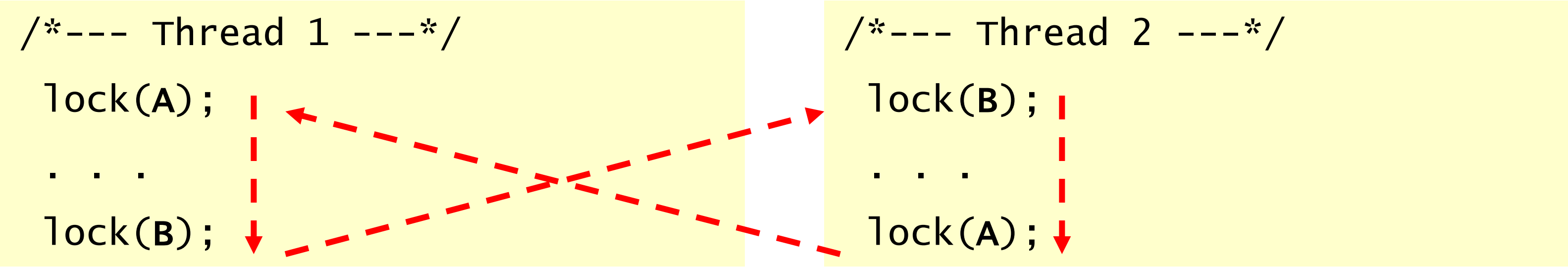
Common Concurrency Bugs

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

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

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

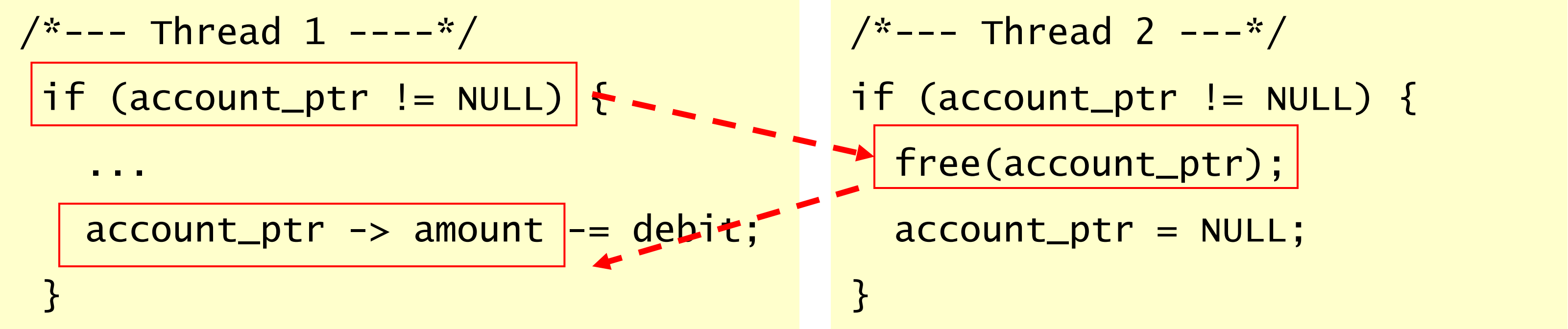
- **Deadlock: hold-and-wait cycles**



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

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

- **Atomicity violation: e.g. a common three-access pattern**



```
/*--- 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

- **Two popular approaches for datarace detection**

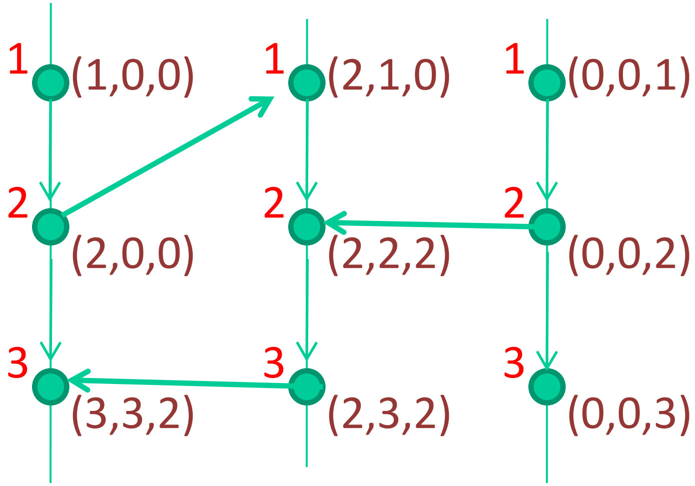
- **Lockset analysis** [Savage et al. 97, ERASER]
  - ***Lockset: set of locks held at a program location***
  - **Method:**
    - **Compute locksets for all locations in a program (statically or dynamically)**
    - **Race: When there are conflicting accesses from program locations with disjoint locksets**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

- **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**
    - **Provides opportunity for improving coverage over alternate schedules (discussed later)**

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 $a_1 \le b_1$ and $a_2 \le b_2$ and $a_3 \le b_3$

- 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)

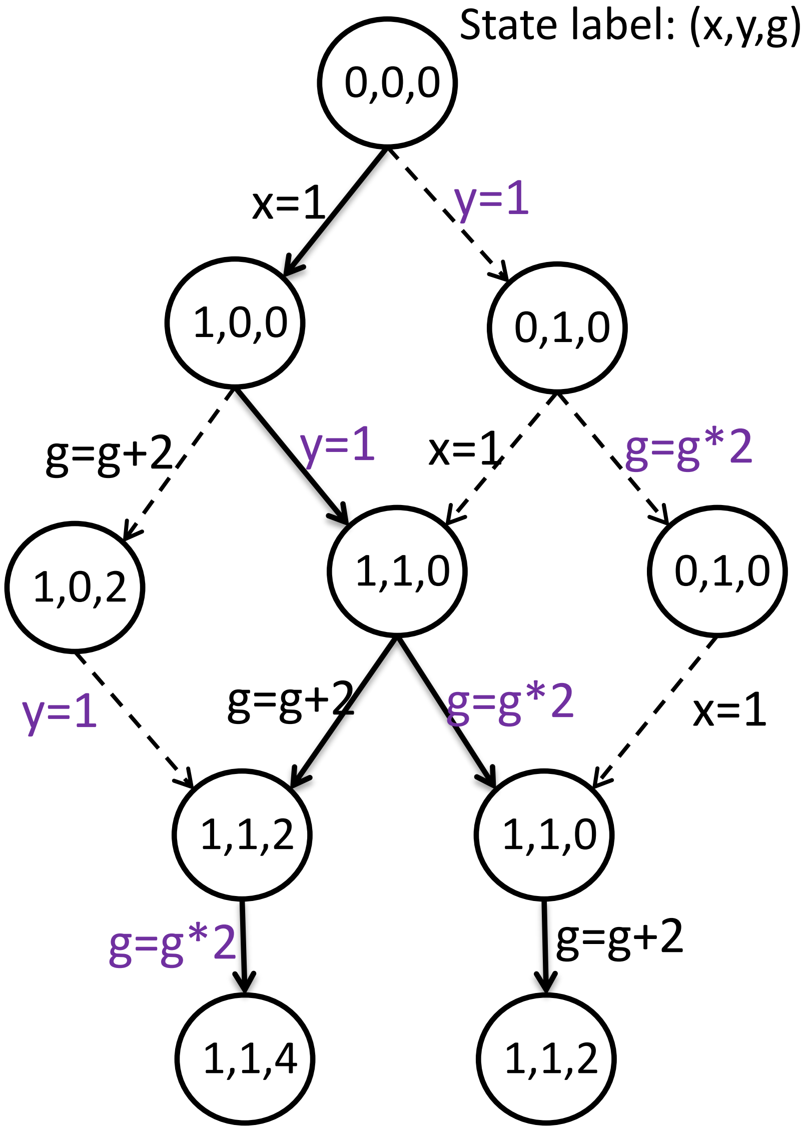
1. Partial Order Reduction (POR)

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.

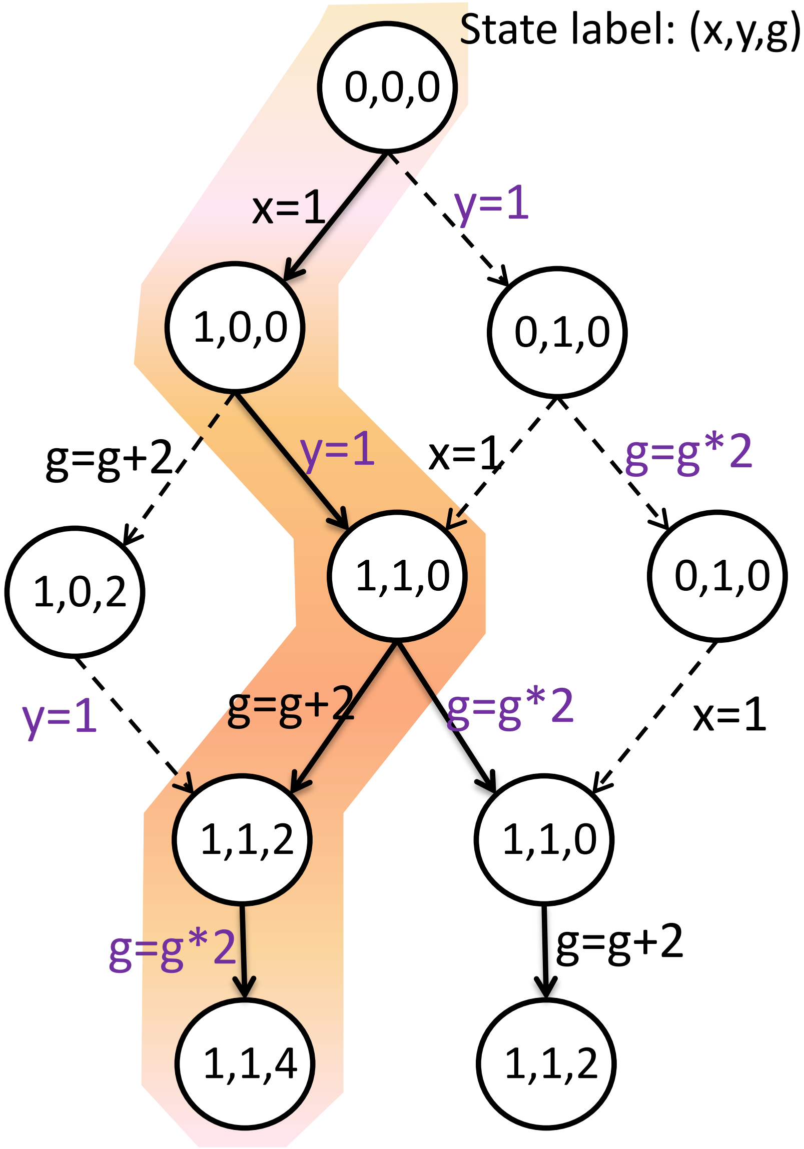
Thread 1	**Thread 2**
x=1	y=1
g=g+2	g=g*2

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.

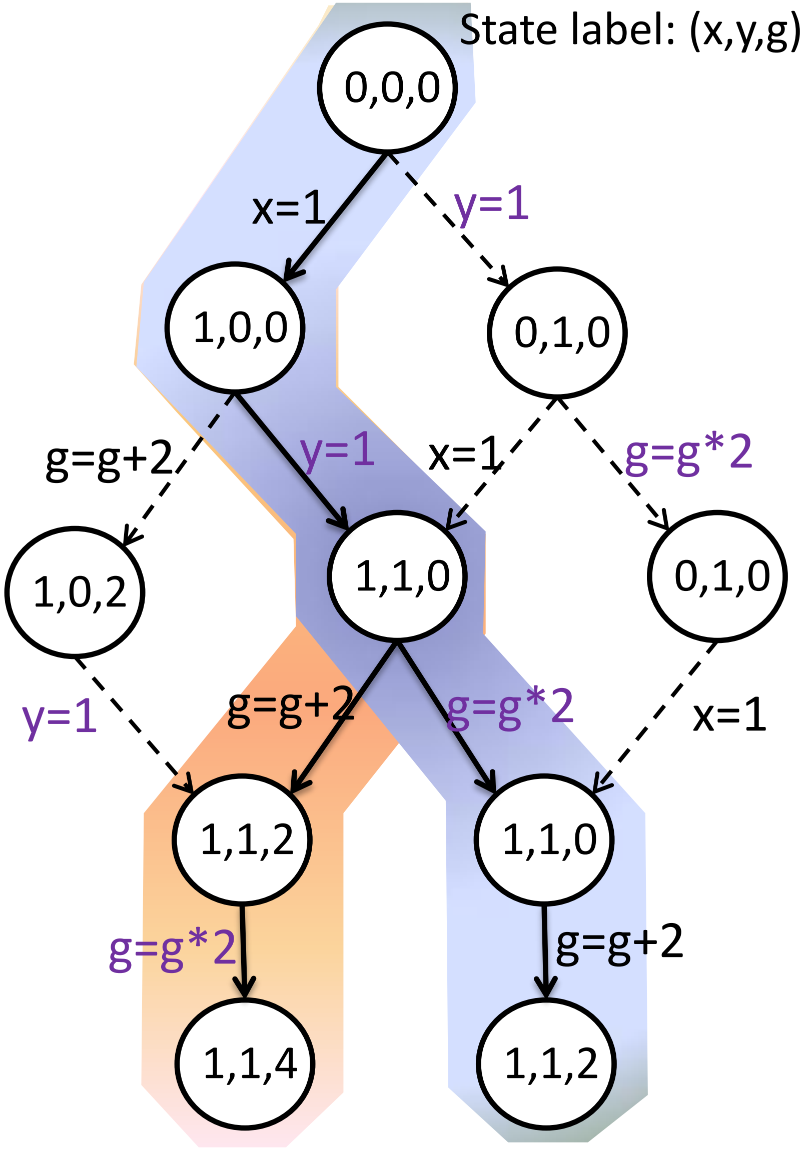
Thread 1	Thread 2
x=1	y=1
g=g+2	g=g*2

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.



State label: (x,y,g)

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]

2. Exploiting Symmetry in MC

There is a lot of redundancy in Replicated Systems

Many reachable states are “symmetric”, i.e.

identical up to thread permutations:

(N_1, T_2, N_3, T_4) (T_1, T_2, N_3, N_4) (T_1, N_2, N_3, T_4)

Counter Abstraction: collapse the above into

$$\langle \#N : 2, \#T : 2, \#C : 0 \rangle$$

Local States and Counter Abstraction

```
<#A000: 3, #A001: 0, #A002: 0, #A003: 0, #A004: 0, #A005: 0, #A006: 0, #A007: 0, #A008: 0, #A009: 0, #A010: 0, #A011: 0,
 #A012: 0, #A013: 0, #A014: 0, #A015: 0, #A016: 0, #A017: 0, #A018: 0, #A019: 0, #A020: 0, #A021: 0, #A022: 0, #A023: 0,
 #A024: 0, #A025: 0, #A026: 0, #A027: 0, #A028: 0, #A029: 0, #A030: 0, #A031: 0, #A032: 0, #A033: 0, #A034: 0, #A035: 0,
 #A036: 0, #A037: 0, #A038: 0, #A039: 0, #A040: 0, #A041: 0, #A042: 0, #A043: 0, #A044: 0, #A045: 0, #A046: 0, #A047: 0,
 #A048: 0, #A049: 0, #A050: 0, #A051: 0, #A052: 0, #A053: 0, #A054: 0, #A055: 0, #A056: 0, #A057: 0, #A058: 0, #A059: 0,
 #A060: 0, #A061: 0, #A062: 0,                                                                   : 0, #A070: 0, #A071: 0,
 #A072: 0, #A073: 0, #A074: 0,                                                                   : 0, #A082: 0, #A083: 0,
 #A084: 0, #A085: 0, #A086: 0,                                                                   : 0, #A094: 0, #A095: 0,
 #A096: 0, #A097: 0, #A098: 0,                                                                   : 0, #A106: 0, #A107: 0,
 #A108: 0, #A109: 0, #A110: 0,                                                                   : 0, #A118: 0, #A119: 0,
 #A120: 0, #A121: 0, #A122: 0, #A123: 0, #A124: 0, #A125: 0, #A126: 0, #A127: 0, #A128: 0, #A129: 0, #A130: 0, #A131: 0,
 #A132: 0, #A133: 0, #A134: 0, #A135: 0, #A136: 0, #A137: 0, #A138: 0, #A139: 0, #A140: 0, #A141: 0, #A142: 0, #A143: 0,
 #A144: 0, #A145: 0, #A146: 0, #A147: 0, #A148: 0, #A149: 0, #A150: 0, #A151: 0, #A152: 0, #A153: 0, #A154: 0, #A155: 0,
 #A156: 0, #A157: 0, #A158: 0, #A159: 0, #A160: 0, #A161: 0, #A162: 0, #A163: 0, #A164: 0, #A165: 0, #A166: 0, #A167: 0,
 #A168: 0, #A169: 0, #A170: 0, #A171: 0, #A172: 0, #A173: 0, #A174: 0, #A175: 0, #A176: 0, #A177: 0, #A178: 0, #A179: 0,
 #A180: 0, #A181: 0, #A182: 0, #A183: 0, #A184: 0, #A185: 0, #A186: 0, #A187: 0, #A188: 0, #A189: 0, #A190: 0, #A191: 0,
 #A192: 0, #A193: 0, #A194: 0, #A195: 0, #A196: 0, #A197: 0, #A198: 0, #A199: 0, #A200: 0, #A201: 0, #A202: 0, #A203: 0,
 #A204: 0, #A205: 0, #A206: 0, #A207: 0>
```

local state explosion

Combatting Local State Explosion in Symbolic Exploration

Solution: **omit zero-valued counters** from global states!

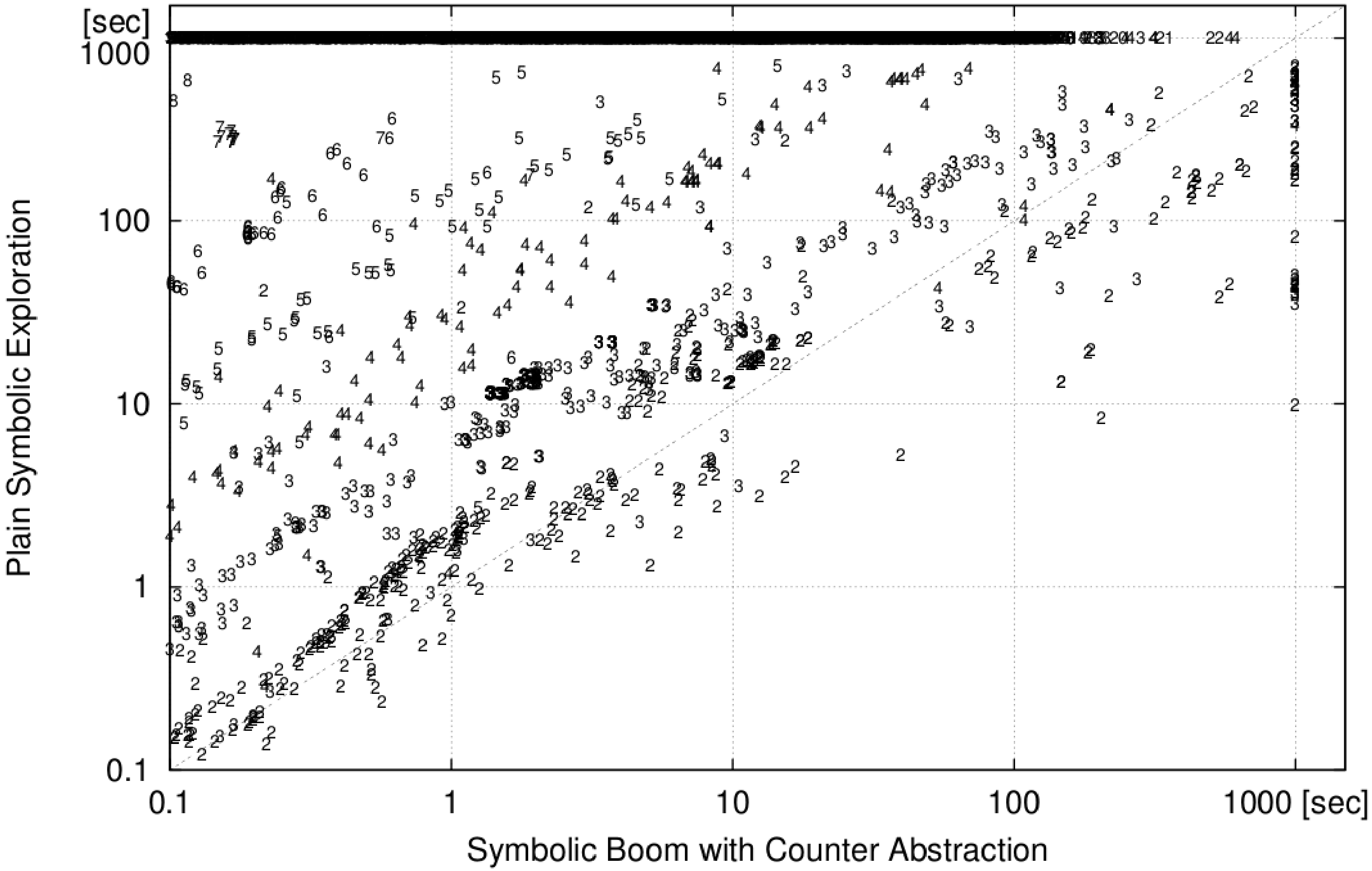
→ abstract state space down from n^{2^v} to $n^{\min\{n,2^v\}}$

Huge improvement if $n \ll 2^v$!

Symbolic Counter Abstraction: *n* threads, *v* local variables

[Basler et al. CAV 09]

Computing Reachable Program States



3. Context-Bounded Analysis

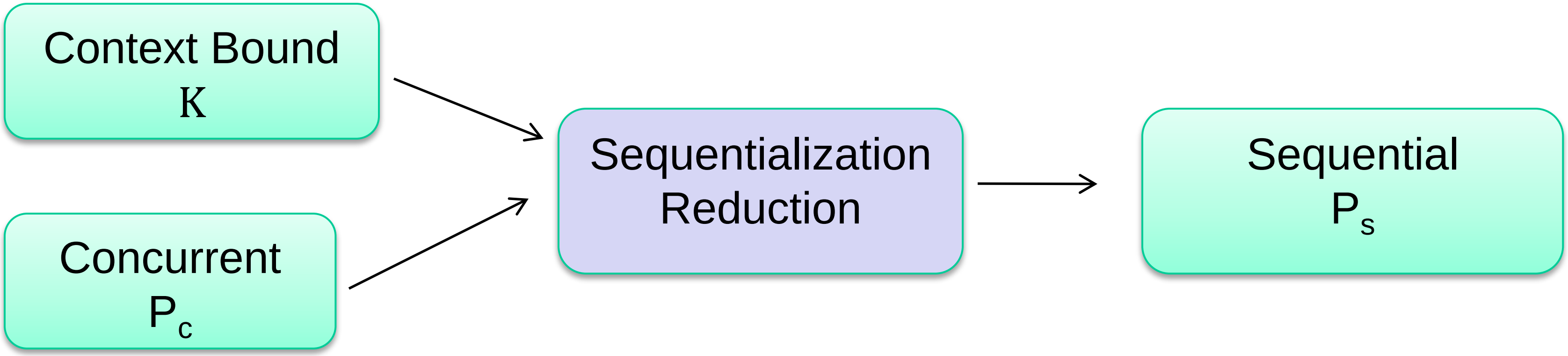
- **Recall**
  - **The general problem of verifying a concurrent program (recursive procedures with synchronization) is undecidable.**
  - **We have seen various strategies to get around undecidability**
    - Exploiting patterns of synchronization
    - Restricting synchronization & communication
 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

CBA using Sequentialization

[Lal & Reps, CAV 08]

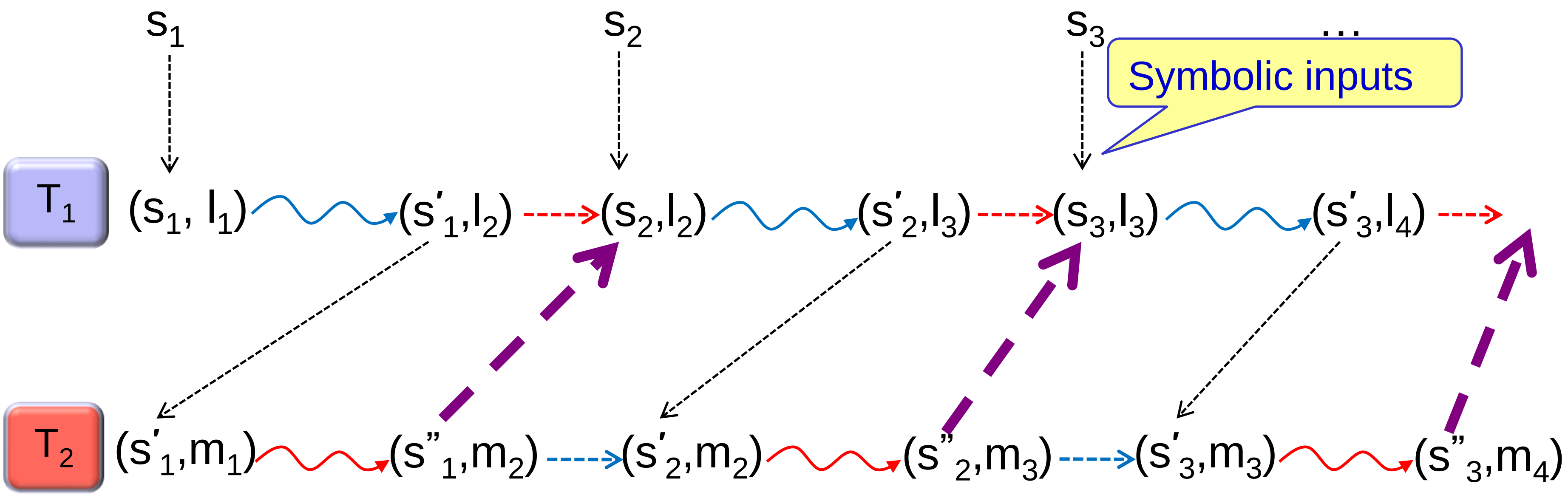
- ❑ **Sequentialization: Reduce CBA to sequential program analysis**



- 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

Sequentialization: From Concurrent to Sequential

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



s_1 s_2 s_3 …

Symbolic inputs

T_1 (s_1, l_1) (s'_1,l_2) (s_2,l_2) (s'_2,l_3) (s_3,l_3) (s'_3,l_4)

T_2 (s'_1,m_1) (s''_1,m_2) (s'_2,m_2) (s''_2,m_3) (s'_3,m_3) (s''_3,m_4)

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: From Concurrent to Sequential

❑ **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

4. 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

- Obtain an Interference Skeleton (IS) from (unrolled) Program
  - Global read and write events and their program order
  - Encoded as Φ_{IS}
- SC axioms for reads/writes in IS
  - Quantified first-order logic formula Π
- 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)

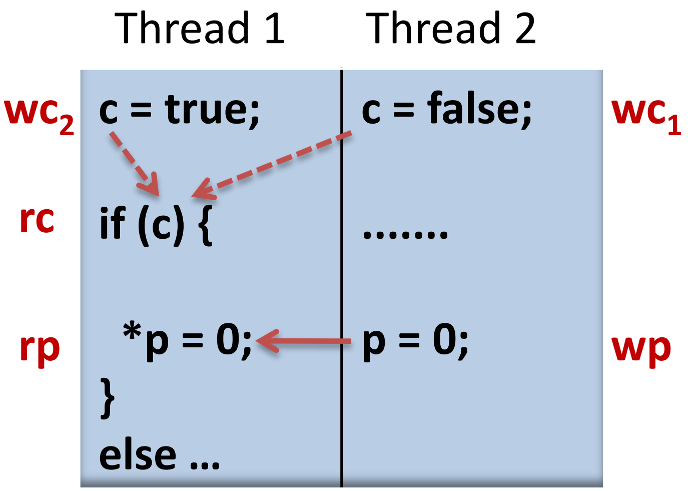
Sequential Consistency Axioms

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

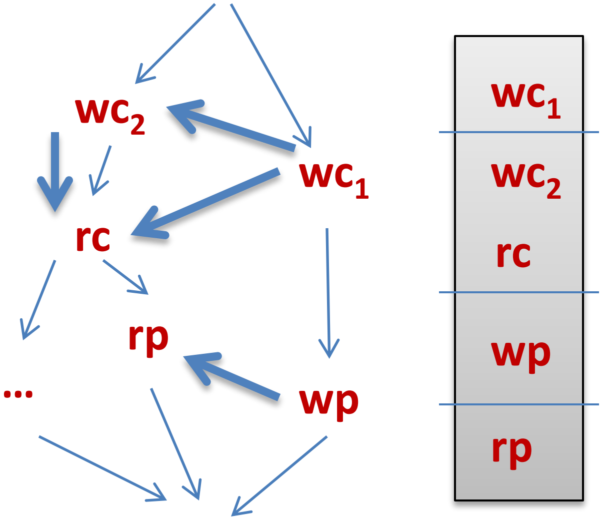
- Specified in typed first-order logic
  - read **r**, write **w**: Access type
- **Link** Predicate: **link (r,w)**
  - holds if read **r** observes write **w** in an execution
  - Exclusive : link (r,w) => ∀ w'. ¬ link (r,w')
- **Must-Happen-before** Predicate : **hb (w,r)**
  - **w** must happen before **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
wc2   c = true;       c = false;   wc1
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) /\ val(rc) = true)

en(rp) => en (rc) (Path conditions)
and, en(rp) => val (rc) = true (*)

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

link (rc, wc_1) ∨ 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

5. 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

Static Analysis of Concurrent Programs

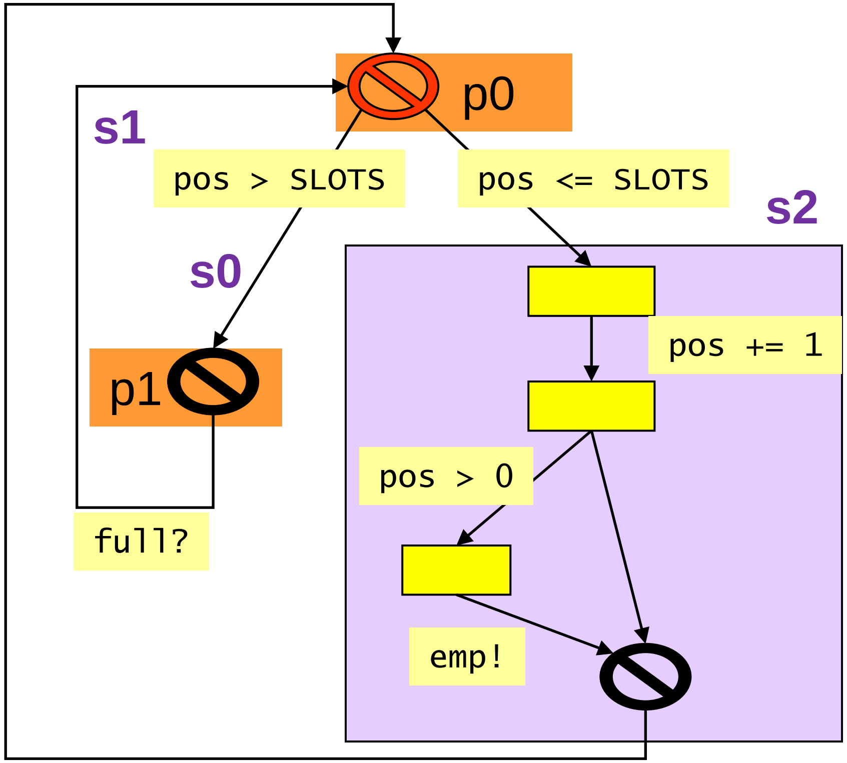
- ❑ **Intuitively, one can reason over a set of product control states**
  - **Not all product (global) control states, but only the *statically reachable* states**
  - **Transaction Graph:**
    - **Each node is a *statically reachable* global control state,**
    - **Each edge is a *transaction*, i.e. 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**

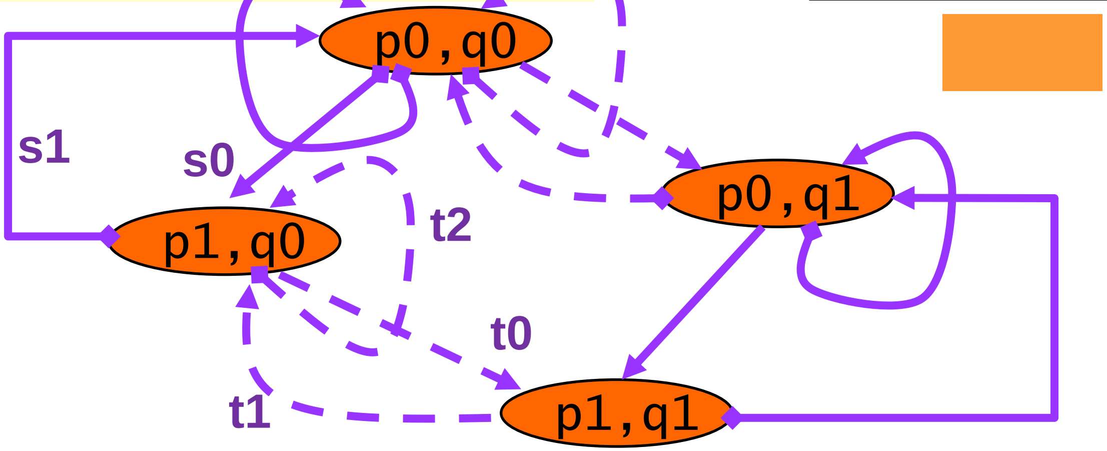
- ❑ **Refinement Approach** [Kahlon et al. TACAS 09]
  - **At any stage, the transaction graph *over-approximates* the set of thread interleavings for sound static analysis or model checking**
  - ***Iteratively refine* the transaction graph by computing *invariants***

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, Sankaranarayanan & Gupta, 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 the 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	# After	Time	#Witness	#Unknown
				(sec)	Invariants	(sec)	MC	
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)

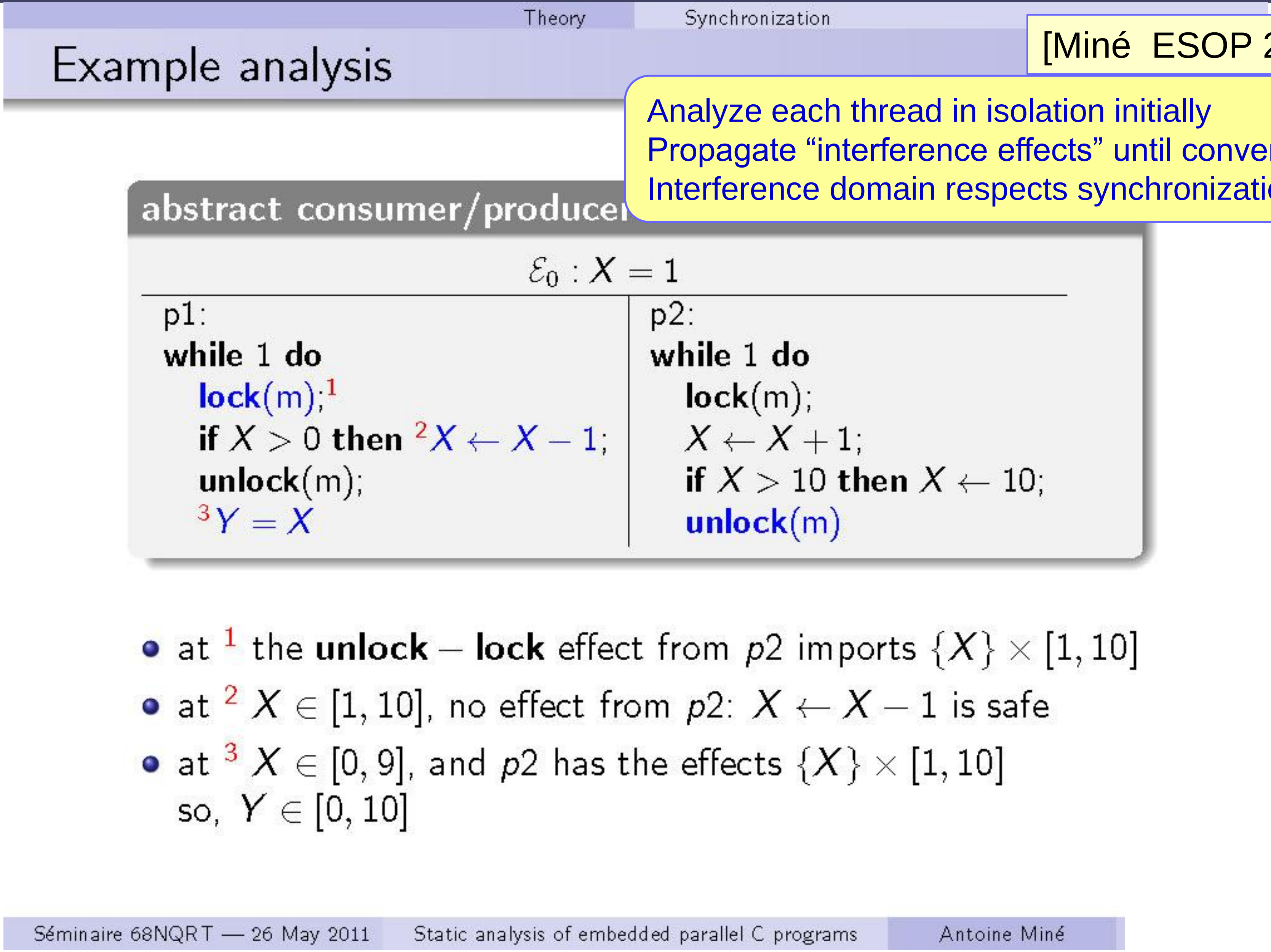
After Phase 2 (Warning Reduction)

After Phase 3 (Model Checking)

Static Analysis: Propagating Interference Effects

[Miné ESOP 2011]

Analyze each thread in isolation initially
Propagate “interference effects” until convergence
Interference domain respects synchronization



Theory Synchronization

Example analysis

abstract consumer/producer

$\mathcal{E}_0 : X = 1$

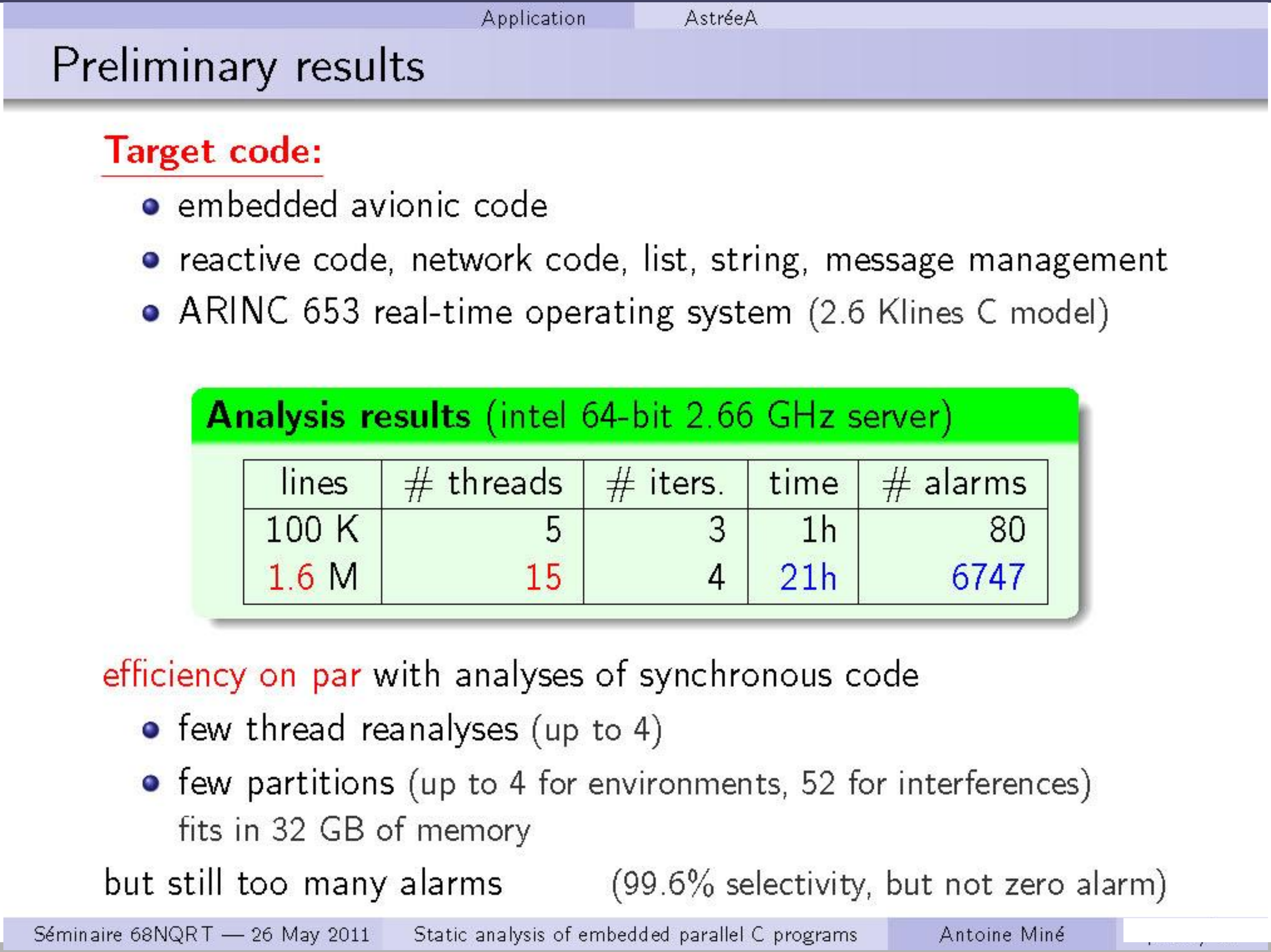
```
p1:
while 1 do
    lock(m);[1]
    if $X > 0$ then [2]$X \leftarrow X - 1$;
    unlock(m);
    [3]$Y = X$
```

```
p2:
while 1 do
    lock(m);
    $X \leftarrow X + 1$;
    if $X > 10$ then $X \leftarrow 10$;
    unlock(m)
```

- at [1] the **unlock** – **lock** effect from $p2$ imports $\{X\} \times [1, 10]$
- at [2] $X \in [1, 10]$, no effect from $p2$: $X \leftarrow X - 1$ is safe
- at [3] $X \in [0, 9]$, and $p2$ has the effects $\{X\} \times [1, 10]$
  so, $Y \in [0, 10]$

Séminaire 68NQRT — 26 May 2011 Static analysis of embedded parallel C programs Antoine Miné

Astrée Analyzer



Preliminary results

Target code:

- embedded avionic code
- reactive code, network code, list, string, message management
- ARINC 653 real-time operating system (2.6 Klines C model)

Analysis results (intel 64-bit 2.66 GHz server)

lines	# threads	# iters.	time	# alarms
100 K	5	3	1h	80
1.6 M	15	4	21h	6747

efficiency on par with analyses of synchronous code

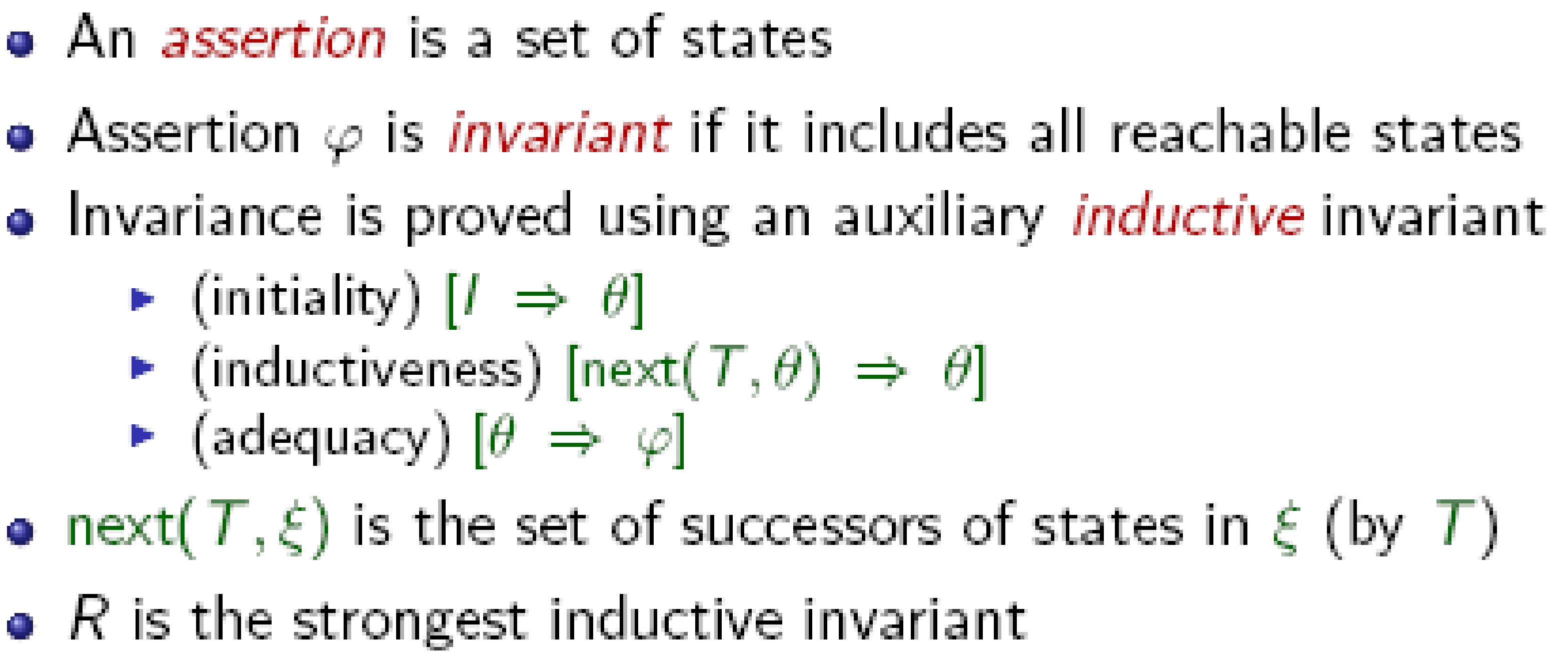
- few thread reanalyses (up to 4)
- few partitions (up to 4 for environments, 52 for interferences)
  fits in 32 GB of memory

but still too many alarms (99.6% selectivity, but not zero alarm)

6. 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
  - (initiality) $[I \Rightarrow \theta]$
  - (inductiveness) $[\mathrm{next}(T, \theta) \Rightarrow \theta]$
  - (adequacy) $[\theta \Rightarrow \varphi]$
- $\mathrm{next}(T, \xi)$ is the set of successors of states in ξ (by T)
- R is the strongest inductive invariant

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, …)

THREADER

[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

Compositional Verification of Multi-threaded Programs

- Given a transition system: $(\varphi_{init}, \rho_1 \vee \cdots \vee \rho_{\mathrm{N}}, \varphi_{err})$
- Find auxiliary assertions $R_1, \ldots, R_{\mathrm{N}}$ (reachable states) and $E_1, \ldots, E_{\mathrm{N}}$ (environment transitions) such that:

$$\begin{array}{lll}
\varphi_{init} & \rightarrow R_i & \text{for } i \in 1..\mathrm{N} \\
R_i \wedge (\rho_i \vee E_i \wedge \rho_i^{=}) & \rightarrow R_i' & \text{for } i \in 1..\mathrm{N} \\
\bigwedge_{i \in 1..\mathrm{N}} R_i \wedge \varphi_{err} & \rightarrow \mathit{false} & \\
(\bigvee_{j \in 1..\mathrm{N} \setminus \{i\}} R_j \wedge \rho_j) & \rightarrow E_i & \text{for } i \in 1..\mathrm{N}
\end{array}$$

Interference considered by thread i

Interference generated by other threads

Outline

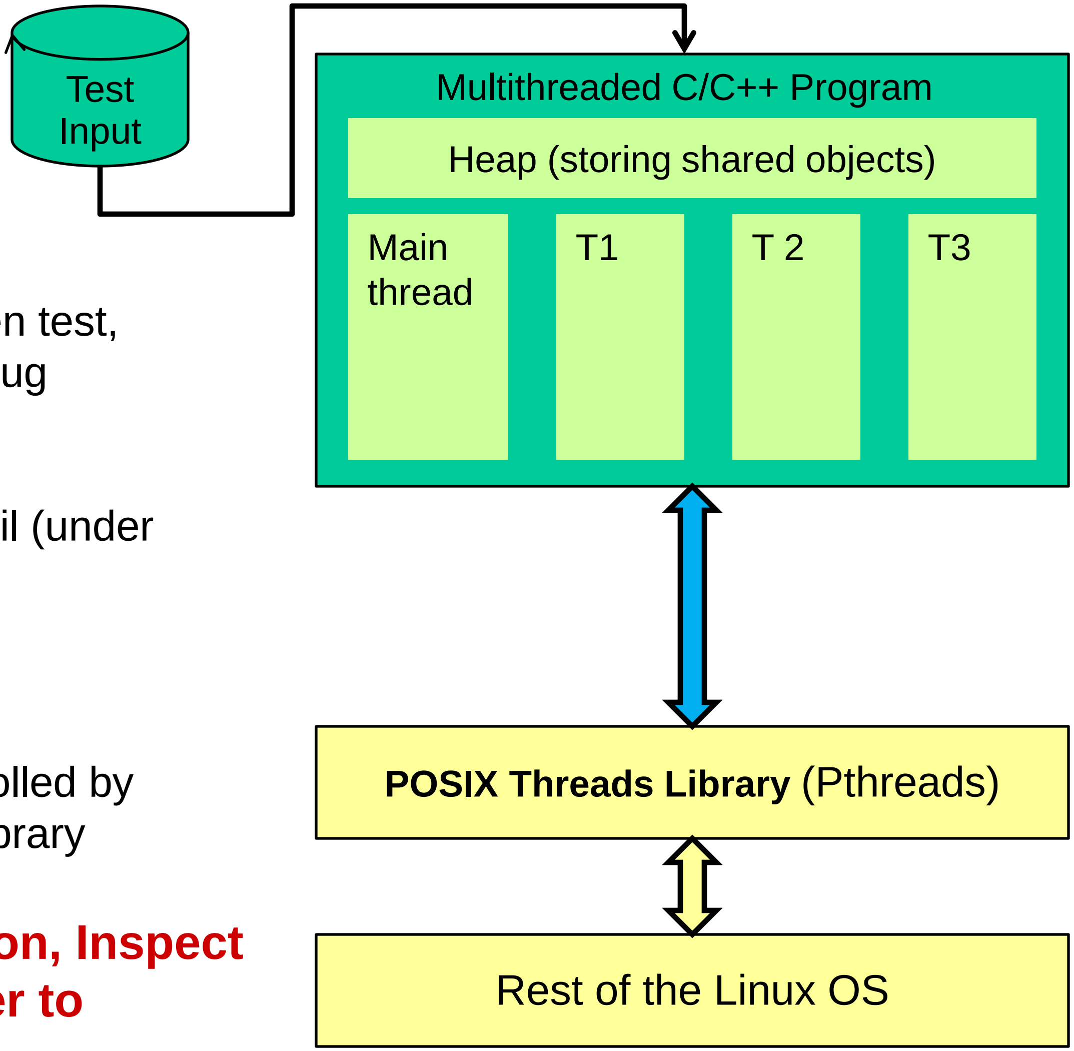
- ✓ Introduction
- ✓ PDS-based Model Checking
  - ✓ Theoretical results
- ✓ Static Veri
  - ✓ Reduct
  - ✓ Boundi nalysis
  - ✓ Abstrac g

PDS-based model checking, Static Verification may not scale to large programs

Interest in Dynamic Analysis based on executions

- ➢ **Dynamic Analysis Methods**
  - – **Preemptive context bounding**
  - – **Predictive analysis**
  - – **Coverage-guided systematic testing**
- ❑ Conclusions

Testing Multi-threaded Programs


Test
Input
Multithreaded C/C++ Program
Heap (storing shared objects)
Main thread
T1
T 2
T3
POSIX Threads Library (Pthreads)
Rest of the Linux OS

User expectation:

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

The reality:

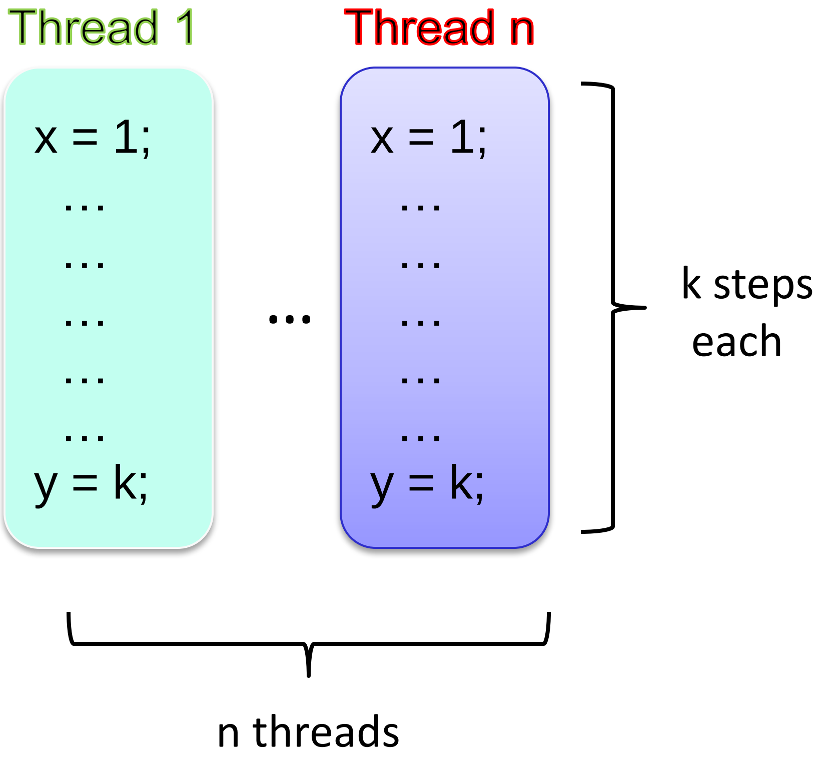
Even if the program may fail (under a certain schedule),
the user likely won't see it

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

State space explosion in all interleaved executions



Thread 1

x = 1;
…
…
…
…
…
y = k;

…

Thread n

x = 1;
…
…
…
…
…
y = k;

k steps each

n threads

- ❑ **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)

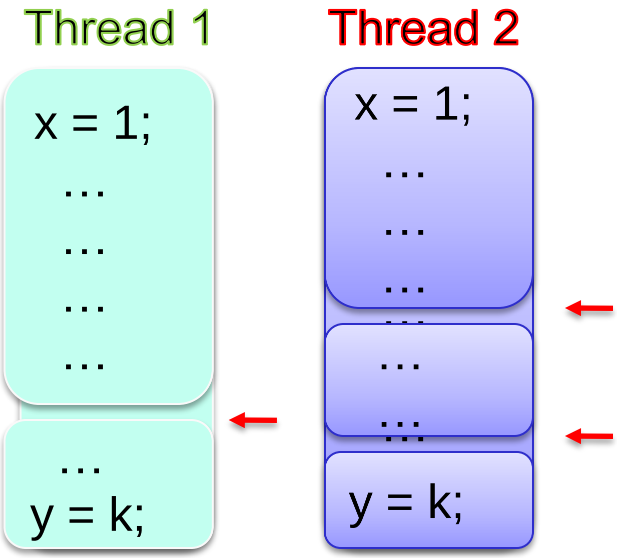
[Musuvathi et al. PLDI 07, OSDI 08]

- □ **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 <= ${}_{nk}C_c \cdot (n+c)!$**

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

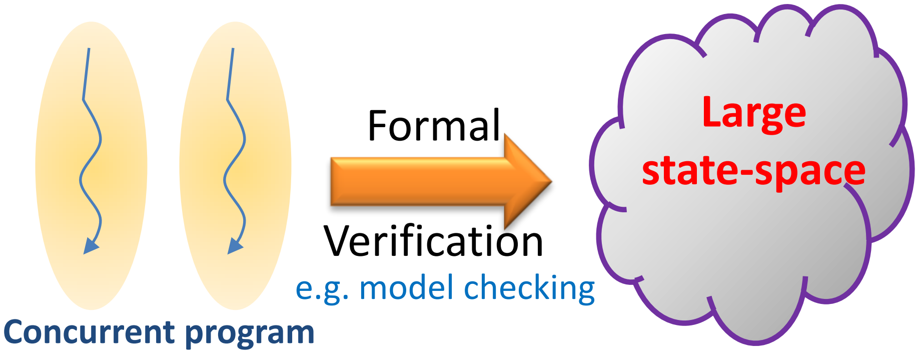
Exponential in n and c, but not in k



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

Many bugs found in a small number of preemptions

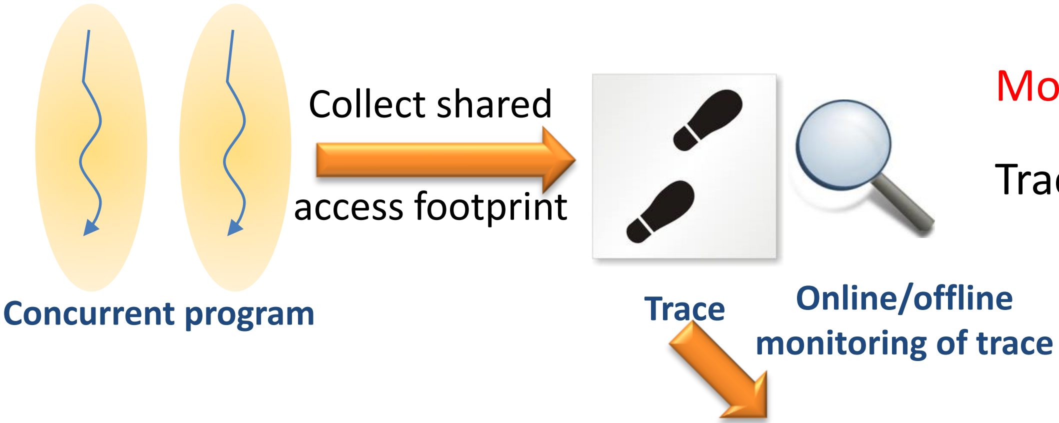
Motivation: Trace Based Verification



Full formal verification is often intractable

Will not talk about this

Alternate approaches



Monitoring problem

Tractable and no false alarms.

Next

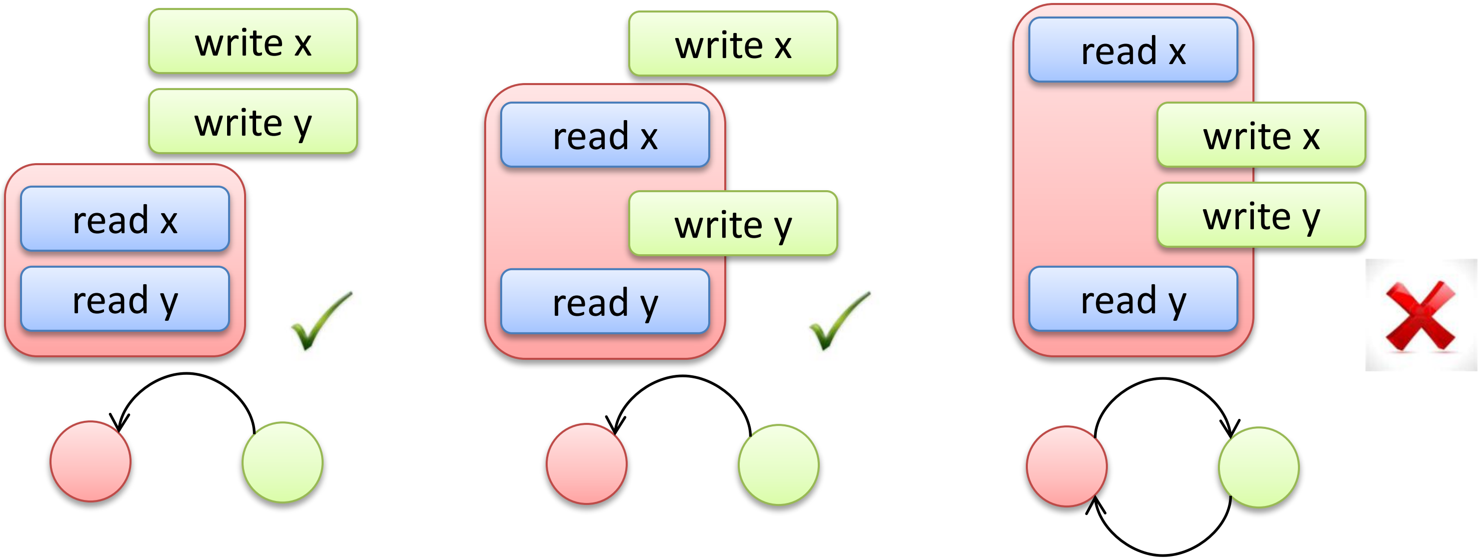
Predictive Analysis problem

Larger set of interleavings is explored.

Predict errors in alternate interleavings

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 (based on traces)

- **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**

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

t_0: x=0,y=0;
t_1: fork(1)
t_2: fork(2)

t_{11}: a=y;
t_{12}: assume(a=0)
t_{13}: x=1;
t_{14}: a=x+1;
t_{15}: x=a;
t_{18}:

t_{21}: b=x;
t_{26}: assume(b$\neq$0)
t_{27}: y=0;
t_{28}:

t_3: join(2)
t_4: join(1)
t_5: assert($x \neq y$);

C program:
multi-threaded,
using *Pthreads*

Execution trace

Symbolic Predictive Model

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

Symbolic Predictive Analysis for Detection of Violations

- **Build a SAT formula (in some quantifier-free first-order logic)**
  - **F_program : a feasible thread interleaving of CTP**
  - **F_property : e.g. an assertion is violated**

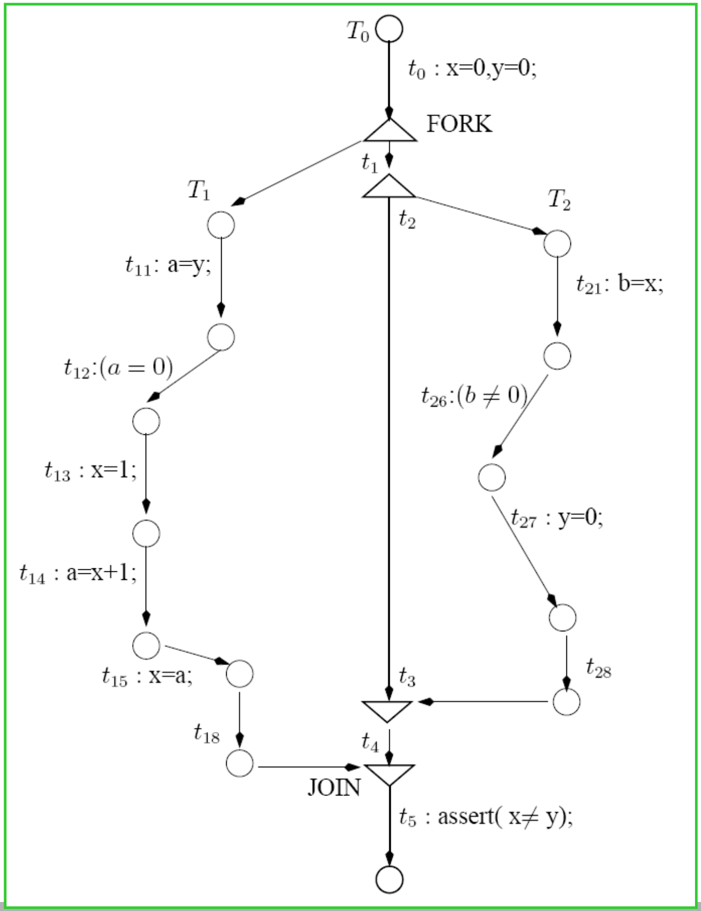
- **Solve using an SMT solver**

  (F_program && F_property)

  Sat → found a real error

  Unsat → no error in any interleaving

- **Improves precision over other predictive techniques, while providing coverage over all possible interleavings over the observed events.**



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:**
  HaPSet (History-aware Predecessor Set)

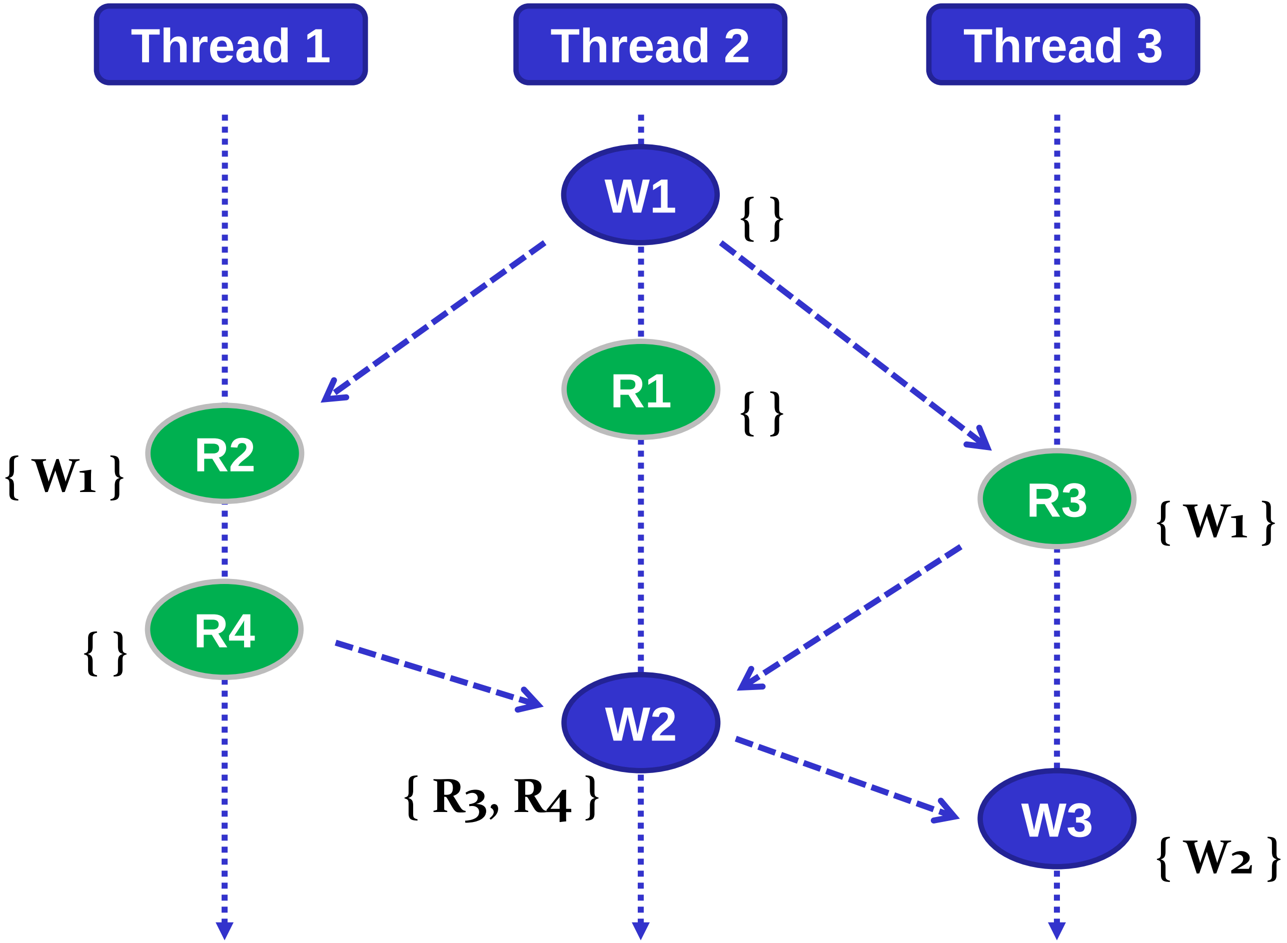
- ❑ **How do we use this metric?**
  - **Use a framework for systematically generating interleavings**
    - **e.g. stateless model checking**
  - **Keep track of HaPSets covered so far**
  - **Instead of DPOR/PCB, use HaPSet to prune away interleavings**
  - **Idea: Don't generate an interleaving to 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}
```

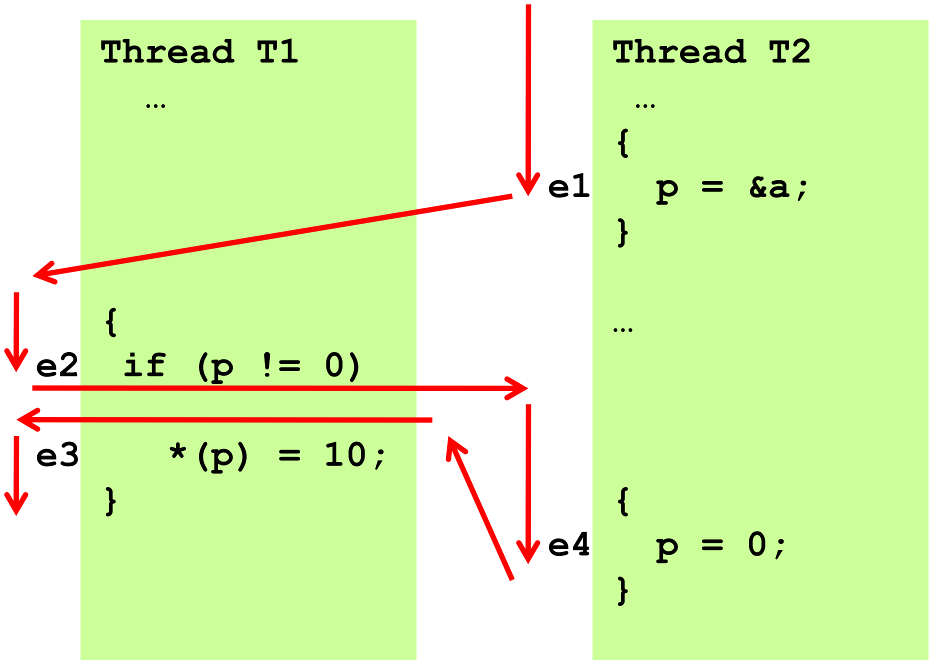
From all good runs

```
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?



```
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]

From all (good and bad) runs

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

Steer search directly to a "bad" run

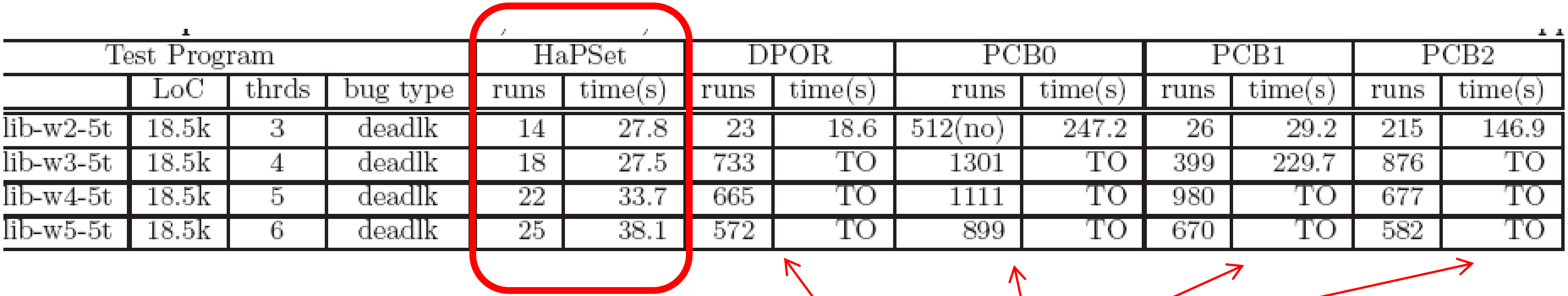
Does HaPSet Guided Search Work?

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

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

HaPSet guided search

Much faster than DPOR or 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**
  - – **Concurrency is pervasive, and very difficult to verify**
  - – **Active area of 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**

- □ **Hierarchy of Practical Challenges**
  - – **Multi-core systems, Many-core systems**
  - – **Distributed systems**

  - – **Great opportunity due to continuing growth of networked multi-core systems**