

Model Checking Concurrent Programs

Aarti Gupta

agupta@nec-labs.com

Systems Analysis and Verification

NEC Laboratories
America

Relentless passion for innovation

<http://www.nec-labs.com>

Acknowledgements

❑ **NEC Systems Analysis & Verification Group**

- Gogul Balakrishnan
- Malay Ganai
- Franjo Ivancic
- Vineet Kahlon
- Weihong Li
- Nadia Papakonstantinou
- Sriram Sankaranarayanan
- Nishant Sinha
- Chao Wang

❑ **Interns**

- Himanshu Jain, Yu Yang, Aleksandr Zaks, ...

Motivation

□ Key Computing Trends



Mobile

Server

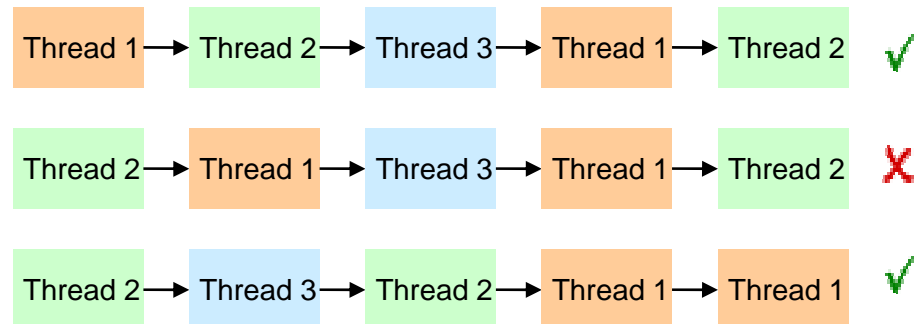
Gaming

High Performance, Low Power

- **Single core solutions don't work**
- **Need multi-core solutions**
- ***Need multi-core programming***

□ Parallel/Multi-threaded Programming

- **Difficult to get right**
 - **manual parallelization**
 - **dependencies due to shared data**
- **Difficult to debug**
 - **too many interleavings of threads**
 - **hard to reproduce schedule**



Goal: Improve SW productivity in the development of concurrent programs

- Find concurrency bugs using powerful program verification & analysis techniques: data races, deadlocks, atomicity violations
- Assist code understanding of concurrency aspects

Outline

- ❑ **Background**
- ❑ **Model Checking Concurrent Programs**
 - **Results for Interacting Pushdown Systems**
- ❑ ***Practical* Model Checking of Concurrent Programs**
 - **Four main strategies**
- ❑ **ConSave Platform**
- ❑ **Summary & Challenges**

Automatic Property Verification

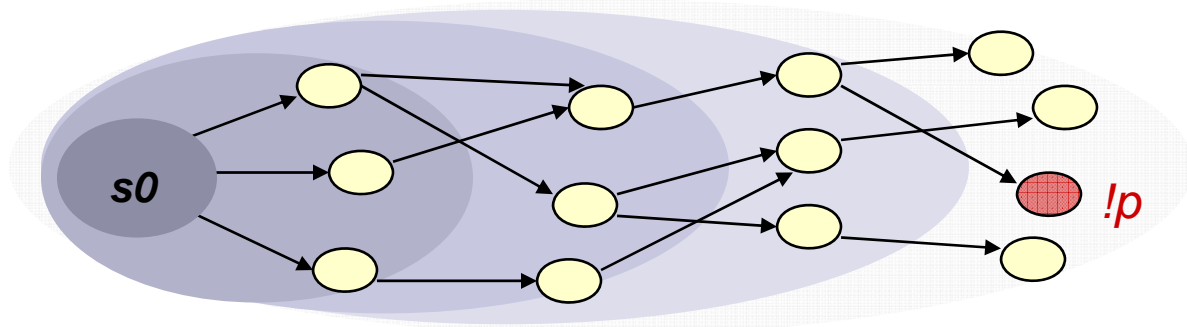
□ Verification Approach: e.g. Model Checking

- Exhaustive state space exploration
- Maintains a representation of visited states (explicit states, BDDs, ckt graphs ...)
- **Expensive**, need abstractions and approximations

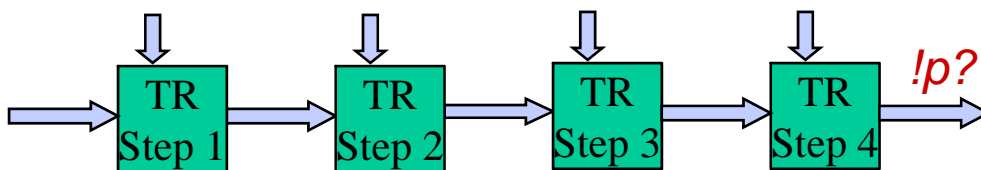
□ Falsification Approach: e.g. *Bounded Model Checking*

- State space search for bugs (counter-examples) or test case inputs
- Typically does not maintain representation of visited states
- Less expensive, but **need good search heuristics**

Model Checking $AG\ p$
Does the set of states
reachable from s_0
contain a bad state(s)?



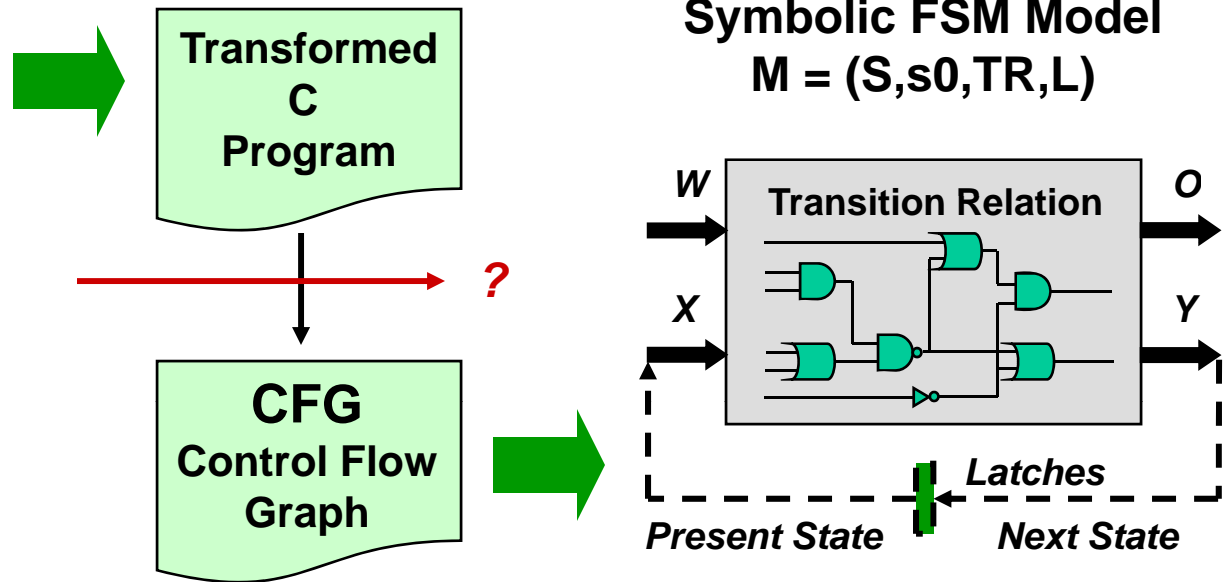
Bounded Model Checking
Is there is a path from
the initial state s_0
to the bad state(s)?



Extracting Program Models

C Program

```
1: void bar() {  
2:   int x = 3 , y = x-3 ;  
3:   while ( x <= 4 ) {  
4:     y++ ;  
5:     x = foo(x);  
6:   }  
7:   y = foo(y);  
8: }  
9:  
10: int foo ( int l ) {  
11:   int t = l+2 ;  
12:   if ( t>6 )  
13:     t -= 3;  
14:   else  
15:     t --;  
16:   return t;  
17: }
```



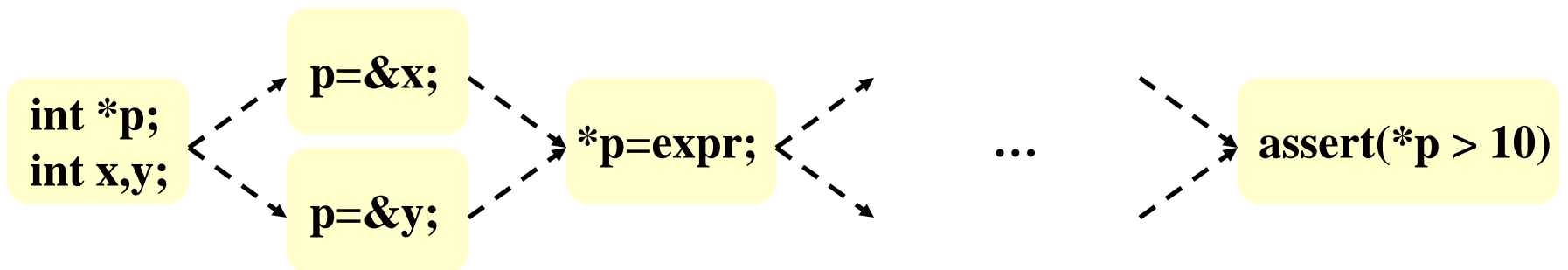
❑ Source-to-source transformations

- For modeling pointers, arrays, structures ...
- For automatic instrumentation of checkers

❑ Control Flow Graph: Intermediate Representation

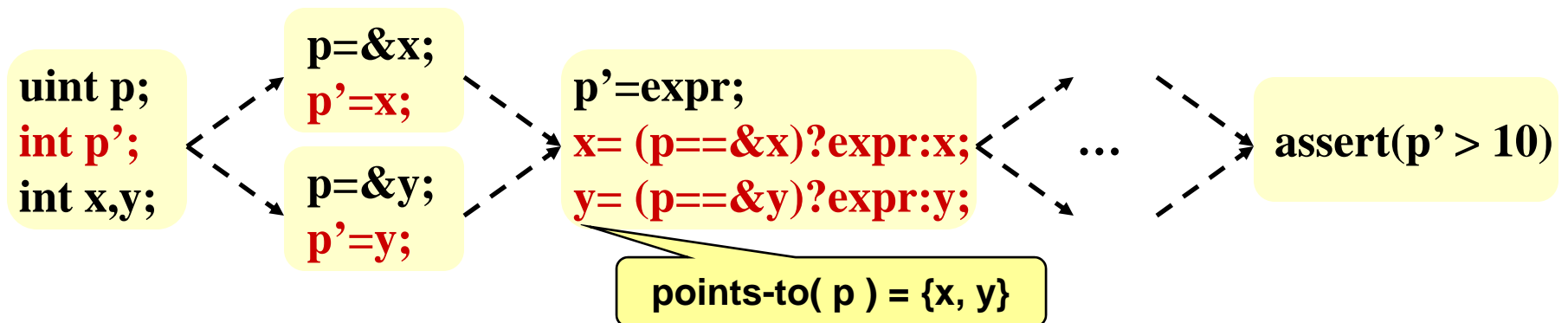
- Well-studied optimizations provide simplification and reduction in size of verification models
- Allows separation of model *building* phase from model *checking* phase

Modeling Pointers (src-to-src transformations)



□ Pointers replaced by auxiliary variables (introduce p' to track $*p$)

Reads/writes transformed to conditional assignments [Semeria & De Micheli 98]



□ Requires sound pointer analysis to derive sound points-to sets

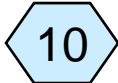
- We use fast (flow/context insensitive) Steensgaard pointer analysis
- Can add Andersen's analysis or context-sensitivity also [Kahlon PLDI 08]

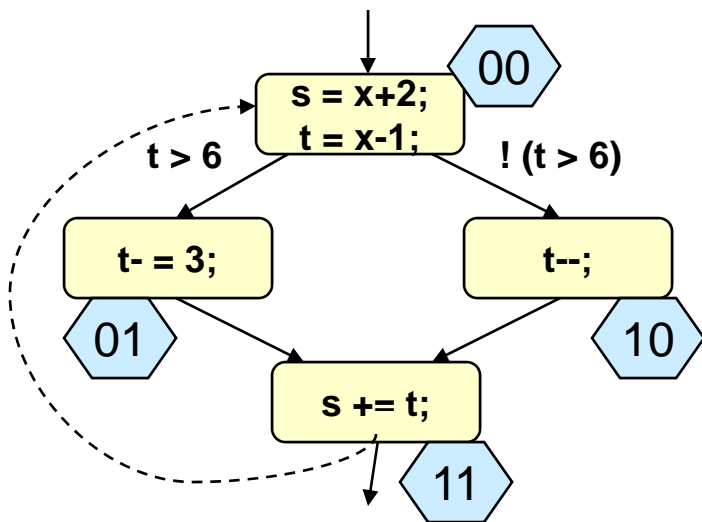
Translations of CFG to Symbolic Models

□ Our target for model checking: Finite state verification model

- Recursive functions are also modeled using a *bounded* call stack
 - Alternative: Boolean programs [Ball & Rajamani 01]
- Recursive data structures are *bounded* up to some user-chosen depth

□ This yields a CFG with only `int` type data variables, i.e. a numeric program

- Program Counter (PC) variables are used to encode basic blocks 
- Each data variable is interpreted as:
 - a **vector of state-bits** for *bit-precise* SAT- or SMT-based model checking
 - an **infinite integer** for static analysis or polyhedra-based model checking



CFG => Finite State (control + data) Machine

Basic blocks => control states (PC variables)

Program variables => data states

Guarded transitions => TR for control states

Parallel assignments => TR for data states

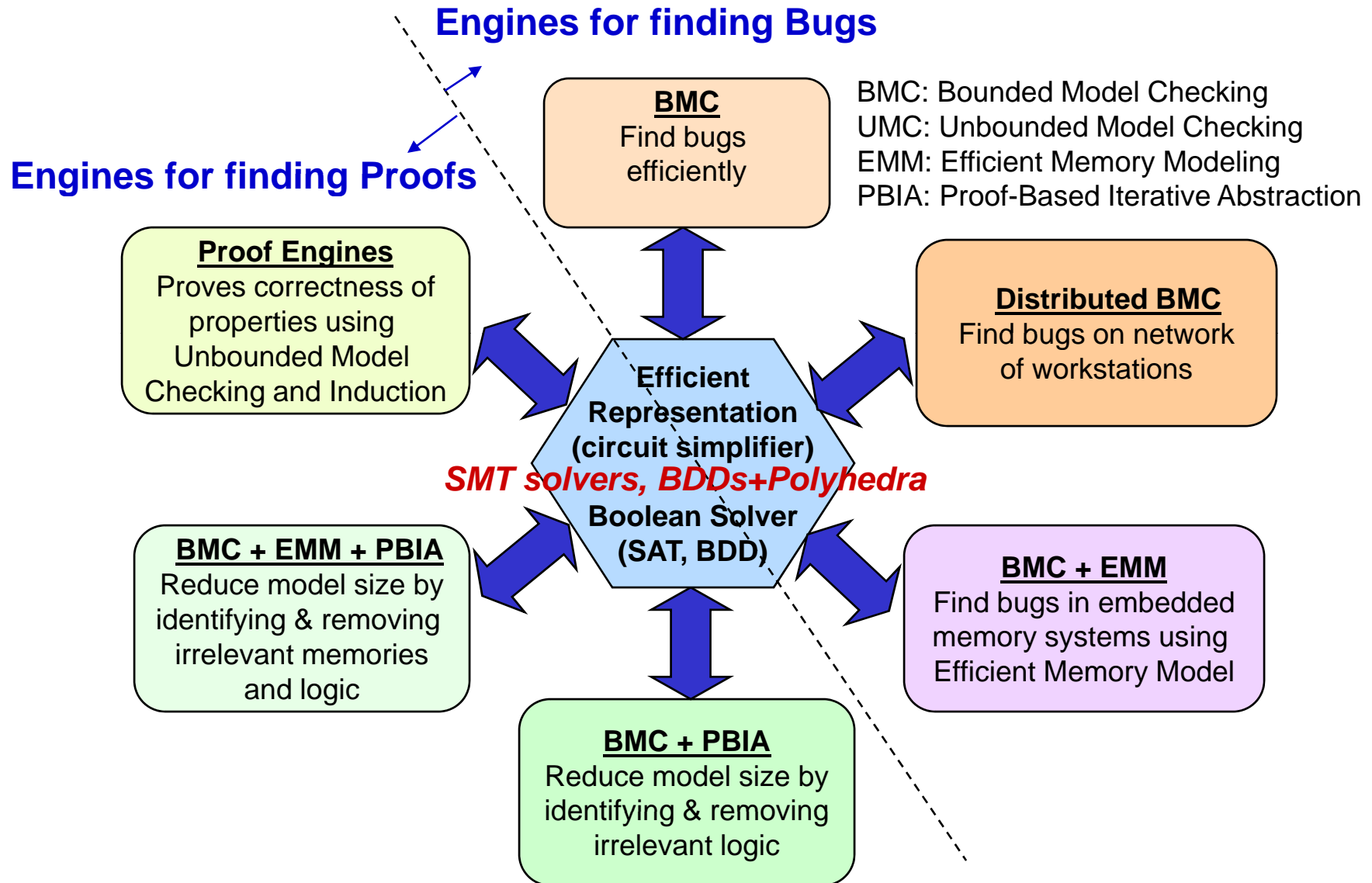
Loop back-edges => transitions between control states

FSMs: **Bit-precise accurate models**

Extended FSMs: finite control, but infinite data (integers)

VeriSol Model Checking Platform

[Ganai *et al.* 05], [Gupta *et al.* 06]



Dataflow Analysis for Concurrent Programs

- ❑ Close relationship between Dataflow Analysis for sequential programs and the model checking problem for Pushdown Systems (PDS)
[Schmidt, Bouajjani *et.al.*, Walukeiwicz]
- ❑ Various extensions of the basic PDS framework have been proposed leading to useful generalizations of the basic dataflow analysis framework
[Reps, Schwoon, Jha *et al.*]
- ❑ Analogous to the sequential case, dataflow analysis for concurrent programs reduces to the model checking problem for interacting PDS systems
- ❑ However, reachability is undecidable for PDSs with Pairwise Rendezvous
[Ramalingam 01]

Model Checking for Interacting PDS

- ❑ **Reachability is undecidable for PDSs with Pairwise Rendezvous**
- ❑ **How to get around the undecidability barrier?**
 - Give up precision
 - **Thread-modular reasoning**
 - **Over-approximation techniques** [Chaki *et al.* 06]
 - Restrict the synchronization/communication models
 - **PA processes** [Esparza and Podelski]
 - **Constrained Dynamic Pushdown Networks** [Bouajjani *et al.*]
 - **Asynchronous Dynamic Pushdown Network** [Bouajjani *et al.*]
 - Give up soundness
 - **Bounded number of context switches** [Qadeer & Rehof 05, CHESS]
 - **Dataflow analysis for bounded-context systems** [Reps *et al.*]
- ❑ **We focus on patterns of synchronization primitives**
 - In practice, recursion and synchronization are relatively “well-behaved”
 - Decidable for PDSs interacting via nested locks [Kahlon, Ivancic & G 05]
 - Undecidable for PDSs interacting via non-nested locks

Model Checking Double-indexed LTL Properties

[Kahlon & G, POPL 07]

- ❑ For $L(F,G)$ and $L(U)$ the model checking problem is undecidable even for system comprised of non-interacting PDSs
 - For decidability, restriction to fragments $L(G, X)$ and $L(X, F, \text{inf}F)$
- ❑ For PDSs interacting via nested locks the model checking problem is decidable only for the fragments
 - $L(G, X)$
 - $L(X, F, \text{inf}F)$
- ❑ For PDSs interacting via
 - Pairwise rendezvous, or
 - Asynchronous rendezvous, or
 - Broadcaststhe model checking problem is decidable only for $L(G, X)$

Practical Model Checking of Concurrent Programs

- ❑ In addition to state space explosion (as in sequential programs)
the complexity bottleneck is exhaustive exploration of interleavings
- ❑ Multi-pronged approach for handling interleavings
 - Avoid interleavings altogether
 - Thread-modular reasoning
 - *Rely on decomposition results for nested locks* **Strategy 1**
 - Avoid redundant interleavings
 - Partial Order Reduction (POR)
 - *Combine POR with symbolic model checking* **Strategy 2**
 - Semantic/Property-based reduction in interleavings
 - *Derive invariants using abstract interpretation* **Strategy 3**
 - *Use property-driven pruning* **Strategy 4**
- ❑ These are (mostly) orthogonal to other techniques
 - Shape analysis, Bounded context analysis, Stateless model checking, ...

Strategy 1: Avoid Interleavings by Decomposition

A concurrent multi-threaded program uses locks in a nested fashion iff along every computation, each thread can only release that lock which it acquired last, and that has not yet been released

❑ Example

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

f -> g: nested locks

f -> h: non-nested locks

- ❑ Programming guidelines typically recommend that programmers use locks in a nested fashion
- ❑ Locks are guaranteed to be nested in Java_{1.4} and C#

Acquisition History: Motivation

```
Thread1 ( ) {  
    f1: acquire(a);  
    f2: acquire(c);  
    f3: release(c);  
    f4: Error1;  
}  
  
Thread2 ( ) {  
    g1: acquire(c);  
    g2: acquire(a);  
    g3: release(a);  
    g4: Error2;  
}
```

- ❑ **Question:** Is it possible to reach Error states simultaneously?
- ❑ **Answer:** f_4 and g_4 are not simultaneously reachable even though $\text{Lock-Set}(f_4) \cap \text{Lock-Set}(g_4) = \emptyset$ [Savage *et al.*]
- ❑ **Tracking Lock-Sets is not enough**

Acquisition History: Definition

Thread1 () {	Thread2 () {
f_1 : acquire(a);	g_1 : acquire(c);
f_2 : acquire(c);	g_2 : acquire(a);
f_3 : release(c);	g_3 : release(a);
f_4 : Error1;	g_4 : Error2;
}	}

- ❑ The **acquisition history** of a lock k at a control location of thread T is the set of locks acquired by T since the last acquisition of k by T
 - Acq-Hist (f_4 , a) = {c}
 - Acq-Hist (g_4 , c) = {a}
- ❑ Acq-Hist(f_1 , k_1) is **consistent** with Acq-Hist(g_2 , k_2) iff the following does not hold:
 $k_1 \in \text{Acq-Hist}(g_2, k_2)$ and $k_2 \in \text{Acq-Hist}(f_1, k_1)$
- ❑ Check on consistent Acq-Hist avoids circular dependencies that can lead to deadlocks, which make states unreachable

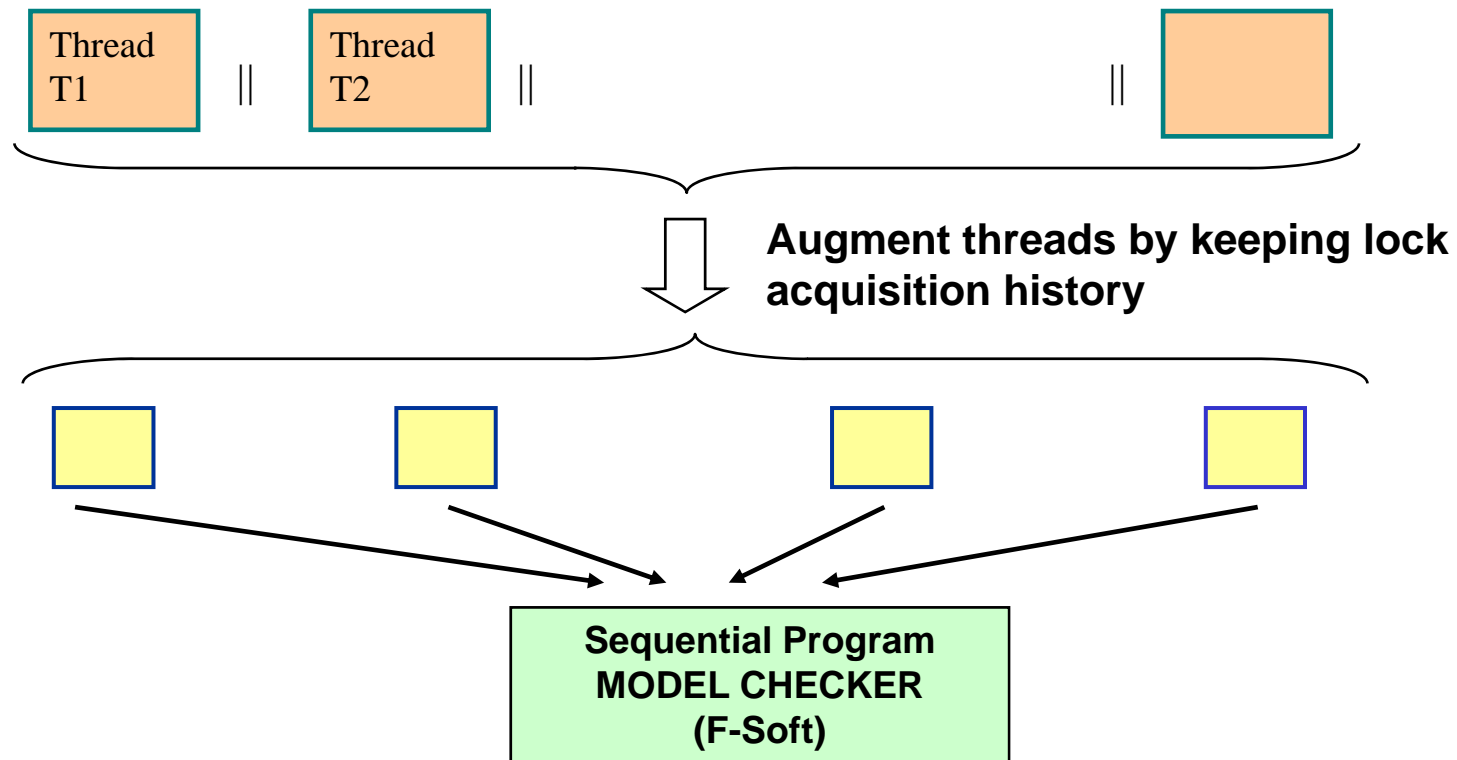
Decomposition Result for Nested Locks

[Kahlon *et al.* CAV 05]

- States c_1 and c_2 in **Thread1** and **Thread2** , respectively, are **simultaneously reachable** iff
 - $\text{Lock-Set}(c_1) \cap \text{Lock-Set}(c_2) = \emptyset$
 - There exists some path with **consistent acquisition histories** i.e., where there do not exist locks k and l such that :
 - $l \in \text{Acq-Hist}(c_1, k)$
 - $k \in \text{Acq-Hist}(c_2, l)$

- **Corollary: By tracking acquisition histories we can decompose model checking for a concurrent program to its individual threads**
 - Augment states with acquisition histories AH
 - Reachability: There exist consistent acquisition histories AH_1 and AH_2 such that the augmented local states (c_1, AH_1) and (c_2, AH_2) are reachable individually in T_1 and T_2 , respectively
 - Polynomial in number of states, exponential in number of locks
 - Context-sensitive static analysis results in small locksets and AHs

Model Checking by Decomposition

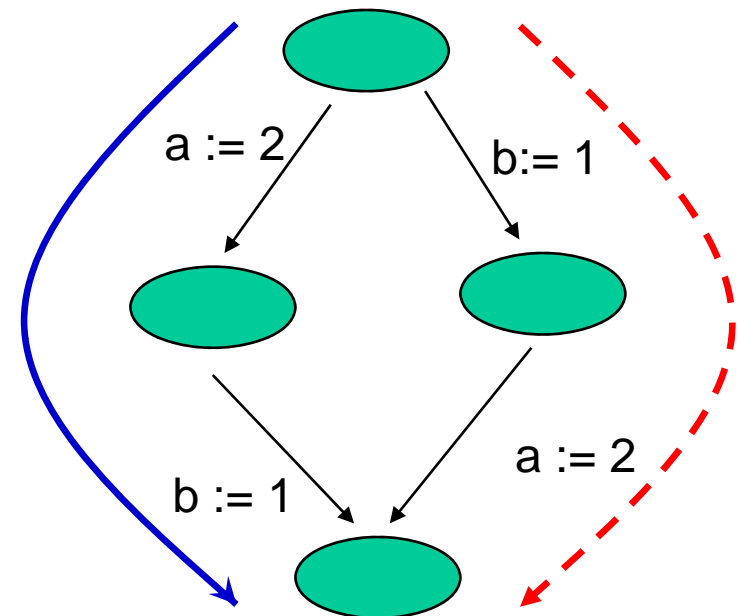


- ❑ **Reachability in multi-threaded program with *nested lock access* is reduced to model checking individual threads** [Kahlon *et al.* CAV 05]
 - Avoids state explosion arising due to concurrency
- ❑ **Model checking LTL properties for threads with nested locks** [Kahlon *et al.* LICS 06, POPL 07]

Strategy 2: Avoid Redundant Thread Interleavings

□ Partial Order Reduction (POR)

- Explore a restricted set of interleavings, ideally one from each equivalence class
- At each state, explore the set of **Persistent** transitions – the smaller the better
- Commonly used in explicit state model checking [SPIN, VeriSoft]



□ Transactions


[Lipton]

- Find atomic code regions (transactions), e.g. by lock analysis
- Consider context switches only at transaction boundaries [Stoller 02]

Persistent Sets using Acquisition Histories

Example

```
Thread1 ( ){  
  f1: acquire(a);  
  f2: acquire(c);  
  f3: release(c);  
  f4: ShVarAccess1;  
  f5: release(a);  
}  
  
Thread2 ( ){  
  g1: acquire(c);  
  g2: acquire(a);  
  g3: release(a);  
  g4: ShVarAccess2;  
  g5: release(c);  
}
```



- ❑ Consider global state (f₄, g₁)
- ❑ Lock-set (f₄) = {a}, Lock-set (g₁) = {c}
Transition from g₁ to g₂ is **included** in the persistent set based on Lock-sets
- ❑ However, there is no need for a context switch at f₄
Why?
- ❑ Thread2 cannot access ShVar at g₄ without Thread1 releasing lock a first
Thus the transition from g₁ to g₂ is **not included** in the persistent set

Bottomline

- ❑ Persistent sets based on Lock Acq-Hist are more refined than those based on Lock-sets [Kahlon, G. and Sinha, CAV 06]

Combining POR + Symbolic Model Checking

❑ Partial Order Reduction (POR)

- Avoid redundant interleavings
- Use acquisition histories to refine persistent sets

❑ Symbolic Model Checking (SMC)

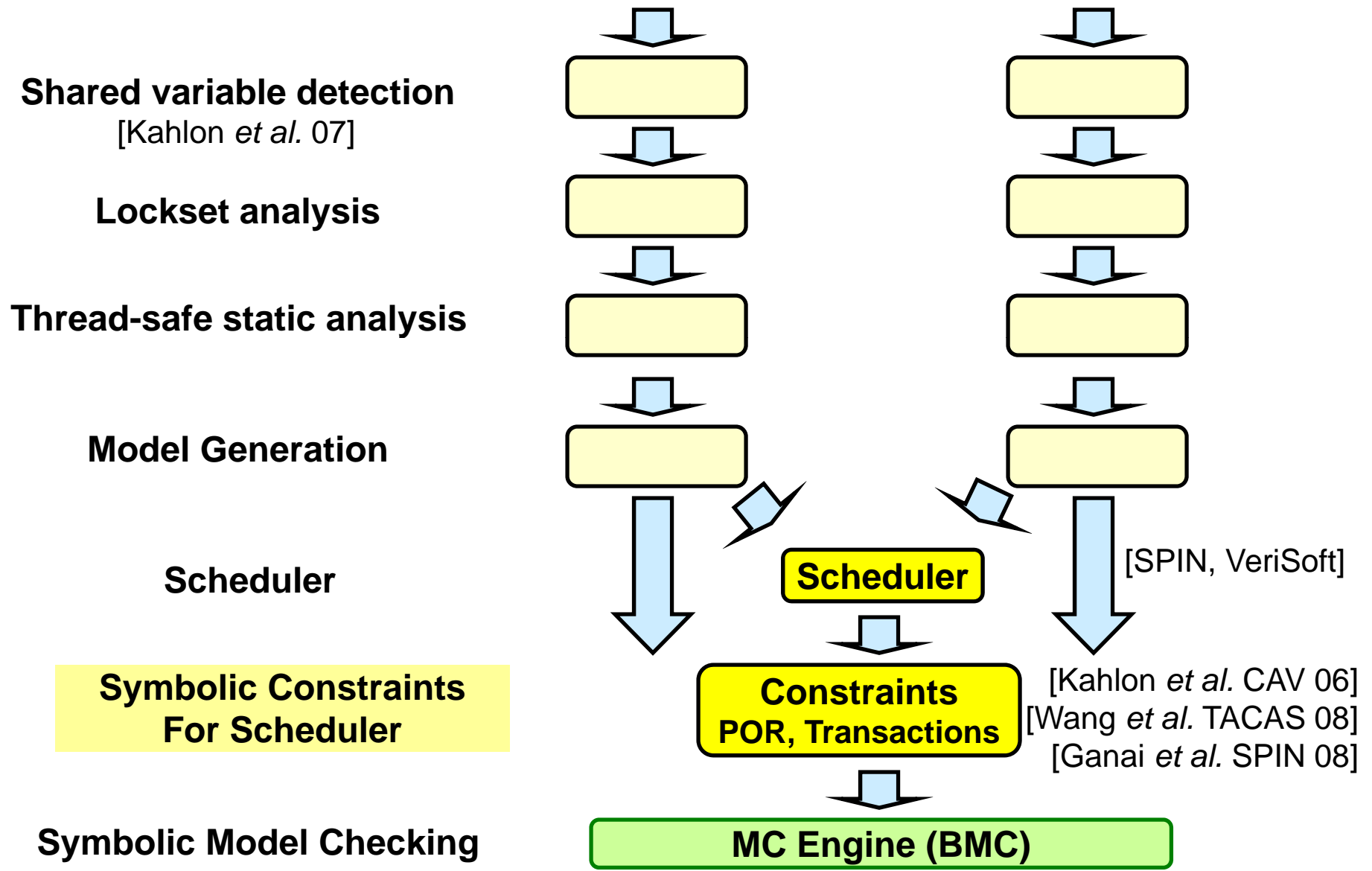
- Compact representation for large state spaces
- SAT, BDDs, SMT Solvers

Goal: To combine them in a synergistic manner

Implementation

- ❑ Build a circuit based model for each thread (as before)
- ❑ Use a scheduler that adds partial order + transaction constraints
- ❑ Carry out symbolic model checking using technique of choice
 - Separation of model building and verification stages allows flexibility

Generic Symbolic Model Checker Framework



Case study: Daisy file system

- Concurrent software benchmark
- 1 KLOC of C-like Java (manually converted to C)
- Simple data structures
- Fine-grained concurrency
- Variety of correctness properties

[Qadeer 04]

- Experimental results for finding 3 known races

[Kahlon *et al.* CAV 06]

SAT-based BMC with ...	Interleaved Execution	POR Reduction	POR + Transactions
Race₁	20 min 6.5 MB	3 sec 5.7 MB	1.4 sec 5.5 MB
Race₂	-	10 hrs 950 MB	12 min 517 Mb
Race₃	-	40 hrs 1870 MB	1.67 hrs 902 MB

Sound Reduction of Thread Interleavings

- ❑ **So far, identification of conflicts/transactions was done statically without considering dataflow facts**
 - Persistent transitions: if they access the same shared variable now, or sometime in the future
 - “Sometime in the future”: Usually over-approximated by reachability in CFG
 - May lead to too many thread interleavings

- ❑ **Strategy 3: Reduce number of thread interleavings by using concurrent dataflow analysis** [Kahlon *et al.* TACAS 09]
 - Reason about simultaneous reachability of global control states
 - Let static analysis perform more reductions, before model checker takes over

- ❑ **Strategy 4: Use Dynamic POR for precise information on conflicts**
 - Backtracks in DFS only if there is an actual conflict [Flanagan & Godefroid 05]
 - We further reduce number of backtracks by property-driven pruning [Wang *et al.* ATVA 08]

- ❑ **Note: These reductions are sound, unlike bounded analysis as in [CHESS]**

Strategy 3: 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  
}
```

**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
```

```
}
```

**No data race
Simultaneously unreachable
Due to locksets (plk)**

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  
}
```

No data race
Simultaneously unreachable
Due to wait-notify ordering constraint

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?

NO, due to invariants at these locations
pg_count is in $(-\infty, \text{LIMIT})$ in T1
pg_count is in $[\text{LIMIT}, +\infty)$ in T2
Therefore, these locations are not simultaneously reachable

How do we get these invariants?
Abstract Interpretation of course :)

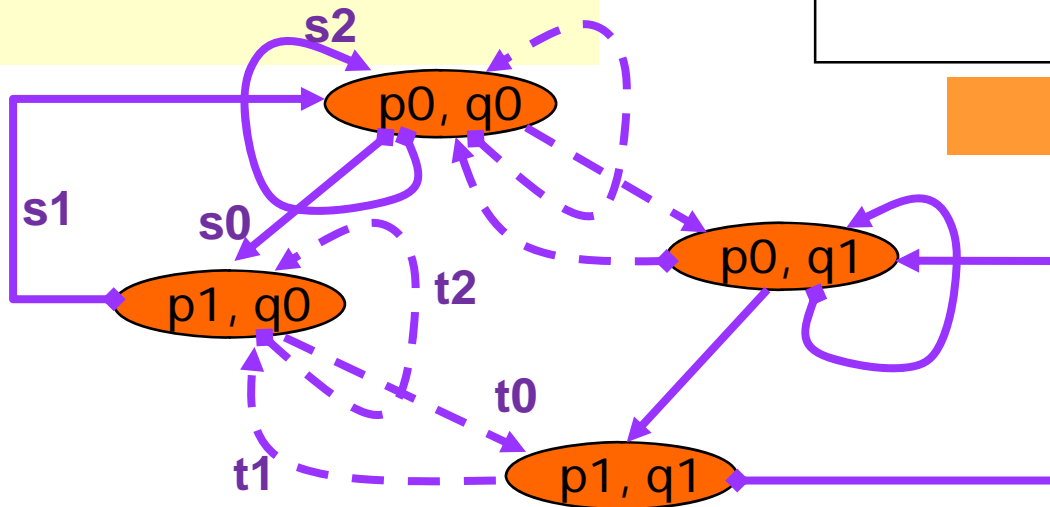
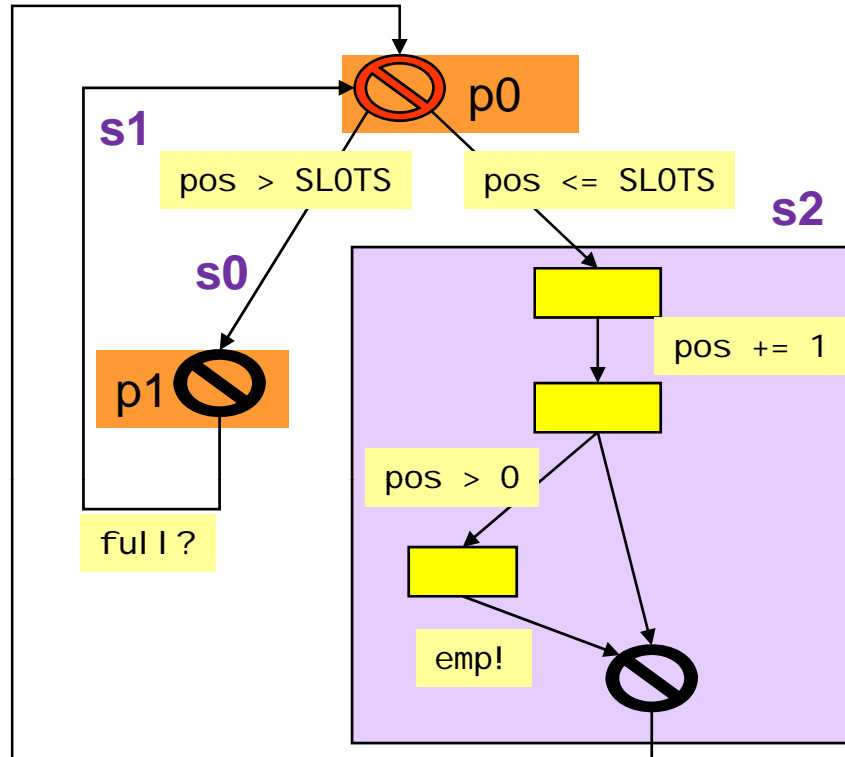
Transaction Graphs

- ❑ **Intuitively, a Transaction Graph is a product graph over control states**
 - Not all product (global) control states, keep only the *reachable* control states
 - An edge denotes an uninterruptible sequence of actions by a *single thread*
 - Note: What is uninterruptible depends on global state, not just local state
- ❑ **Two main (inter-related) problems**
 - How to find which global control states (nodes) are reachable?
 - How to find uninterruptible sequences of actions (transactions)?
- ❑ **We use an iterative approach (described next)**
 - **Unreachable nodes** ← 
 - > May lead to **larger transactions**
 - > Larger transactions correspond to **reduced interference (interleavings)**
 - > Reduced interference may lead to **more proofs of unreachability**
- ❑ **Use abstract interpretation over the transaction graph to find program invariants over the concurrent program**
 - Invariants are used to slice away parts of CFGs, leading to reduced interference

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

Iterative Refinement of Transaction Graphs

[Kahlon, Sankaranarayanan & G, TACAS 09]

□ Transaction Graph: Abstract Representation for Thread Interleavings

- At any stage, the transaction graph captures the set of interleavings that need to be considered for sound static analysis or model checking

□ Initial Transaction Graph

- Use static POR to consider non-redundant interleavings
 - Over control states only, need to consider CFL reachability
- Use synchronization constraints to eliminate unreachable nodes
 - For example, lock-based analysis, or wait-notify ordering constraints
 - Precise transaction identification under synchronization constraints: based on use of Parikh-bounded languages

[Kahlon 08]

□ Iterative Refinement

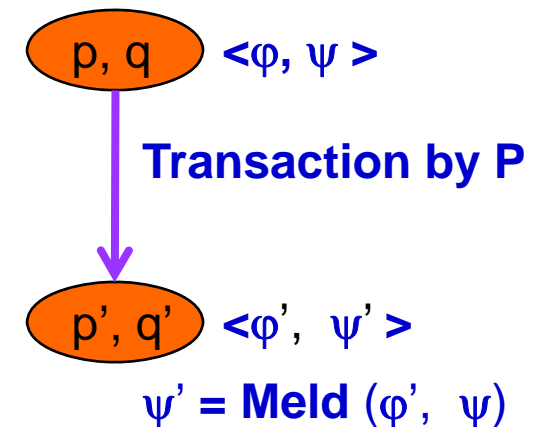
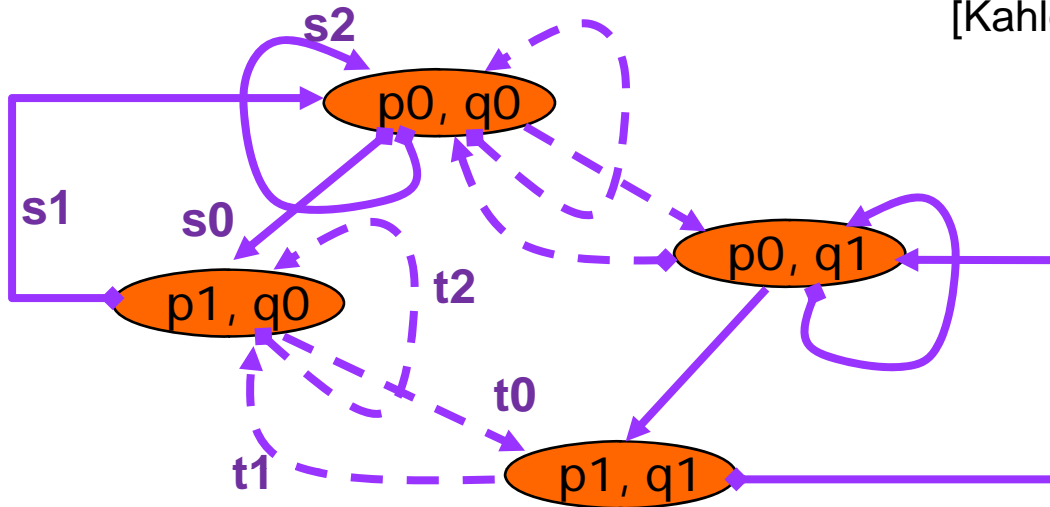
Repeat

- Compute range, octagonal, or polyhedral invariants over the transaction graph
- Use invariants to prove nodes unreachable and to simplify CFGs (slicing, ...)
- Re-compute transactions (static POR, synchronization) on the simplified CFGs

Until transactions cannot be refined further

Abstract Interpretation over Transaction Graphs

[Kahlon, Sankaranarayanan & G, TACAS 09]



- **Compute invariants $\langle \phi, \psi \rangle$ at each node $\langle p, q \rangle$**
 - ϕ holds over the state of thread P (shared + local)
 - ψ holds over the state of thread Q (shared + local)
- **$\langle \phi, \psi \rangle$ must satisfy the consistency condition over shared variables**
 - They must agree on values of the shared variables, i.e. $\phi|_{\text{shared}} \equiv \psi|_{\text{shared}}$
- **Basic operation: Forward propagation (post) over transactional edge**
 - Computed for each edge by sequential static analysis
- **Melding operator : for maintaining consistency**
 - After post-condition $\langle \phi, \psi \rangle \rightarrow \langle \phi', \psi' \rangle$, may also need to update ψ to ψ'
 - $\text{Meld}(\phi, \psi) = \psi'$, such that $\psi \subseteq \psi'$ and $\psi'|_{\text{shared}} \equiv \phi|_{\text{shared}}$

Application: Detection of Data Races

- ❑ Implemented in a tool called CoBe (Concurrency Bench)
- ❑ 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, and
 - at least one access is a write operation
- ❑ Phase 2: Static Warning Reduction
 - Create a Transaction Graph, and perform static reachability analysis
 - POR reductions, synchronization constraints, sound invariants
 - 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 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

After Phase 1 (Warning Generation)

After Phase 2 (Warning Reduction)

After Phase 3 (Model Checking)

CoBe Experiments

❑ Phase 3: Model Checking

- Individual Warnings: POR + BMC
- Found the known data races in 8 of 9 drivers (and some more ...)
- (Note: Did not have driver harnesses, so some of these may be false bugs)

Witness No.	Symbolic POR + BMC			Witness No.	Symbolic POR + BMC		
	Depth	Time (sec)	Mem (MB)		Depth	Time (sec)	Mem (MB)
jfs_dmap: 1	10	0.1	59	ctrace: 1	8	2	62
autofs_expire: 1	9	1.1	60	ctrace: 2	56	10 hr	1.2 G
autofs_expire: 2	29	128	144	ctrace: 3	92	2303	733
ptrace: 1	111	844	249	tty_io: 1	34	0.8	5.7
raid: 1	42	26.1	75	tty_io: 2	32	9.7	14
raid: 2	84	179	156	tty_io: 3	26	31	26
raid: 3	44	32.2	87	ipoib_multicast: 1	6	0.1	58
raid: 4	34	4.2	61	ipoib_multicast: 2	8	0.1	59
raid: 5	40	9.3	59	ipoib_multicast: 3	4	0.1	58
raid: 6	70	70	116	ipoib_multicast: 4	14	0.3	59

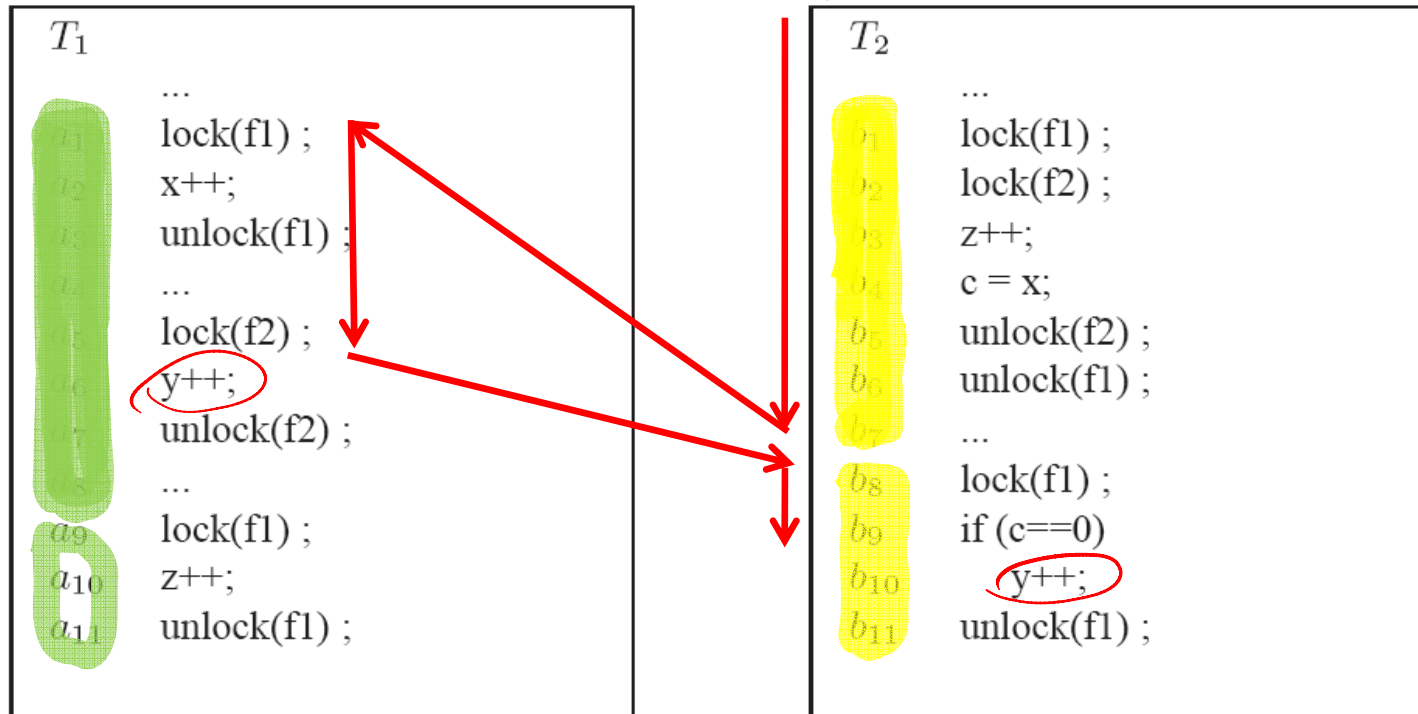
Practical Model Checking of Concurrent Programs

- ❑ In addition to state space explosion (as in sequential programs)
the complexity bottleneck is exhaustive exploration of interleavings
- ✓ Multi-pronged approach for handling interleavings
 - ✓ Avoid interleavings altogether
 - ✓ Thread-modular reasoning
 - ✓ *Rely on decomposition results for nested locks* Strategy 1
 - ✓ Avoid redundant interleavings
 - ✓ Partial Order Reduction (POR)
 - ✓ *Combine POR with symbolic model checking* Strategy 2
 - ✓ Semantic/Property-based reduction in interleavings
 - ✓ *Derive invariants using abstract interpretation* Strategy 3
 - *Use property-driven pruning* Strategy 4
- ❑ These are (mostly) orthogonal to other techniques
 - Shape analysis, Bounded context analysis, Stateless model checking, ...

Strategy 4: Property-Driven Pruning

Where is the data race?

Initial state: $x=y=z=0$

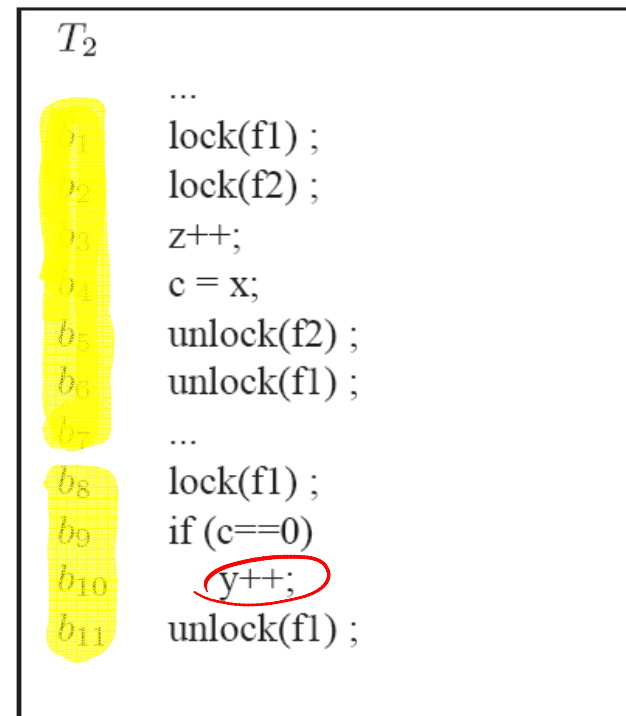
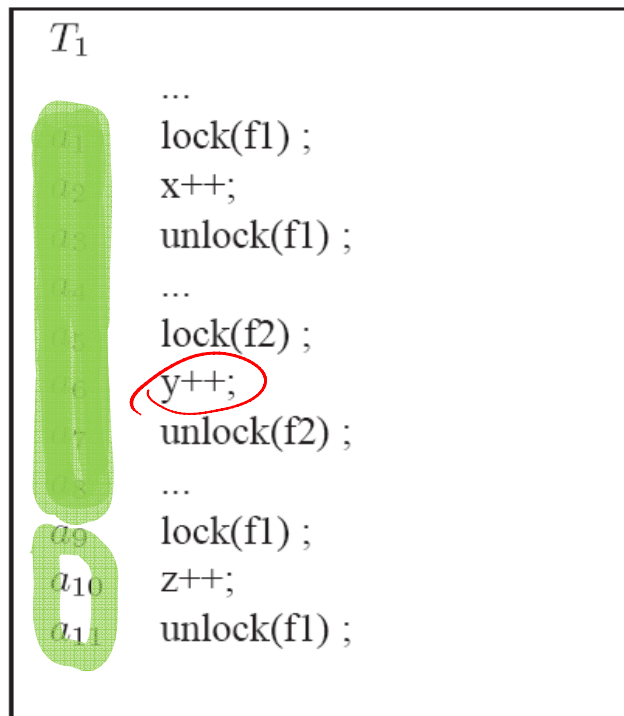


Error trace: $b1-b7, a1-a4, a5, b8-b9, a6, b10$

Both are enabled
There is a data conflict

Motivating Example

How would DPOR find it? ... it would take a while.



Traces: a1-a4, a5-a8, a9-a11, b1-b7, b8-b11

DPOR
reduction

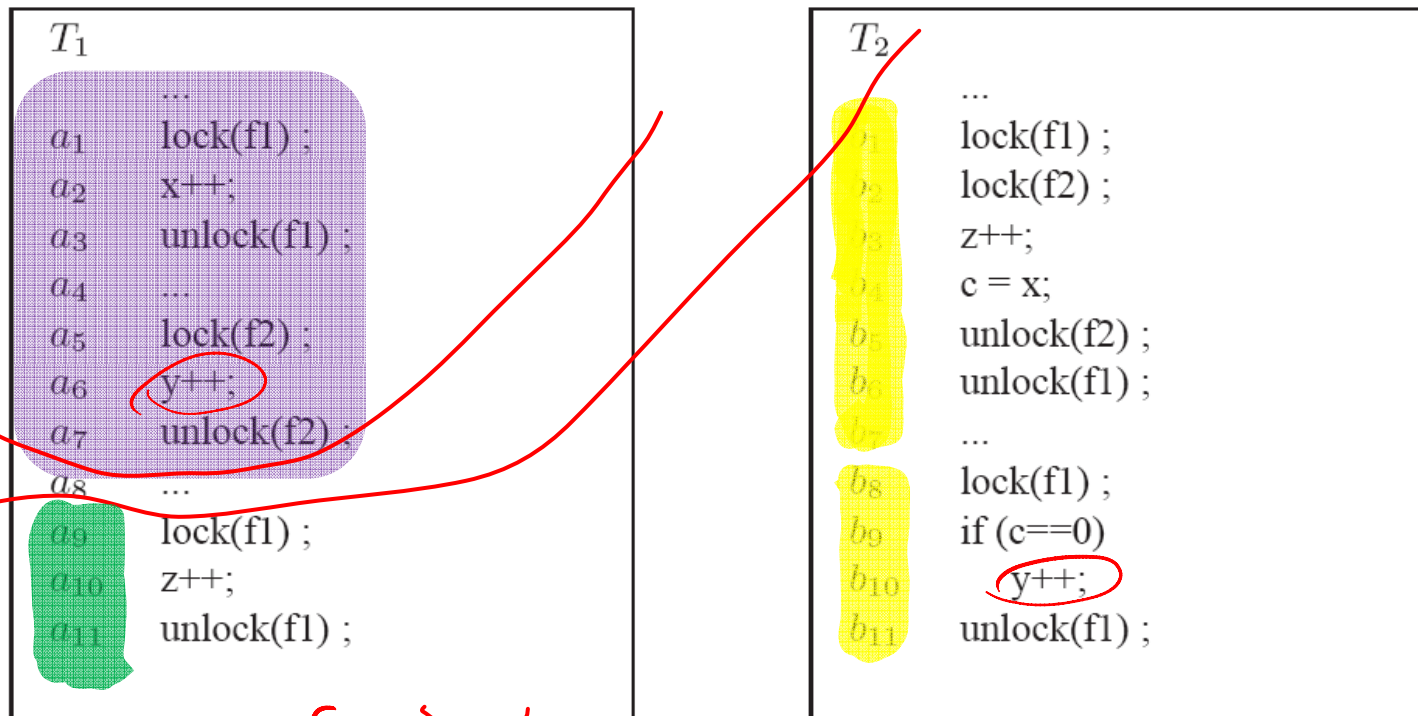
a1-a4, a5-a8, b1-b7, a9-a11, b8-b11
a1-a4, a5-a8, b1-b7, b8-b11, a9-a11
a1-a4,

Error: b1-b7, a1-a4, a5, b8-b9, {a6, b10}

systematic search
in a DFS order

Motivating Example

Can we do better than DPOR?



Traces: *prefix* { a1-a4, a5-a8, a1-a4, a5-a8, a1-a4, a5-a8, a1-a4, }

sub-tree { a9-a11, b1-b7, b8-b11, b1-b7, a9-a11, b8-b11, b1-b7, b8-b11, a9-a11 }

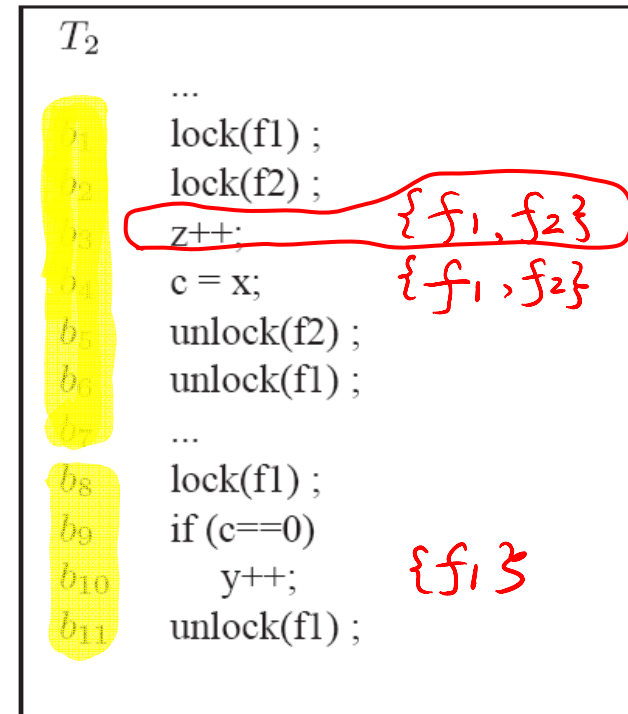
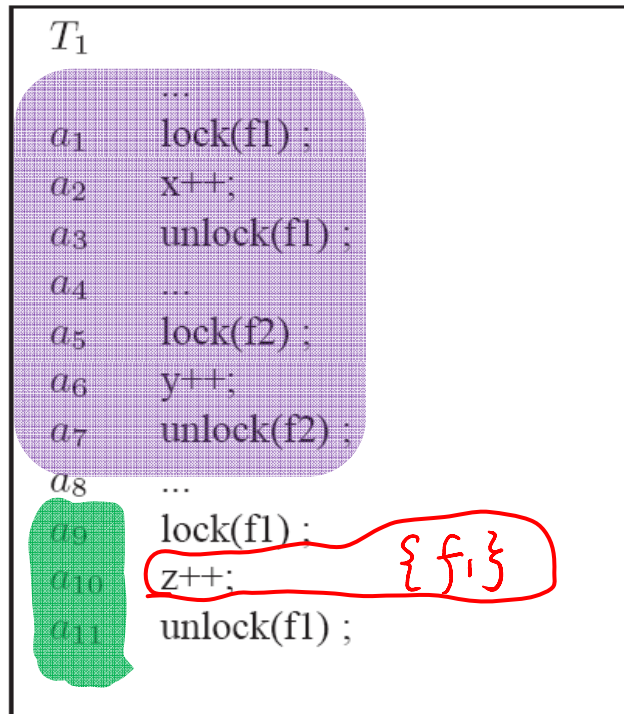
Error: b1-b7, a1-a4, a5, b8-b9, {a6, b10}

In this search sub-space,
a9-a11 and b1-b11
run concurrently

This sub-space does not
have data race!!!

Lockset Analysis: Is the sub-space race-free?

For each variable access, compute the *set of held locks (lockset)*



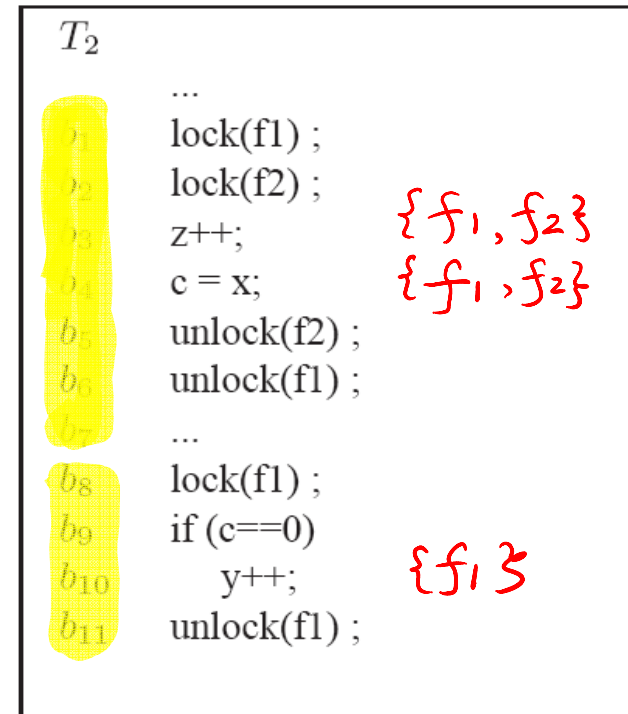
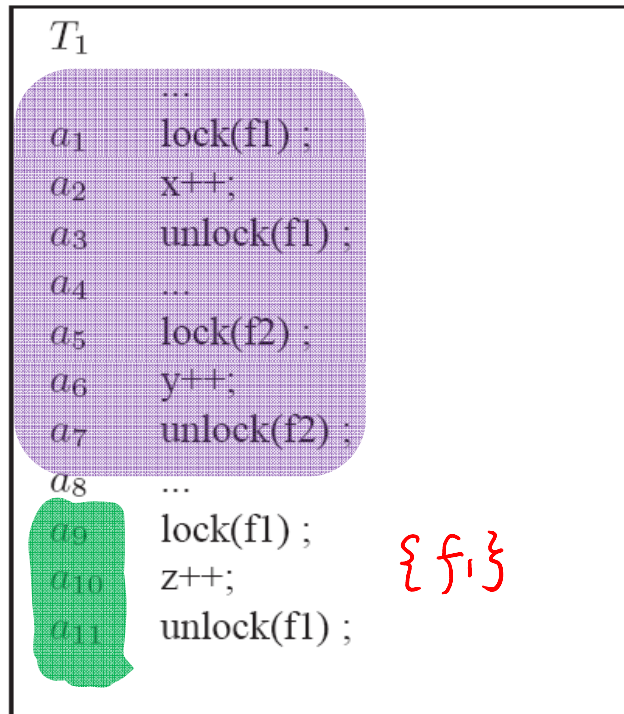
the intersection is not empty
→ can not be enabled at the same time

In this search sub-space,
a9-a11 and b1-b11
run concurrently

→ This sub-space does not
have data race!!!

Lockset Analysis: Is the sub-space race-free?

RaceFreeSubSpace: **Prune away equivalence classes that do not affect property**



Identifying the locksets is a thread-local computation → scalable

This reduction is beyond DPOR, but fits seamlessly with dynamic model checking

Property-Driven Pruning (PDP): Experiments

Test Program							Runtime (s)		# of Trans (k)		# of Traces		Race-free Chk		
name	loc	thrd	gvar	accs	lock	race	dpor	PDP	dpor	PDP	dpor	PDP	chks	yes	skip
fdrd2	66	2	3	3	2	1	3	1	2	0.6	89	14	88	75	75
fdrd4	66	2	3	3	2	1	11	3	10	4	233	68	232	165	165
qsort	743	2	2	2000	5	0	17	8	12	8	4	1	2	2	2
pfscan-good	918	2	21	118	4	0	179	15	71	10	2519	182	398	217	217
pfscan-bug	918	2	21	39	4	1	3	1	1	1	31	10	5	5	6
aget-0.4	1098	3	5	72	1	0	183	1	103	0.1	3432	1	6	6	9
aget-0.4	1098	4	5	78	1	0	>1h	1	-	0.1	-	1	9	9	18
aget-0.4	1098	5	5	84	1	0	>1h	1	-	0.1	-	1	12	12	30
bzip2smp	6358	4	9	18	3	0	128	3	63	2	1465	45	48	5	5
bzip2smp	6358	5	9	18	3	0	203	4	99	2	2316	45	48	5	7
bzip2smp	6358	6	9	18	3	0	287	4	135	2	3167	45	48	5	9
bzip2smp2	6358	4	9	269	3	0	291	136	63	21	1573	45	48	5	5
bzip2smp2	6358	5	9	269	3	0	487	155	85	21	2532	45	48	5	7
bzip2smp2	6358	6	9	269	3	0	672	164	116	21	3491	45	48	5	9
bzip2smp2	6358	10	9	269	3	0	1435	183	223	21	7327	45	48	5	17

#A #B

Reduction: (#A - #B)

Fusion: Dynamic Tests + Symbolic Analysis

- ❑ **Target: Property-driven learning and pruning with DPOR**
- ❑ **Execute target program under a thread schedule to generate a concrete trace (one interleaving)**
- ❑ **Symbolically analyze the concrete trace**
 - **CHECK**
 - Consider the **observed transitions** of the trace
 - Create a symbolic problem for checking **all feasible interleavings** of these transitions
 - **PRUNE**
 - Consider also **(the abstractions of) the unobserved branches**
 - Create a symbolic problem for checking **all feasible interleavings**
 - If no violation is possible, then **skip** the related backtrack point
- ❑ **Continue executing target program under another thread schedule to generate a concrete trace**
 - **Avoid enumerating thread schedules already considered**

Fusion: Dynamic Tests + Symbolic Analysis

Table 1. Comparing the performance of *Fusion* and *DPOR*

Test Program				Fusion (in C3)			DPOR (in Inspect)		
name	# threads	global-ops	property	executions	transitions	time (s)	executions	transitions	time (s)
fa02-1	2	21	false	1	32	0.2	34	545	6.6
fa02-5	2	73	false	1	84	0.8	190	8349	47.5
fa02-10	2	138	false	1	149	1.4	390	29904	108.6
pBch4-5	2	28	false	2	59	0.5	64	472	13.8
pBch4-10	2	48	false	2	89	0.6	274	2082	55.9
pBch4-20	2	88	false	2	149	1.3	1144	10842	248.7
pBch4ok-1	2	12	true	4	49	1.9	5	50	1.4
pBch4ok-3	2	28	true	11	211	6.9	152	1445	32.7
pBch4ok-4	2	36	true	18	385	19.6	1164	10779	255.8
pBch4ok-5	2	44	true	27	641	40.1	-	-	>3600

Putting it All Together: ConSave Platform

❑ Existing Solutions

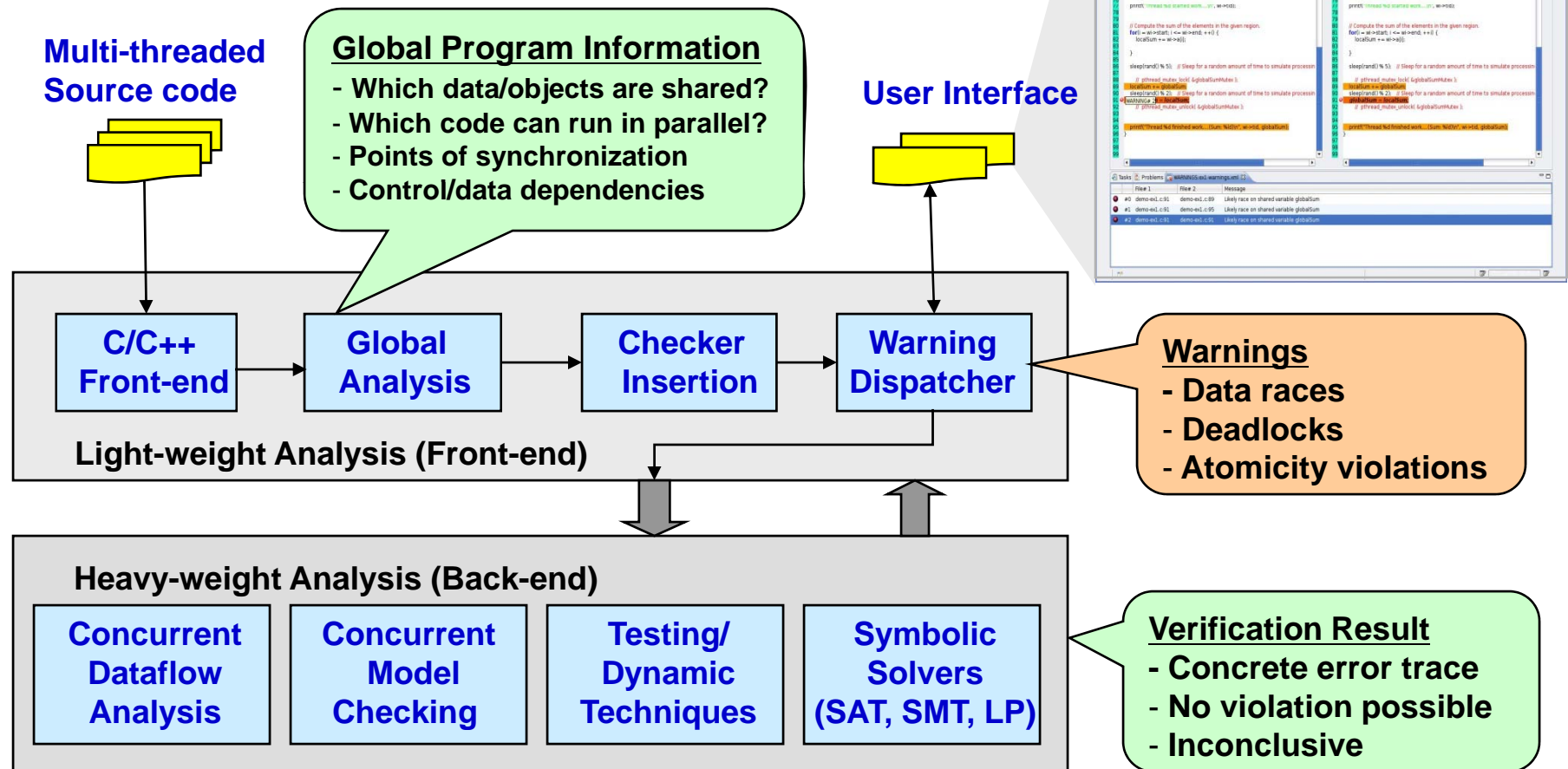
- Testing/dynamic verification: **poor coverage**
- Static program analysis: **too many bogus warnings**
- Model checking: **does not scale**

❑ ConSave: Cooperative, Staged Framework

- **Generate warnings cheaply, reduce warnings by staging analyses**
 - On-demand precise analysis
 - Precision supported by high performance SAT/SMT solvers
- **Highlights**
 - Dynamic testing/verification combined with symbolic analysis
 - Concurrent dataflow analysis w/automatic transaction identification
 - Partial order reduction with symbolic model checking

ConSave Platform for Concurrent Program Verification

- ❑ Light-weight front-end: warning generation
- ❑ Heavy-weight back-end: warning reduction



Summary and Other Challenges

❑ Concurrent program verification

- Concurrency is pervasive, and very difficult to verify
- Many promising technologies in formal methods
 - Testing/dynamic verification, Static analysis, Model checking, ...
 - Controlling complexity of interleavings is key
- Accuracy in models AND efficiency of analysis are needed for practical impact
 - Don't give up too early on large models, on precision
 - Advancements in Decision Procedures (SAT, SMT, ...) offer hope
- Great opportunity, especially with proliferation of multi-cores

❑ Better program analyses

- Pointer alias analysis, shared variable detection, ...
- Heap shapes and properties

❑ Modular component interfaces

- Required for scaling up to large systems (MLOC)
- Practical difficulties can be addressed by systematic development practices, but there should be a clear return on invested effort