Staged Concurrent Program Analysis*

Nishant Sinha
Joint work with Chao Wang
System Analysis and Verification Group,
NEC Labs, Princeton, NJ

Analyzing Concurrent Programs

- .. is HARD!
- Extensive work on analysis of Concurrent Programs
  - **Static** analysis: SPIN, Java Path Finder, ...
  - **Dynamic/Runtime** analysis: Verisoft, Eraser, CHESS, ...
  - **Combinations**: FUSION, ...
- Wide variety of bugs: data race, deadlock, assertion violation, atomicity violation, ...
This talk

- A new **staged symbolic analysis** technique
  - static analysis
  - analyze multiple paths, schedules and inputs *simultaneously*
  - find bugs
  - sometimes, absence of bugs too
  - rethinking from basics
Two Sources of Inefficiency

- Bi-modal Reasoning
  - alternating intra- and inter-thread reasoning
  - duplicated intra-thread reasoning

- Scheduler
  - does not model interference directly

T_1 :
  x = 3;
  t = x;
  a = t + 1;
  b = a + 3;
  assert(b > 4);

T_2 :
  ..... 
  ..... 
  ..... 
  x = 5;
  .....
Bi-modal Reasoning

Thread $T_1$

$L_1$: $a_0 = x$;
$a_1 = a_0 + 1$;
$z_1 = a_1$;
$a_2 = a_1 + 1$; $z_2 = a_2$;
... 
$a_{99} = a_{98} + 1$; $z_{99} = a_{99}$;
$a_{100} = a_{99} + 1$;
$L_2$: $x = a_{100}$;

Thread $T_2$

$L_3$: $x = 0$;
... 
$L_4$: $x = 5$;
... 
A: assert ($x == 5$ | | $x >= 105$);

- **Goal**: Infer ($a_{100} = a_0 + 100$) only once (not for each interleaving)
- **Path compression** methods only work inside atomic `transactions`

*Arrows show interference*
Scheduler

- Omnipresent in concurrent analysis
  - Explicit: context-switching
  - Symbolic: auxiliary variable \[ V_i (sch = i) \Rightarrow R_i \] 
- Does not model interference directly

```c
if (c) {
    *p = 0;
    p = 0;
}
c = true;
```

Context-bounding helps but is not property-driven
Background: Bounded Programs

- Verifying Concurrent Programs is not decidable
  - even with finite data (Boolean Programs)
- Our focus: **Bounded Programs**
  - Loops, Recursion unrolled finitely
  - therefore, bounded thread creation and heap
  - Real programs (not Boolean)
    - contain pointers, arrays, structures, etc.
    - may contain infinite datatypes (with decidable theory)
  - Decidable
  - Witnesses found are real but Proofs may be spurious
Program Representation

- Concurrent Control Flow Graph (CCFG)
  - Extension of sequential CFGs
  - Thread Fork, Join nodes
  - Functions modeled with call/return edges
  - Locks/Synchronization as shared variables
    - guarded assignments to model test-and-set

- Memory modeling
  - Compute shared location accesses using *flow-insensitive pointer analysis*
  - *Global* heap array + *Local* heap for each thread
  - Transform statements
    - one global access per statement
    - \( *p = l \leadsto \text{MemG}[p] = l \); (if \( p \) accesses a shared location)
int x;
void add_global ()
{
  if ( x < 1 ) x = x + 1;
  else x = x + 2;
}

int main (int argc, char *argv[])
{
  pthread_t t1, t2;
  x = 0;
  pthread_create(&t1, NULL, NULL, add_global);
  pthread_create(&t2, NULL, NULL, add_global);

  pthread_join(t1);
  pthread_join(t2);
  assert(x == 3);
}
Avoid Bi-modal Reasoning

- Obvious idea: Summarize each thread first!

- But, summarize in presence of concurrency?

```c
int x; //global
int func (int a) {
    if (a) return x;
    else return x + 1;
}

int func2 () {
    x = 3;
    ......
}

ret -> ite (a0, x0, x0+1)
```
Interference Abstraction

- Reading a shared location $x$ may not correspond to last write to $x$ in the same thread
  - *interfering* concurrent write to $x$

- Idea: **Interference Abstraction**
  - introduce a symbolic variable for each read
  - *decouple* reads and writes
  - couple them *later*

- Contrast with **state abstraction** at a program point by duplicating shared variables
  - e.g., translation to sequential program under context bounds
  - num of shared accesses $\times$ num of shared vars
  - Interference Abstraction: linear in the number of reads
Staged Concurrent Program Analysis

Interference-Modular Summarization (summary with only global accesses)

Compose Summaries (axioms encode interference between global accesses)

Check properties (Using a decision procedure)

- Without a scheduler
- Only inter-thread reasoning
- Find concrete property violations

Summarization involves only intra-thread reasoning
Stage 1: Summarization
Stage 1: Summarization

- Interference-Modular Summarization
  - do away precisely with local control and data flow
  - keep the reads and writes of shared variables intact

- Why?
  - avoid bi-modal reasoning
  - because only global accesses matter for inter-thread reasoning

- How?
  - Data flow analysis modulo Interference Abstraction
int x;
void add_global ()
{
    if ( x < 1 ) x = x + 1;
    else x = x + 2;
}

int main (int argc, char *argv[])
{
    pthread_t t1, t2;
    x = 0;
    pthread_create(&t1, NULL, NULL, add_global);
    pthread_create(&t2, NULL, NULL, add_global);

    pthread_join(t1);
    pthread_join(t2);
    assert(x == 3);
}
Access | loc | val | occ
--- | --- | --- | ---
W1 | @x | 0 | true
R1 | @x | r1 | true
R2 | @x | r2 | r1 < 1
W2 | @x | r2 + 1 | r1 < 1
R4 | @x | r4 | true
Example: Summary

Interference Skeleton (IS)
Summarization Rules

\[ \text{MemG}[l] = r; \]
\[ \text{Fresh access } A = (\Psi, l', v) \]
\[ \text{Add } E \rightarrow A \text{ to Skeleton} \]

Intra-Thread Join

\[ \text{Extends to standard Sharir-Pnueli, RHS style interprocedural analysis} \]
\[ \text{Function Summarization and Reuse} \]
Stage 2: Axiomatic Composition
Stage 2: Axiomatic Composition

- Interference Skeleton -> Feasible Program Executions?
  - need to couple the reads with writes
  - not via a scheduler!

- Idea: Compose Axiomatically
  - Axioms of Sequential Consistency (SC)
    - each read **must link** with **some write**
    - read must link with **last such write** in execution order
  - SC predominantly employed for straight line programs
    - how do we generalize to programs with branching?
Sequential Consistency Axioms

- Specified in **typed first-order logic**
  - read \( r \), write \( w \): Access type

- **Link** Predicate: \( \text{link} \left( r, w \right) \)
  - holds if \( r \) obtains value from write \( w \) in an execution
  - Exclusive: \( \text{link} \left( r, w \right) \rightarrow \forall w'. \neg \text{link} \left( r, w' \right) \)

- **Must-Happen-before** Predicate: \( \text{hb} \left( w, r \right) \)
  - \( w \) must happen before \( r \) in the execution
  - strict partial order
SC Axioms (contd.)

- \( \Pi = \Pi_1 \land \Pi_2 \land \Pi_3 \)
- \( \Pi_1 \) (must link some, only if occurs)
  - \( \forall r. \text{occ}(r) \iff \exists w. \text{occ}(w) \land \text{link}(r,w) \)
- \( \Pi_2 \) (local consistency)
  - \( \forall r, w. \text{link}(r,w) \Rightarrow (\text{loc}(r) = \text{loc}(w) \land \text{val}(r) = \text{val}(w) \land \text{hb}(w,r)) \)
- \( \Pi_3 \) (global consistency)
  - \( \forall r, w. \text{link}(r,w) \Rightarrow \forall w'. (\text{occ}(w') \land \text{hbet}(w, w', r)) \Rightarrow \text{loc}(w) \neq \text{loc}(w') \)

Incorporate \text{occ} predicate to handle branching
Instantiating Axioms

- Explicit instantiation for all reads and writes

- $\Pi_1 := occ(r) \leftrightarrow (occ(w_1) \land \text{link}(r_2, w_1))$
  $\lor occ(w'_2) \land \text{link}(r_2, w'_2)\ldots$

- $\Pi_2 := \text{link}(r_2, w'_2) \Rightarrow \text{loc}(r_2) = \text{loc}(w'_2) \land \text{val}(r_2) = \text{val}(w'_2)$
  $\land \text{hb}(w'_2, r_2)$

- $\Pi_3 := \text{link}(r_2, w'_2) \Rightarrow$
  $\text{hbet}(w'_2, w_2, r_2) \land \text{occ}(w'_2) \Rightarrow (\text{loc}(w'_2) \neq \text{loc}(r_2))$

- At most **cubic** in number of reads and writes
Efficient Encoding

- Employ UFs over theory of integers
  - avoid quantified axioms for \textbf{link} and \textbf{hb}
- Link Predicate:
  - \( \text{link} (r,w) \Leftrightarrow \text{ID} (r) = \text{ID} (w) \)
  - assign unique IDs to all writes
- Must Happen-Before Predicate
  - \( \text{hb} (w,r) \Leftrightarrow \text{Clk} (w) < \text{Clk} (r) \)
- Interference Pruning (few slides later)
Finding Bugs
Stage 3: Finding Bugs

- Data races, say between r, w
  - $\Phi_P := \neg \text{hb}(r, w) \land \neg \text{hb}(w, r)$

- Assertion Violation
  - $\Phi_P := \text{path condition for violation}$

- Full Encoding
  - $\Phi := \Phi_{IS} \land \Pi \land \Phi_P$
  - Discharged to an SMT solver

- Theorem: $\Phi$ is satisfiable iff property violated in the bounded program
Example

Goal: Detect NULL pointer access violation
- suppose the solver links Rp with Wp ($\Pi_1, \Pi_2$)
- and, both occ(Rp) and occ(Wp) hold ($\Pi_1$)

occ(Rp) $\Rightarrow$ occ (Rc)
also, occ(Rp) $\Rightarrow$ val (Rc) = true ($\Phi_{IS}$)

link (Rc, Wc₁) $\lor$ link (Rc, Wc₂) ($\Pi_1$)
link (Rc, Wc₁) leads to conflict ($\Pi_2$)
so, link (Rc, Wc₂) and link (Rp, Wp)
so, hb (Wc₂, Rc) and hb (Wp, Rp) ($\Pi_2$)

linearize to obtain a feasible trace
Interference Pruning

- $\Pi$ may have many redundant instantiations
  - Many r-w interferences are infeasible
  - $\Pi_1$: $\neg \text{link}(r,w)$ holds ($w'$ occurs after $w$, before $r$ in all runs)
  - $\Pi_2$: $\neg \text{hb}(w,r)$ holds ($w$ occurs after $r$ in all runs)
  - $\Pi_3$: $\neg \text{hbet}(w,w',r)$ (for some $w$, $w'$, $r$)

- Static analysis of Interference Skeleton
  - Prune away infeasible r-w interferences
Implementation

- FUSION framework for analyzing concurrent programs
  - combines dynamic and symbolic analysis
  - used to obtain (bounded) program slices
- Yices SMT solver
- Compared with/without summarization (S), pruning optimization (O)
## Experiments

<table>
<thead>
<tr>
<th>Bm (#Thr)</th>
<th>N</th>
<th>E</th>
<th>R</th>
<th>W</th>
<th>-S (FSE’09)</th>
<th>+S</th>
</tr>
</thead>
<tbody>
<tr>
<td>SB(2)</td>
<td>108</td>
<td>107</td>
<td>6</td>
<td>19</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SB(3)</td>
<td>723</td>
<td>722</td>
<td>270</td>
<td>289</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>Ind (20)</td>
<td>1312</td>
<td>1439</td>
<td>110</td>
<td>291</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>Ind (29)</td>
<td>2446</td>
<td>2691</td>
<td>360</td>
<td>887</td>
<td>129</td>
<td>6</td>
</tr>
<tr>
<td>Ind (30)</td>
<td>2859</td>
<td>3149</td>
<td>468</td>
<td>1104</td>
<td>517</td>
<td>7</td>
</tr>
<tr>
<td>Ind (31)</td>
<td>3398</td>
<td>3747</td>
<td>594</td>
<td>1332</td>
<td>&gt;1800</td>
<td>13</td>
</tr>
<tr>
<td>Ind (32)</td>
<td>4585</td>
<td>5065</td>
<td>888</td>
<td>1856</td>
<td>&gt;1800</td>
<td>104</td>
</tr>
<tr>
<td>acc (11)</td>
<td>906</td>
<td>905</td>
<td>134</td>
<td>372</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>acc (21)</td>
<td>1748</td>
<td>1747</td>
<td>708</td>
<td>25</td>
<td>&gt;1800</td>
<td>10</td>
</tr>
</tbody>
</table>
## Experiments

| Bm (#Thr) | |N| |E| |R| |W| +S-O | +S+O |
|---|---|---|---|---|---|---|---|---|---|
| SB(2) | 108 | 107 | 6 | 19 | 1 | 1 |
| SB(3) | 723 | 722 | 270 | 289 | 711 | 3 |
| Ind (20) | 1312 | 1439 | 110 | 291 | 355 | 0.1 |
| Ind (29) | 2446 | 2691 | 360 | 887 | >1800 | 6 |
| Ind (30) | 2859 | 3149 | 468 | 1104 | >1800 | 7 |
| Ind (31) | 3398 | 3747 | 594 | 1332 | >1800 | 13 |
| Ind (32) | 4585 | 5065 | 888 | 1856 | >1800 | 104 |
| acc (11) | 906 | 905 | 134 | 372 | 121 | 1 |
| acc (21) | 1748 | 1747 | 708 | 25 | >1800 | 10 |
Conclusions

- **Avoiding Bi-modal** reasoning leads to significant (possibly exponential) speedups

- **Sequential Consistency (SC)** axioms to compose shared memory programs
  - model interference directly
  - avoid scheduler

- Future work: Automated axiom instantiations
Thanks!

Questions?
FUSION framework

We control the scheduling for you

We repeatedly run the program till all possible scenarios are tested

✓ No False Bugs
✓ Easier for developers to use