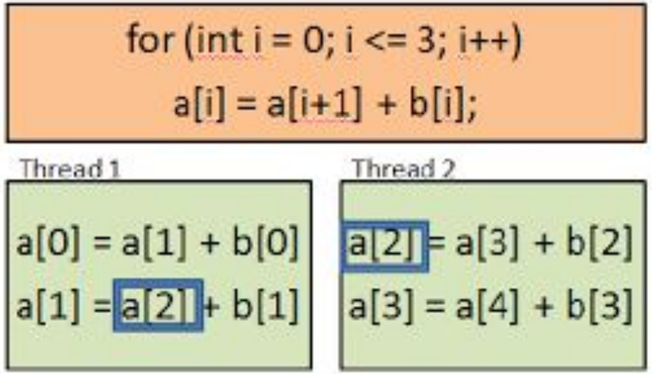
Generating Data Race Witnesses by an SMT-based Analysis

Mahmoud Said, Chao Wang, Zijiang Yang, and Karem Sakallah3

Data Race

When multiple threads access the same variable at the same time, and one of them is changing that variable.



a[2] is updated in the second thread before a[1] uses it in the first thread. Wrong a[1] gets generated.

Motivations

Data races exhibit many different behaviors

A single problem can take weeks for programmers to identify.

Previous techniques don’t scale, none can reliably reproduce the data race

Events

$Tid = \{1,\ldots,n\}$ - set of thread indices

Every operation is an *event* - defined as (*tid*, *type*, *var*, *val*)

tid - thread index, type - {read, write, fork, join, acquire, release, wait, notify, notifyAll}

var - shared variable or synchronization object, val - concrete value or child thread index

Example: e_i : (1, *fork*, –, 2) - e_i.tid = 1, e_i.type = *fork*, e_i.val = 2, e.idx = i

Partial Order and Linearization

Traces are a total order on the set of events π = {e1, . . . , en}

Define partially ordered set T_π = (T, ⊆) where ⊆ is a partial order such that:

- e_i.tid = e_j.tid and e_i appears before e_j in π, then $e_i \subseteq e_j$
- e_i = (tid_1, fork, −, tid_2) and e_j is the first event of thread tid_2 in π, then $e_i \subseteq e_j$
- e_i = (tid_1, join, −, tid_2) and e_j is the last event of thread tid_2 in π, then $e_j \subseteq e_i$
- ⊆ is transitively closed

Not all T_π correspond to actual program executions, so a sequentially consistent linearization τ_π of T_π satisfies Write-Read and Synchronization Consistency

The set of all T_π forms the search space for the witness generation algorithm

Encoding Solution

$$\psi_\pi \ := \ \alpha_\pi \wedge \beta_\pi \wedge \gamma_\pi \wedge \rho_\pi$$

Partial Order

Write-Read Consistency

Synchronization consistency

Data race property

Encoding Partial Order

Introduce Event Order (EO) variable to represent position in linearization of T_π

$o_{e.idx}$ is the EO for e, domain of o_i is $[1\ldots|\pi|]$

Enforce total order within each thread, enforces order between first event of thread and a corresponding fork, or last event and corresponding join. t.first and t.last are events in thread t

FORK and JOIN are the set of all fork and join operations in T_π, for e in FORK e.val is the child index thread, $(t_{e.val})$.first.idx is index of the first event in child thread

$$\alpha_\pi \equiv \left(\bigwedge_{t=1}^{T} \left(o_{e_1^t.idx} < \cdots < o_{e_n^t.idx} \right) \wedge \bigwedge_{e \in FORK} \left(o_{e.idx} < o_{(t_{e.val}).first.idx} \right) \wedge \bigwedge_{e \in JOIN} \left(o_{(t_{e.val}).last.idx} < o_{e.idx} \right) \right) \quad (1)$$

Example

$$\begin{aligned}
e_0 &: (1, fork, -, 2)\\
e_1 &: (1, write, x, 1)\\
e_2 &: (1, acquire, o, -)\\
e_3 &: (1, write, x, 0)\\
e_4 &: (1, wait, o, -)\\
e_5 &: (2, acquire, o, -)
\end{aligned}$$

$$\begin{aligned}
e_6 &: (2, read, x, 0)\\
e_7 &: (2, notifyAll, o, -)\\
e_8 &: (2, release, o, -)\\
e_9 &: (2, read, x, 0)\\
e_{10} &: (1, release, o, -)
\end{aligned}$$

partial order:

$$\begin{aligned}
\alpha_1 &: o_0 < o_1 < o_2 < o_3 < o_4 < o_{10}\\
\alpha_2 &: o_5 < o_6 < o_7 < o_8 < o_9\\
\alpha_3 &: o_0 < o_5
\end{aligned}$$

Encoding Write-Read Consistency

The equation handles 2 cases:

1. e has no thread immediate write predecessor, so the value read is the vars initial value

2. e follows a write event e1 in its predecessor write of the same value set and all other writes to e.var happen before e1 or after e

Definition 3. Write-Read Consistency: *A linearization l is write-read consistent iff for any read event e (1) if there exists a write event e' such that $e' = e.liwp$, then $e.val = e'.val$; (2) if e' does not exist, then $e.val = e.var.init$. Here $e.var.init$ is the initial value of variable $e.var$.*

$$\beta_\pi \equiv \bigwedge_{e \in \pi \wedge e.type = read} \left(\left((e.tiwp = null) \wedge (e.val = e.var.init) \wedge \bigwedge_{e1 \in e.pws} (o_{e.idx} < o_{e1.idx}) \right) \vee \bigvee_{e1 \in e.pwsv} \left(\begin{array}{l} (o_{e1.idx} < o_{e.idx}) \wedge \\ \bigwedge_{e2 \in e.pws \wedge e2 \neq e1} (o_{e.idx} < o_{e2.idx} \vee o_{e2.idx} < o_{e1.idx}) \end{array} \right) \right)$$

Example

$e_0 : (1, fork, -, 2)$
$e_1 : (1, write, x, 1)$
$e_2 : (1, acquire, o, -)$
$e_3 : (1, write, x, 0)$
$e_4 : (1, wait, o, -)$
$e_5 : (2, acquire, o, -)$
$e_6 : (2, read, x, 0)$
$e_7 : (2, notifyAll, o, -)$
$e_8 : (2, release, o, -)$
$e_9 : (2, read, x, 0)$
$e_{10} : (1, release, o, -)$

write-read consistency:

$$\beta : (o_6 < o_1 \vee o_3 < o_6) \\ \wedge (o_9 < o_1 \vee o_3 < o_9)$$

Encoding Data Race

PDR is a set of potential data races in T_{π}, each data race is event pairs (e_1, e_2) such that they are in different threads, access the same variable, and one of them is a write.

A data race exists in a trace only if e1 is immediately followed by e2

Let e1' and e2' be events right before e1 and e2, respectively

Let e1" and e2" be events right after e1 and e2, respectively

Therefore a data race will exist by the equation holding on these constraints

$$\rho_\pi \equiv \bigvee_{(e1,e2)\in PDR} ((o_{e1'.idx} < o_{e2.idx} < o_{e1''.idx}) \wedge (o_{e2'.idx} < o_{e1.idx} < o_{e2''.idx})) \quad (3)$$

Symbolic Encoding of Synchronization Consistency

Replace object variables with simple type variables for SMT solvers

Assume no recursive locks here

Each object o has:

- An integer variable o_o with domain [0,...N], if $o_o = 0$ it is free otherwise o_o is the index of the thread that owns it
- N boolean variables o_{w_t}, this variable is true if thread t is in o's wait set

Encoding Synchronization Consistency cont.

(t, acquire, o, –) - $o_o = 0 \rightarrow o'_o = t.$

(t, release, o, –) - $o_o = t \rightarrow o'_o = 0.$

(t, wait, o, –) - $(o_o = t \rightarrow o'_{w\_t} \wedge o_o = 0)$. $(o_o = 0 \wedge \neg o_{w\_t}) \rightarrow o'_o = t$

(t, notifyAll, o, –) - $o_o = t \rightarrow \bigwedge_{t1 \in o.wait} \neg o'_{w\_t1}$

(t, notify, o, –) - H_{w_t} = 0 if thread is not waiting on o, only 1 H_{w_t} = 1

$$\bigwedge_{1 \leq t \leq N} (\neg o_{w\_t} \rightarrow \neg H_{w\_t} = 0) \text{ , } (\bigvee_{1 \leq t \leq N} o_{w\_t}) \rightarrow (\Sigma_{1 \leq t \leq N} H_{w\_t} = 1)$$

$$\bigwedge_{t \in Tid} (H_{w\_t} = 1 \rightarrow \neg o'_{w\_t} \wedge H_{w\_t} = 0 \rightarrow o'_{w\_t} = o_{w\_t})$$

Recursive Lock Free Example

$e_0 : (1, fork, -, 2)$
$e_1 : (1, write, x, 1)$
$e_2 : (1, acquire, o, -)$
$e_3 : (1, write, x, 0)$
$e_4 : (1, wait, o, -)$
$e_5 : (2, acquire, o, -)$
$e_6 : (2, read, x, 0)$
$e_7 : (2, notifyAll, o, -)$
$e_8 : (2, release, o, -)$
$e_9 : (2, read, x, 0)$
$e_{10} : (1, release, o, -)$

Synchronization Event	Interpretation	Predecessor Write Set	Predecessor Write Set with Same Value
$e_2 : (1, acquire, o, -)$	$o_o = 0 \rightarrow o'_o = 1$	$o_o : \{e_5, e_8\}$	$o_o : \{e_8\}$
$e_4 : (1, wait, o, -)$	$o_o = 1 \rightarrow o'_{w\_1} \wedge o'_o = 0$	$o_o : \{e_2, e_5, e_8\}$	$o_o : \{e_2\}$
e'_4	$o_o = 0 \wedge \neg o_{w\_1} \rightarrow o'_o = 1$	$o_o : \{e_4, e_8, e_5\}$ $o_{w\_1} : \{e_4, e_7\}$	$o_o : \{e_4, e_8\}, o_{w\_1} : \{e_7\}$
$e_5 : (2, acquire, o, -)$	$o_o = 0 \rightarrow o'_o = 2$	$o_o : \{e_2, e_4, e'_4, e_{10}\}$	$o_o : \{e_4, e_{10}\}$
$e_7 : (2, notifyAll, o, -)$	$o_o = 2 \rightarrow \neg o'_{w\_1}$	$o_o : \{e_2, e_4, e'_4, e_5, e_{10}\}$	$o_o : \{e_5\}$
$e_8 : (2, release, o, -)$	$o_o = 2 \rightarrow o'_o = 0$	$o_o : \{e_2, e_4, e'_4, e_5, e_{10}\}$	$o_o : \{e_5\}$
$e_{10} : (1, release, o, -)$	$o_o = 1 \rightarrow o'_o = 0$	$o_o : \{e'_4, e_5, e_8\}$	$o_o : \{e'_4\}$

Recursive Lock Free

v.iv - initial value = not in waitset and not owning object o

v.av - assumed value = value specified in subformula e.assume

v.wv - written value = value specified in subformula e.update

If v.av = v.iv, e can be anywhere before a write to v

If e follows an event e1 in v_e.pwsv, e happens after e1 so e can be e', other writes don't interfere

$$\gamma_e \equiv \bigwedge_{v \in e.assume} \left(\left(v_e.av = v.iv \wedge v_e.first \wedge \bigwedge_{e_1 \in v_e.pws} o_e.idx < o_{e_1}.idx \right) \vee \bigvee_{e_1 \in v_e.pwsv} \left((o_e.idx < o_{e_1}.idx) \wedge \bigwedge_{e_2 \in v_e.pws \wedge e_2 \neq e_1} (o_e.idx < o_{e_2}.idx \vee o_{e_2}.idx < o_{e_1}.idx) \right) \right)$$

With Recursive Locks

Define variable $depth_o^t$ denotes depth of object o locked by thread t, increase value of depth for each acquire and decrease for each release

Event	Encoding	Encoding with Optimization
e_2	$(o_2 < o_5 \wedge o_2 < o_8) \vee ((o_8 < o_2) \wedge (o_5 < o_8 \vee o_2 < o_5))$	$(o_2 < o_5) \vee (o_8 < o_2)$
e_4	$(o_2 < o_4) \wedge (o_5 < o_2 \vee o_4 < o_5) \wedge (o_8 < o_2 \vee o_4 < o_8)$	$(o_5 < o_2 \vee o_4 < o_5) \wedge (o_8 < o_2 \vee o_4 < o_8)$
e'_4	$\begin{pmatrix} (o_4 < o_{4'}) \wedge (o_5 < o_4 \vee o_{4'} < o_5) \\ \wedge (o_8 < o_4 \vee o_{4'} < o_8) \\ (o_8 < o_{4'}) \wedge (o_4 < o_8 \vee o_{4'} < o_4) \\ \wedge (o_5 < o_8 \vee o_{4'} < o_5) \end{pmatrix} \vee$	$\begin{pmatrix} (o_4 < o_{4'}) \wedge (o_5 < o_4 \vee o_{4'} < o_5) \\ \wedge (o_8 < o_4 \vee o_{4'} < o_8) \end{pmatrix} \vee$ $((o_8 < o_{4'}) \wedge (o_4 < o_8))$
	$(o_7 < o'_4) \wedge (o_4 < o_7 \vee o'_4 < o_4)$	$(o_7 < o'_4) \wedge (o_4 < o_7)$
e_5	$(o_5 < o_2 \wedge o_5 < o_4 \wedge o_5 < o'_4 \wedge o_5 < o_{10}) \vee$ $\begin{pmatrix} (o_4 < o_5) \wedge (o_2 < o_4 \vee o_5 < o_2) \wedge \\ (o'_4 < o_4 \vee o_5 < o'_4) \wedge (o_{10} < o_4 \vee o_5 < o_{10}) \\ (o_{10} < o_5) \wedge (o_2 < o_{10} \vee o_5 < o_2) \wedge \\ (o'_4 < o_{10} \vee o_5 < o'_4) \wedge (o_4 < o_{10} \vee o_5 < o_4) \end{pmatrix} \vee$	$(o_5 < o_2) \vee (o_{10} < o_5) \vee$ $(o_4 < o_5 \wedge o_5 < o'_4 \wedge o_5 < o_{10})$
e_7	$(o_5 < o_7) \wedge (o_2 < o_5 \vee o_7 < o_2) \wedge (o_4 < o_5 \vee o_7 < o_4) \wedge$ $(o'_4 < o_5 \vee o_7 < o'_4) \wedge (o_{10} < o_5 \vee o_7 < o_{10})$	$(o_2 < o_5 \vee o_7 < o_2) \wedge (o_4 < o_5 \vee o_7 < o_4) \wedge$ $(o'_4 < o_5 \vee o_7 < o'_4) \wedge (o_{10} < o_5 \vee o_7 < o_{10})$
e_8	$(o_5 < o_8) \wedge (o_2 < o_5 \vee o_8 < o_2) \wedge (o_4 < o_5 \vee o_8 < o_4) \wedge$ $(o'_4 < o_5 \vee o_8 < o'_4) \wedge (o_{10} < o_5 \vee o_8 < o_{10})$	$(o_2 < o_5 \vee o_8 < o_2) \wedge (o_4 < o_5 \vee o_8 < o_4) \wedge$ $(o'_4 < o_5 \vee o_8 < o'_4) \wedge (o_{10} < o_5 \vee o_8 < o_{10})$
e_{10}	$(o'_4 < o_{10}) \wedge (o_5 < o'_4 \vee o_{10} < o_5) \wedge (o_8 < o'_4 \vee o_{10} < o_8)$	$(o_5 < o'_4 \vee o_{10} < o_5) \wedge (o_8 < o'_4 \vee o_{10} < o_8)$

- An event $e : (t, acquire, o, -)$ is called the *first acquire event* if $e.depth_o^t = 0$. Its corresponding constraint is $o_o = 0 \rightarrow o'_o = t$.
- For event $e : (t, acquire, o, -)$ that is not a first acquire event, its corresponding constraint is $o_o = t \rightarrow o'_o = t$.
- An event $e : (t, release, o, -)$ is called the *last release event* if $e.depth_o^t = 0$. Its corresponding constraint is $o_o = t \rightarrow o'_o = 0$.
- For event $e : (t, release, o, -)$ that is not a last release event, its corresponding constraint is $o_o = t \rightarrow o'_o = t$.

Maximal Proof (?) & Bogus Warnings

Only mention is that they improve over previous maximal techniques

Theorem 1. *Let π be the given multithreaded trace. There exists a data race witness in a sequentially consistent linearization of $\mathcal{T}_\pi$ iff ψ_π is satisfiable:*

$$\psi_\pi \equiv \alpha_\pi \wedge \beta_\pi \wedge \gamma_\pi \wedge \rho_\pi$$

Therefore any data race warnings that don't satisfy this equation are bogus

Using Yices SMT Solver

$$\psi_\pi = (\alpha_\pi \wedge \beta_\pi \wedge \gamma_\pi) \wedge \rho_\pi$$

Divide constraints into 2 parts

First construct the first part, add data race event pair (e1, e2) as retractable assertion which can be removed after satisfying. If result is SAT then return witness otherwise a witness doesn’t exist.

Retract first pair and add the next event pair to SMT solver

Since original input has bogus warnings, the output can still contain these bogus warnings

Evaluation Cont.

Only up to Medium Traces

Not very feasible to use

Contribution

Improve existing data race detection algorithms to give all possible witnesses for data races

Improve runtime greatly by using FOL (no evidence given in paper)

Can be applied to future data race detection algorithms as well

Future Directions

Utilizing newer data race detection algorithms

Evolution aware

Questions

How come a large project wasn't used, it doesn't really seem that scalable in their claims since I'm sure people would want to use this for larger traces

Is it possible to pinpoint the more likely witnesses for data races?

What exactly do recursive locks do to a program other than using for constraints?