concurrent programs highly non-deterministic

data race = 2 conflicting events right next to each other in trace

conflicting events = 2 read/write events, at least one event is write event

you can find potential data races by rerunning the program and obtain different traces

alternative is finding valid reorderings of a trace

Trace A may not contain data races but its reorderd Trace B or C may

Possible to predict trace orderings that exhibit data races → dynamic data race prediction

Exhaustive predictive methods → identify as many reorderings as possible

Efficient predictive methods → O(n) runtime
efficient favourable over being exhaustive
→ compromise completeness and soundness

complete = all valid reorderings that exhibit data races can be predicted
any not predicted race → false negative

sound = races reported can be observed via appropriate reordering of trace, unsound → wrongly classified data races → false positive

Lockset method unsound, Happens-before method incomplete

events processed in stream-based fashion = online
offline may get better results if trace in its entire form is present

Happens-before method

1. Trace from specific program run
2. Derive happens-before ordering relation from trace
3. Two conflicting events are unordered → data race

$e < f$ → e ordered before f = e connected to f
→ e must be executed before f

Req.:

- Strict partial order
  → partial: not all events need to be ordered
  → strict partial order: ordering relation a) transitive + b) not reflexive
  a) $e < f$, $f < g$ → $e < g$ b) event cannot happen itself

If two conflicting events e, f: not $e < f$ nor $f < e$

→ (e,f) = data race pair

→ trace can be reordered such that e, f appear next to each other

If (e,f) is data race then (f,e) is also data race

(e,f) and (f,e) = distinct representative for same data race

Lamport's happens-before relation

Program order condition

e,f events in same thread where e appears before f, then $e < f$

Critical section order condition

$acq(y)$, $rel(y)$ same lock y, both events result from diff. threads and $rel(y)$ appears before $acq(y)$ in trace, then $rel(y) < acq(y)$

Vector Clocks

every thread time stamp in array

Initially all entries 0ed except thread t, set to 1

$inc([k_1, \ldots, k_{i-1}, k_i, k_{i+1}, \ldots k_n], i] = [k_1, \ldots, k_{i-1}, k_i+1, \ldots, k_n]$

$sync([i_1, \ldots, i_n], [j_1, \ldots, j_n]) = [max(i_1, j_1), \ldots, max(i_n, j_n)]$

program order cond. → inc

critical section order cond. → sync for rel(y) < acq(y)
then inc

$[pre_1, pre_2]$_fun(x)_$[post_1, post_2]$

Find HB data races by comparing pre vector clocks

$V_1 < V_2$ if ∀ array positions i, $V_1[i] \leq V_2[i]$
and array pos j exists with $V_1[j] < V_2[j]$

If neither $V_1 < V_2$, nor $V_1 > V_2$ then V_1, V_2
are incomparable

Vector clocks offline

1st pass: store all pre vector clocks of read/write
events

2nd pass: find reorderings of trace with data races

→ extra space needed for storing pre vector clocks
→ extra time needed because two passes
but all HB data races can be identified

online

- → pass through trace and store every read/write event if they are incomparable (data race)
- → but only keep event if it represents data race, non-data-race events are only stored until next read/write-evenrt

→ invariant: $V_1, V_2 \in W(x) \cup R(x)$, $!happensBefore(V_1, V_2)$ and $!happensBefore(V_2, V_1)$

→ $\forall (V_1, V_2)$ where $V_1, V_2 \in W(x)$ write-write-data race

→ $\forall (V_1, V_2)$ where $V_1 \in W(x)$ and $V_2 \in R(x)$ write-read data race