Goldilocks: Efficiently computing the Happens-Before Relation Using Locksets

- Authors: T. Elmas, S. Qadeer, and S. Tasiran
- Proceedings of the Workshop on Formal Approaches to Testing and Runtime Verification, 2006
- Presented in CS595C: Architectural Support for Dynamic Software Analysis

Goal

- A new dynamic race detection algorithm which is
  - efficient as simple lockset-based algorithms and
  - precise as vector-clock-based algorithms

More Precisely...

- Checks race freedom in constant time and
- Captures the happens-before relation precisely, i.e., declare a race exactly when two accesses to a shared variable are not ordered by the happens-before relation.
- Reports a data race on an execution if and only if there exists a data race in that execution (sufficient and necessary conditions)

Lockset based algorithm

Checks whether a given execution σ has a data-race.

- $LH(t)$: the set of locks held by t
- $LS(x)$: the set of locks that the system thinks the shared variable x can be accessed without raising data races.
- A data race is declared if $LH(t) \cap LS(x)$ is empty.

Motivation Example

```
T1: acq(m1); acq(m2); x=1; rel(m1); rel(m2)
T2: acq(m2); acq(m3); x=2; rel(m2); rel(m3)
T3: acq(m1); acq(m3); x=3; rel(m1); rel(m3)
```

- Simple lockset-based race detection algorithm
  - Each shared variable is protected by a fixed unique lock ($LS(x) = m2$)
  - Consider the execution $T_1; T_2; T_3$, a data race is reported since $LH(T_3) = \{m1, m3\}$ and $LH(T_3) \cap LS(x) = \emptyset$.

Motivation Example

```
T1: acq(m1); acq(m2); x=1; rel(m1); rel(m2)
T2: acq(m2); acq(m3); x=2; rel(m2); rel(m3)
T3: acq(m1); acq(m3); x=3; rel(m1); rel(m3)
```

- Less-conservative algorithm
  - Each shared variable is protected by a dynamic lock set. ($LS(x)$ is updated to $LH(t)$ after a race free access to x by a thread t)
  - Consider the execution $T_1; T_2; T_3$, no data race reported since $LH(T_3) = \{m1, m3\}$, $LS(x) = \{m2, m3\}$ and $LH(T_3) \cap LS(x) \neq \emptyset$.

Motivation Example II

```
Class IntBox{ int x;}
Inbox a=new IntBox(); // IntBox o1 created
Inbox b=new IntBox(); // IntBox o2 created
T1: acq(m1); a.x=++; rel(m1);
T2: acq(m1); acq(m2); tmp=a; a=b; b=tmp; rel(m1); rel(m2)
T3: acq(m2); b.x=++; rel(m2);
```

- Less-conservative algorithm
- Consider the execution $T_1; T_2; T_3;$
  - Initiall, $LS(o1) = LS(o1.x) = \{m1\}$ and $LS(o2) = LS(o2.x) = \{m2\}$.
  - after T1, LS(o1.x) is assigned by $LH(T1)$, which is $\{m1\}$
  - after T2, $LS(o1), LS(o_2)$ are assigned by $LH(T2)$, which is

$\{m1, m2\}$, but $LS(o1.x) = \{m1\}$ and $LS(o2.x) = \{m2\}$ remain the same, since they are not directly accessed.

– a data race is reported since $LH(T_3) = \{m2\}$, $LS(o1.x) = \{m1\}$ and $LH(T_3) \cap LS(o1.x) = \emptyset$. (Note that b points to o1)

- However, none of them violate happens-before relations.
- Previous lockset-based algorithms are sound but raise false alarms.
- The Goldilocks algorithm is the first sound and complete lockset-based algorithm.

Preliminaries

- An object o has data and volatile fields denoted as d and v respectively.
- A data variable is (o, d).
- A synchronization variable is (o, v), and each object o has a special synchronization variable (o, l) referred to itself.
- An execution is $\sigma = s_1 \rightarrow_{t_1}^{\alpha_1} s_2 \rightarrow_{t_2}^{\alpha_2} \ldots s_n \rightarrow_{t_n}^{\alpha_n} s_{n+1}$
- s_i is some system state, and α_i is one of the following actions executed by thread t_i:
  - $acq(o)$: acquire a lock on object o. acq executed by t is blocked until (o, l) is null and then set (o, l) to t.
  - $rel(o)$: release a lock on object o. rel is failed if $o.l = null$; o.w., set $o.l = null$.

- $read(o, d), write(o, d)$: access the data field of object o.
- $read(o, v), write(o, v)$: read and write the volatile field of o.
- fork(u): create a new thred with id u.
- join(u): blocks until thread u terminates.
- alloc(o):create a new object.

Happens-before relation

- Given $\sigma = s_1 \rightarrow^{\alpha_1}_{t_1} s_2 \rightarrow^{\alpha_2}_{t_2} \ldots s_n \rightarrow^{\alpha_n}_{t_n} s_{n+1}$
- The happens-before relation is the smallest transitively-closed relation on the set $\{1, 2, ..., n\}$, such that $k \hookrightarrow l$ if $1 \leq k \leq l \leq n$ and one of the following holds:
  - $t_k = t_l$
  - $\alpha_k = rel(o)$ and $\alpha_l = acq(o)$
  - $\alpha_k = write(o, v)$ and $\alpha_l = read(o, v)$
  - $\alpha_k = fork(t_l)$
  - $\alpha_l = join(t_k)$

Data-race free execution

- σ is data-free on a data variable (o, d) if
  - for all $\alpha_k, \alpha_l \in \{read(o,d), write(o,d)\}$, $k \hookrightarrow l$ or $l \hookrightarrow k$.
- How to define "concurrent read and exclusive write"?

Goldilocks Algorithm

- $LS(o, d)$ is a set of locks and thread ids, which is updated according to the actions along the execution.
- Locks refer to the lockset having any of them may access (o, d) without raising races.
- Thread ids refer to those threads may access (o, d) without raising races.
- Initially, $LS(o, d) = \emptyset$.
- Update rules
  - $\alpha = read(o, d)$ or $\alpha = write(o, d)$:
    if $LS(o, d) \neq \emptyset$ and $t \notin LS(o, d)$, report data race on (o, d);
    o.w., $LS(o, d) := \{t\}$
  - $\alpha = read(o, v)$:

for each (o, d), if $(o, v) \in LS(o, d)$, add t to $LS(o, d)$

- $\alpha = write(o, v)$:
  for each (o, d), if $t \in LS(o, d)$, add (o, v) to $LS(o, d)$
- $\alpha = acq(o)$:
  for each (o, d), if $(o, l) \in LS(o, d)$, add t to $LS(o, d)$
- $\alpha = rel(o)$:
  for each (o, d), if $t \in LS(o, d)$, add (o, l) to $LS(o, d)$
- $\alpha = fork(u)$:
  for each (o, d), if $t \in LS(o, d)$, add u to $LS(o, d)$
- $\alpha = join(u)$:
  for each (o, d), if $u \in LS(o, d)$, add t to $LS(o, d)$
- $\alpha = alloc(o)$:
  for each d, $LS(o, d) := \emptyset$.

- Invariants maintained by Goldilocks
  - If $(o', l) \in LS(o, d)$, the last access to (o, d) happens before a

subsequent $acq(o')$

- If $(o', v) \in LS(o, d)$, the last access to (o, d) happens before a subsequent $read(o', v)$
- If $t \in LS(o, d)$ at an access to (o, d), the last access to (o, d) happens before this access performed by t.

- Behind the algorithm:
  - Compute the transitive closure of happens-before edges
  - t is in $LS(o, d)$ if and only if there exists a sequence of happens-before edges between the last access to (o, d) and the next action performed by t.
  - t is added as soon as such sequence is established.

Theorem

Consider $\sigma = s_1 \rightarrow^{\alpha_1}_{t_1} s_2 \rightarrow^{\alpha_2}_{t_2} \ldots s_n \rightarrow^{\alpha_n}_{t_n} s_{n+1}$. $t_n \in LS_n(o,d)$ if and only if $i \hookrightarrow n$, where $i \in [1..n-1]$, α_i and α_n access (o,d), and for all $j \in [i+1, n-1]$, α_j does not access (o,d).

Come back to our example

```
Class IntBox{ int x;}
Inbox a=new IntBox(); // IntBox o1 created
Inbox b=new IntBox(); // IntBox o2 created
T1: acq(m1); a.x=++; rel(m1);
T2: acq(m1); acq(m2); tmp=a; a=b; b=tmp; rel(m1); rel(m2)
T3: acq(m2); b.x=++; rel(m2);
```

Consider the execution $T_1; T_2; T_3;$

Thread ID	Action	rule	$LS(o_1, x)$
T1	a:=IntBox()		$\emptyset$
T1	b:=IntBox()		
T1	acq(m1)		
T1	a.x++	First access to (o_1, x)	$\{T1\}$
T1	rel(m1)	$T1 \in LS(o_1.x)$, add $m1$	$\{T1, m1\}$
T2	acq(m1)	$m1 \in LS(o_1.x)$, add $T2$	$\{T1, m1, T2\}$
T2	acq(m2)		
T2	tmp:=a		
T2	a:=b		
T2	b:=tmp		
T2	rel(m1)	$T2 \in LS(o_1, x)$, add $m1$	$\{T1, m1, T2\}$
T2	rel(m2)	$T2 \in LS(o_1, x)$, add $m1$	$\{T1, m1, T2, m2\}$
T3	acq(m2)	$m2 \in LS(o_1, x)$, add $T3$	$\{T1, m1, T2, m2, T3\}$
T3	b.x++	access to (o_1, x), $T3 \in LS(o_1, x)$, **no race**	$\{T3\}$
T3	rel(m2)	$T3 \in LS(o_1, x)$, add $m2$	$\{T3, m2\}$

Implementation

- Short circuit checks: constant time look up
- Lazy evaluation of lockset update rules: maintain an update list and update only when the data is accessing

More about implementation

$Handle - Action(t, \alpha)$

- if($\alpha \in \{read(o, d), write(o, d)\}$)\{
  if($(o, d).owner \neq t$ and $(o, d).alock$ is not held by t)\{
  $Apply - Lockset - Rules(t, \alpha)$;
  Randomly assign $(o.d).alock$ to a lock held by t;\}\}
- else append (t, α) to the update list;

Evaluation

- Run the instrumented version of the Kaffe JVM
- Benchmarks: 220 lines-6000 lines with 7 to 3 threads
- Extra running time(Overhead) = uninstrumented running time $\times$ slowdown

	Un	VC	BE	Gold
running time	1.9s-28.2s	51.3s-243.8s	46.1s-157.5s	33.1s-117.5s
slowdown	0	0.8-63.1	0.6-40.6	0.1-25.1