Sherlock: Scalable Deadlock Detection for Concurrent Programs

Mahdi Eslamimehr, Jens Palsberg

David Wellig
ETH Zürich

11.03.2015

Clarification of terms

Types of deadlock-detection techniques:

- static: detection without running the program
- dynamic: gathers information during one or more runs
- hybrid: combination of static and dynamic

Scalable:

- The property of an algorithm of being suitably efficient and practical when applied to large situations.

Sherlock is a *dynamic* deadlock-detection algorithm for Java programs, which works well for *large schedules* and which especially determines the schedule to a deadlock.

Types of deadlocks treated by Sherlock

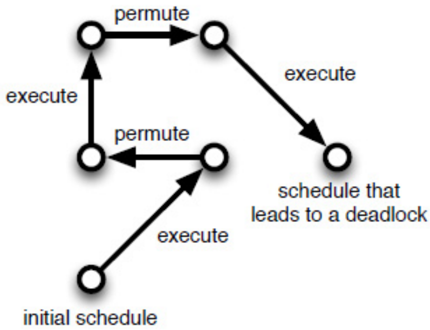
input...	
T_1	T_2
$1:\ \text{synchronized}(A)\{$ $2:\quad \text{synchronized}(B)\{\}$ $\quad\quad \ldots\}\}$	$1:\ \text{synchronized}(B)\{$ $2:\quad \text{synchronized}(A)\{\}$ $\quad\quad \ldots\}\}$

- synchronized$(A)\{s\}$: Thread acquires lock of A and executes statement s.
- In the state $(T_1 \rhd 2, T_2 \rhd 2)$ we are deadlocked.

Basic idea of Sherlock

Basic structure of the algorithm:

$$(execute \circ permute)^i \circ execute, \quad i \in \mathbb{N}.$$



Execute function

Interface of the *execute* function:

$$\begin{aligned} execute : & Program \times Schedule \times Candidate \to \\ & Input \times Schedule \times Bool \oplus \{none\} \end{aligned}$$

Suppose we have program p, schedule s and candidate c:

$$(a, trace, found) = execute(p, s, c).$$

If *found* is true, *trace* is the actual schedule that leads to the candidate c which then in fact is a deadlock. a is the input to the program.

execute uses concolic execution, which is an execution that records constraints.

Simple example for concolic execution

Let y be an input, and s a statement. Consider:

$$x = 6;\ \text{if } (y > 4)\{s\};$$

How to find an input that leads to s?

First run:

$$y = 0 \Rightarrow (y > 4) \text{ prevents executing } s \Rightarrow (y > 4) \text{ is recorded.}$$

Second run:

$$y = 8 \Rightarrow \text{Success.}$$

Permute function (1)

Interface of the *permute* function:

$$permute : Schedule \times Candidate \rightarrow Schedule \oplus \{none\}$$

permute permutes the events of schedule $\pi = \langle e_1, \ldots, e_n \rangle$, ie.

$$permute(\pi, c) = \langle e_{\sigma(1)}, \ldots, e_{\sigma(n)} \rangle, \quad \text{for a } \sigma \in S_n,$$

where S_n is the set of all permutations of n elements. Furthermore σ has to satisfy the following constraints

$$a_\pi \wedge \beta_\pi \wedge \Psi_{(V,E)} \wedge \delta_c.$$

Permute function (2)

Constraints:

- a_π: Happens-before relation.
- β_π: Write-read consistency. Read event reads value written by most recent write event.
- $\Psi_{(V,E)}$: Lock-order constraints. (V, E) is the lock-order graph. Nodes V are events that acquire locks.
- δ_c: Representation of deadlock-candidate c.

A closer look on Sherlock

```
(Deadlock set) Sherlock(Program p) {
    (Cycle set) candidates = GoodLock(p)
    Schedule s0 = initialRun(p)
    (Deadlock set) dlocks = ∅

    for each Cycle c ∈ candidates do {
        boolean found = false
        boolean stalled = false
        int i = 0
        Schedule s = s0
        while (¬ found) ∧ (¬ stalled) ∧ (i ≤ 1000) {
            case execute(p, s, c) of
                (Input × Schedule × boolean) (a, trace, true) : {
                    dlocks = dlocks ∪ {(c, a, trace)}
                    found = true
                }
                (Input × Schedule × boolean) (a, trace, false) : {
                    case permute(trace, c) of
                        Schedule s' : {s = s'}
                        none : {stalled = true}
                }
                none : {stalled = true}
            }
            i = i + 1
        }
    }

    return dlocks
}
```

Example for Sherlock (1)

$y =$ to be determined	
T_1	T_2
$1: x := 6$	$1: x := 2$
$2:$ synchronized$(A)\{$	$2:$ synchronized$(B)\{$
$3:$ if$(y > 4)\{$	$3:$ if$(y^2 + 5 < x^2)\{$
$4:$ synchronized$(B)\{\}$	$4:$ synchronized$(A)\{\}$
$\}\}$	$\}\}$

We use abbreviations for the events:

$$e_i = (T_1, i),\ \ 1 \le i \le 4, \quad e_{i+4} = (T_2, i),\ \ 1 \le i \le 4.$$

Initial run:

$$y = 0 \ \Rightarrow\ s = \langle e_1, e_2, e_3, e_5, e_6, e_7 \rangle.$$

Example for Sherlock (2)

First iteration of while-loop: Record of $(y > 4)$.

$$y = 5 \;\Rightarrow\; trace = \langle e_1, e_2, e_3, e_5, e_6, e_7, e_4 \rangle$$

Permute on *trace*:

$$s = \langle e_5, e_6, e_7, e_1, e_2, e_3, e_4 \rangle$$

Second iteration of while-loop:

$$trace = \langle e_5, e_6, e_7, e_1, e_2, e_3, e_4 \rangle$$

Permute on *trace*:

$$s = \langle e_5, e_6, e_1, e_2, e_7, e_3, e_4 \rangle$$

Third iteration of while-loop:

$$trace = \langle e_5, e_6, e_1, e_2, e_7, e_3, e_4, e_8 \rangle$$

Experimental results (1)

	Static	Hybrid	Dynamic							
benchmarks	Chord	GoodLock	DeadlockFuzzer	ConTest	Jcarder	Java HotSpot	DCJJ	Sherlock		
								total	= new	+ DCJJ
Sor	1	7	0	0	0	0	0	1	1	0
TSP	1	9	0	0	0	0	0	1	1	0
Hedc	24	23	1	0	0	0	1	20	19	1
Elevator	4	13	0	0	0	1	1	5	4	1
ArrayList	9	11	7	6	2	1	7	9	6	3
TreeSet	8	11	7	5	1	3	8	5	0	5
HashSet	11	10	3	1	0	2	5	5	0	5
Vector	3	14	0	1	0	0	1	4	4	0
RayTracer	1	8	0	1	0	0	1	2	1	1
MolDyn	3	6	1	1	1	1	1	1	0	1
MonteCarlo	2	23	0	1	1	1	1	2	1	1
Derby	5	10	2	0	0	0	2	4	3	1
Colt	6	11	0	0	0	0	0	3	3	0
Avrora	78	29	4	2	1	2	4	7	3	4
Tomcat	119	411	9	10	3	4	11	18	10	8
Batic	73	33	5	4	1	3	7	10	3	7
Eclipse	89	389	9	8	4	6	13	23	12	11
FOP	15	11	1	1	0	0	2	4	2	2
H2	25	17	0	1	0	0	1	3	2	1
PMD	20	8	2	2	0	1	3	4	2	2
Sunflow	31	11	1	2	0	2	2	6	4	2
Xalan	42	210	3	4	0	2	4	9	5	4
TOTAL	570	1275	55	50	14	29	75	146	86	60

Figure: Numbers of deadlocks detected by technique

Experimental results (2)

	Sherlock		
schedule length	total =	new +	DCJJ
$10^2 - 10^3$	5	0	5
$10^3 - 10^4$	20	9	11
$10^4 - 10^5$	39	12	27
$10^5 - 10^6$	49	38	11
$10^6 - 10^7$	24	18	6
$10^7 - 10^8$	9	9	0
	146	86	60

Observations:

- Sherlock detects many deadlocks for more than 10^6 execution steps.
- Sherlock only missed 15 detections of DCJJ.

Conclusions, Limitations

Conclusion:

- Sherlock finds more deadlocks than any other dynamic detection technique.
- Thanks to *permute* Sherlock scales also to long schedules.

Limitation:

- Sherlock up to now only supports synchronized methods and statements. Wait, notify and notify all are not supported.
- To find deadlock candidates Sherlock relies on GoodLock. If GoodLock misses a deadlock, so does Sherlock.

Thank you for your attention.