Efficient Concurrency-Bug Detection Across Inputs

State-of-art solution

Step1. Input design:产生test inputs来保证code coverage

Step2. Bug Detection: 使用dynamic bug-detection tool找到可能的buggy interleaving

Step3. Bug Validation: 对每个input，执行多次。来排除suspicious interleaving

但这种方法会降低10-100倍的速度

Motivation

- 不同输入会导致相同的interleaving，现有工具会产生duplicate bug reports.

	# All Race Pairs			# All Atom. Vio.			# Buggy Race Pairs			# Buggy Atom. Vio.		
App.	Total	Unique	**DupRate**	Total	Unique	**DupRate**	Total	Unique	**DupRate**	Total	Unique	**DupRate**
Click	3114	848	**3.6**	6145	2298	**2.7**	6	1	**6.0**	6	1	**6.0**
FFT	300	66	**4.5**	1423	369	**3.8**	28	4	**7.0**	35	5	**7.0**
LU	238	58	**4.1**	874	163	**5.4**	28	4	**7.0**	21	3	**7.0**
Mozilla	1991	481	**4.1**	2459	723	**3.4**	42	6	**7.0**	42	7	**6.0**
PBZIP2	293	65	**4.5**	499	143	**3.5**	32	8	**4.0**	39	11	**3.5**

Table 2: Inefficiency in data-race and atomicity-violation detection across inputs (DupRate measures the average number of inputs that expose the same data race or atomicity violation)

- 减少冗余的工作，不产生duplicate bug reports.

- 使用CFP(concurrency function pair)解决

CFP指在某个Input下，可以并行执行的函数对

```
/*Thread 1*/
foo1(){
  lock(L);
  foo2();
  unlock(L);
  ...
  ...
}
```

```
/*Thread 2*/
lock(L);
bar();
unlock(L);
```

Figure 2: An example of concurrent functions (For illustration purpose, the vertical position of each code statement in the figure represents when the statement is executed)

Method

- Measure CFP for each input and generate aggregated CFP.
- Identify selected inputs and selected functions under these inputs. 用贪心算法
- Apply existing data-race and atomicity-violation detector to selected inputs functions under selected inputs.

Input1: CFP_1=$\{\{f_1,f_2\}, \{f_2,f_3\}, \{f_2,f_4\}, \{f_4,f_5\}\}$
Input2: CFP_2=$\{\{f_1,f_2\}, \{f_3,f_4\}, \{f_3,f_5\}\}$
Input3: CFP_3=$\{\{f_2,f_3\}, \{f_3,f_4\}\}$
$CFP_{Aggregated}$ =$\{\{f_1,f_2\}, \{f_2,f_3\}, \{f_2,f_4\}, \{f_4,f_5\}, \{f_3,f_4\}, \{f_3,f_5\}\}$

Step 1:
Selected input −− Input1
Selected functions −− $\{f_1, f_2, f_3, f_4, f_5\}$
$CFP_{Uncovered}$ = $\{\{f_3,f_4\}, \{f_3,f_5\}\}$

Step 2:
Selected input −− Input2
Selected functions −− $\{f_3, f_4, f_5\}$
$CFP_{Uncovered}$ = $\emptyset$

Figure 8: A toy example of input/function selection

如何产生CFP

- 根据同步信息来判断是否能并行执行

- 分析run-time trace of each thread
  - 计算locks和vector timestamp
  - 根据上一步的信息计算CFP

```
/* .ent, .exi: function entrance, exit;
  .ent_exi.lockset: lockset protecting the critical
       section that holds both entrance and exit;
  .vec_time: vector timestamps calculated using
       order−enforcing synchronization. */
Bool concurrentFunction(f1, f2)
{
  if(f1.thread_id == f2.thread_id) return false;

  /*Can f1 start between f2's entrance and exit?*/
  if ((f1.ent.lockset ∩ f2.ent_exi.lockset) != ∅) return false;
  if (f1.ent.vec_time < f2.ent.vec_time) return false;
  if (f1.ent.vec_time > f2.exi.vec_time) return false;

  /*Can f2 start between f1's entrance and exit?*/
  if ((f2.ent.lockset ∩ f1.ent_exi.lockset) != ∅) return false;
  if (f2.ent.vec_time < f1.ent.vec_time) return false;
  if (f2.ent.vec_time > f1.exi.vec_time) return false;

  return true; /*f1 and f2 are concurrent*/
}
```

Figure 3: Pseudo code that judges concurrent functions