

第七章 并发程序测试与分析

蔡彦

计算机科学国家重点实验室
中国科学院软件研究所

yancai@ios.ac.cn

大纲


- 1) 并发执行简介
- 2) 偏序关系
- 3) 精准并发测试
- 4) 最新技术与趋势

1. 并发程序及其发展现状
2. 基本概念
 - 基础概念
 - 并发缺陷分类
3. 并发程序测试方法
 - 基本动态测试
 - 随机延迟扰动
4. 并发缺陷检测方法
5. 静态方法
 - 数据流分析
 - 符号执行
 - 模型检验
6. 动态方法
 - LockSet
 - 约束求解
 - 偏序方法-流式算法
 - 偏序方法-图算法
7. 最新技术与研究趋势
 - ToccRace
 - Eagle
8. 并发测试与分析技术总结

(内容回顾) 基本概念

- 数据竞争：多个线程同时访问共享数据，且其中至少一个操作为写操作。
- 左图中，line 2和4构成对指针p的数据竞争，在右图的线程交错中，会导致UAF
- 引起的危害：
 - 破坏数据一致性
 - 进一步发展为UAF、DF等漏洞

	t ₁	t ₂
1	p = malloc()	p[0] = 0
2		
3	free(p)	
4	p = null	



	t ₁	t ₂
1	p = malloc()	
2	free(p)	
3	p = null	
4		p[0] = 0

if(p!=NULL) p[0]=0;

(内容回顾) 动态方法-LockSet

共享变量a, 锁 l_1 和 l_2

初始: $C(a) = \{l_1, l_2\}$, $\text{locks_held}(t_1) = \text{locks_held}(t_2) = \{\}$

	t_1	t_2	
$\text{locks_held}(t_1) = \{l_1\}$	1 $lock(l_1)$		
$C(a) = C(a) \cap \text{locks_held}(t_1) = \{l_1\}$	2 $a = 0$		
$\text{locks_held}(t_1) = \{\}$	3 $unlock(l_1)$		
	4	$lock(l_2)$	$\text{locks_held}(t_2) = \{l_2\}$
	5	$a = 1$	$C(a) = C(a) \cap \text{locks_held}(t_2) = \{\}$
	6	$unlock(l_2)$	$\text{locks_held}(t_2) = \{\}$

RACE!



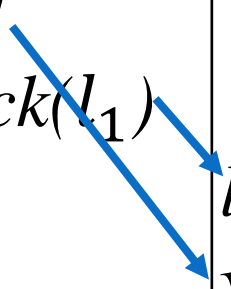
数据竞争: line 2, line 5

(内容回顾) 动态方法-流式算法

➤ Happen-Before关系：由图灵奖得主Leslie Lamport在1978年提出。用 \rightarrow 表示，指满足以下条件的最小的传递性关系：

- 如果事件 e_1 和 e_2 属于同一个线程，且 e_1 发生在前，则 $e_1 \rightarrow e_2$ 。
- 如果事件 e_1 是一个消息的发送者，而 e_2 是该消息的接受者，则 $e_1 \rightarrow e_2$ 。（此处的消息指广义的消息，在并发程序中，包含线程创建、等待、共享变量读写、锁等事件）

	t_1	t_2
1	$lock(l_1)$	
2	$a = 1$	
3	$unlock(l_1)$	
4		$lock(l_1)$
5		$y = a + 1$
6		$unlock(l_1)$



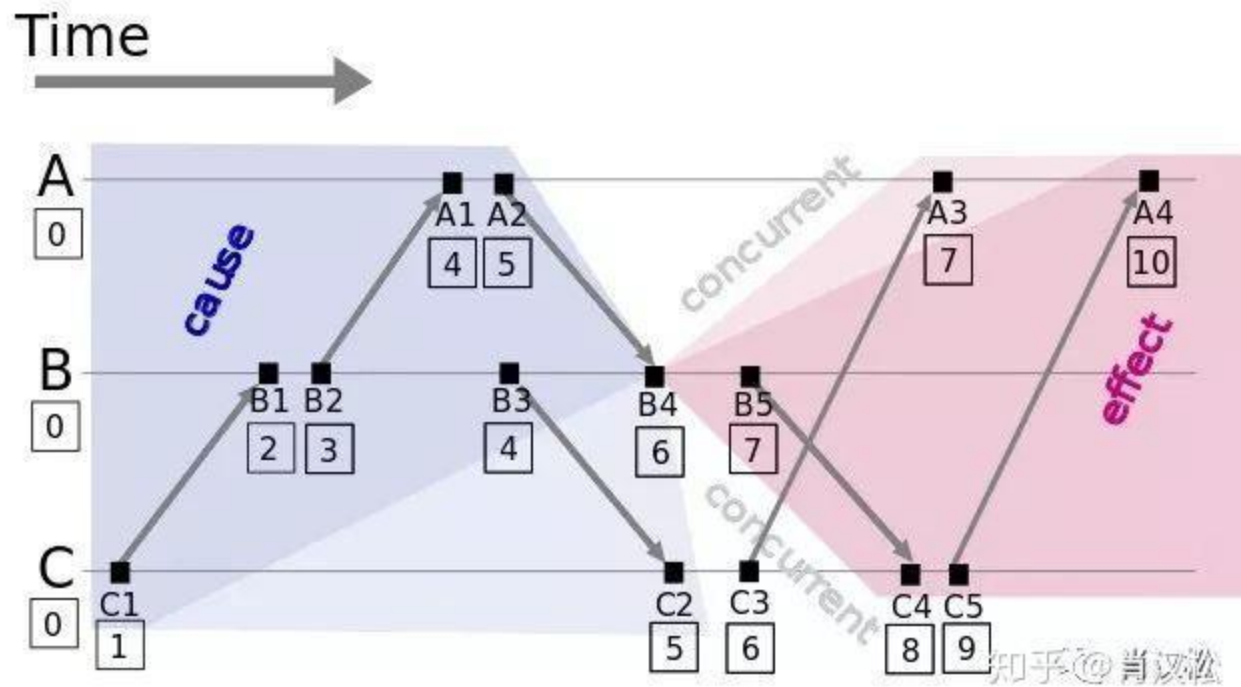
HB关系：（忽略同线程）

$13 \rightarrow 14$

$12 \rightarrow 15$

(内容回顾) 动态方法-流式算法

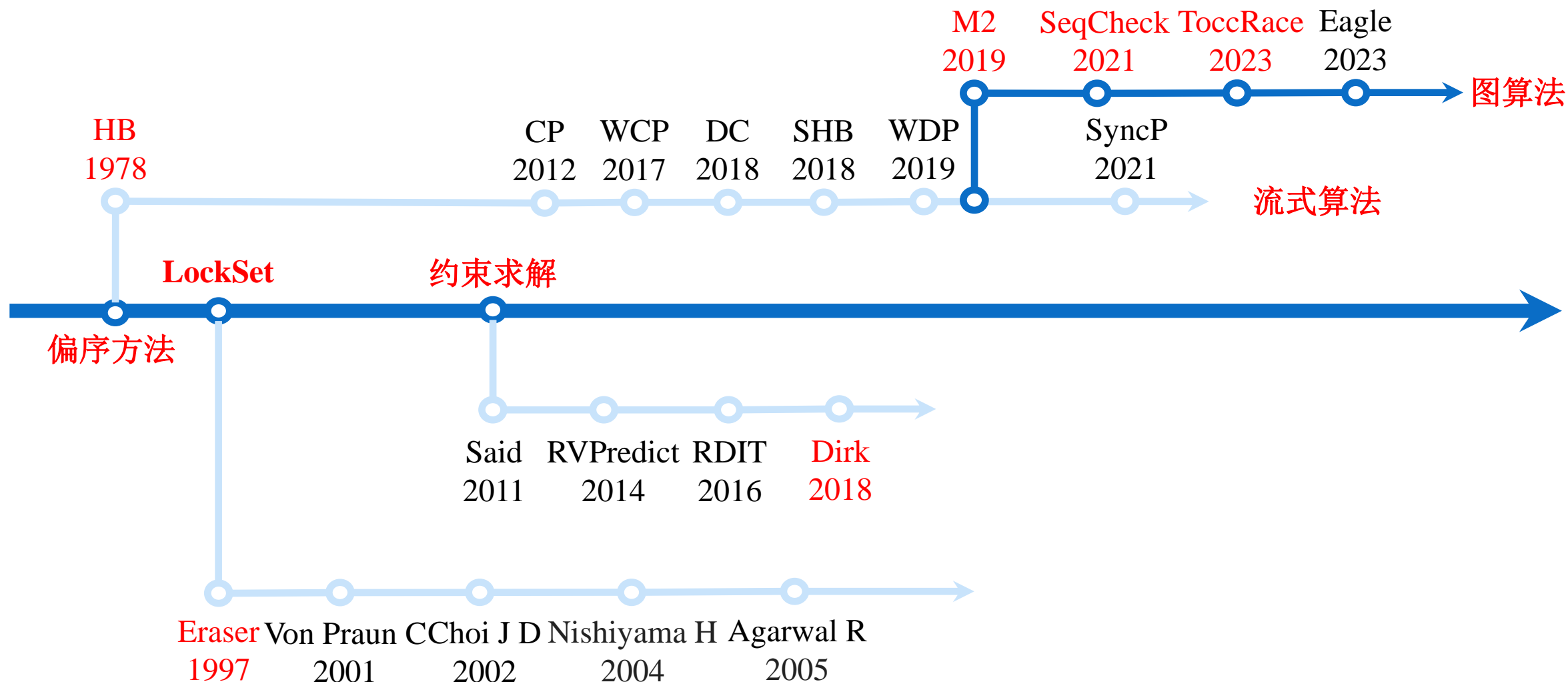
- Lamport clock: 每个线程都用一个integer来模拟自己的时间，对于本地事件或者消息的发送和接收按照以下规则来更新时间：
 - 每一次发生本地事件，该线程的时间+=1
 - 每一次发送消息，发送的线程的时间先+=1，然后再发送。发送的消息中包含它的时间
 - 每一次接收消息，接收的线程先获取消息中带有时间，再和它自己的local时间对比，取最大值后，再+=1作为自己新的local时间





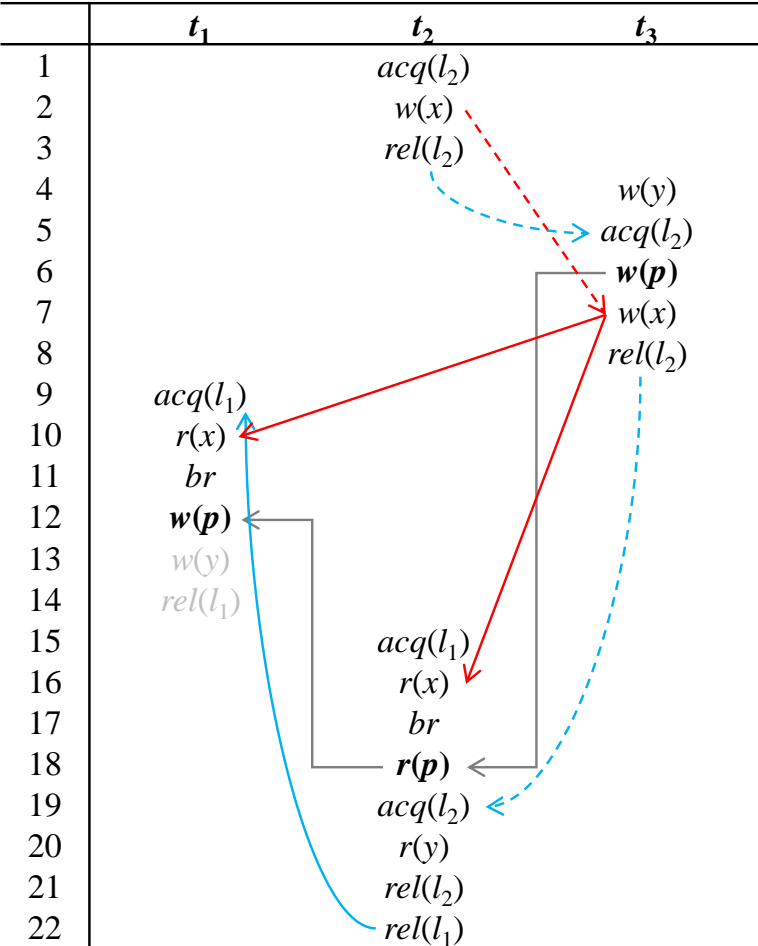
6. 动态方法（第二部分）

并发缺陷检测-动态方法



动态方法-图算法

已知 $\sigma = \langle e_1, \dots, e_{22} \rangle$, $\rho = \langle e_6, e_{18}, e_{12} \rangle$, $\mathcal{A} = \emptyset$ 。判断 ρ 能否真实发生。



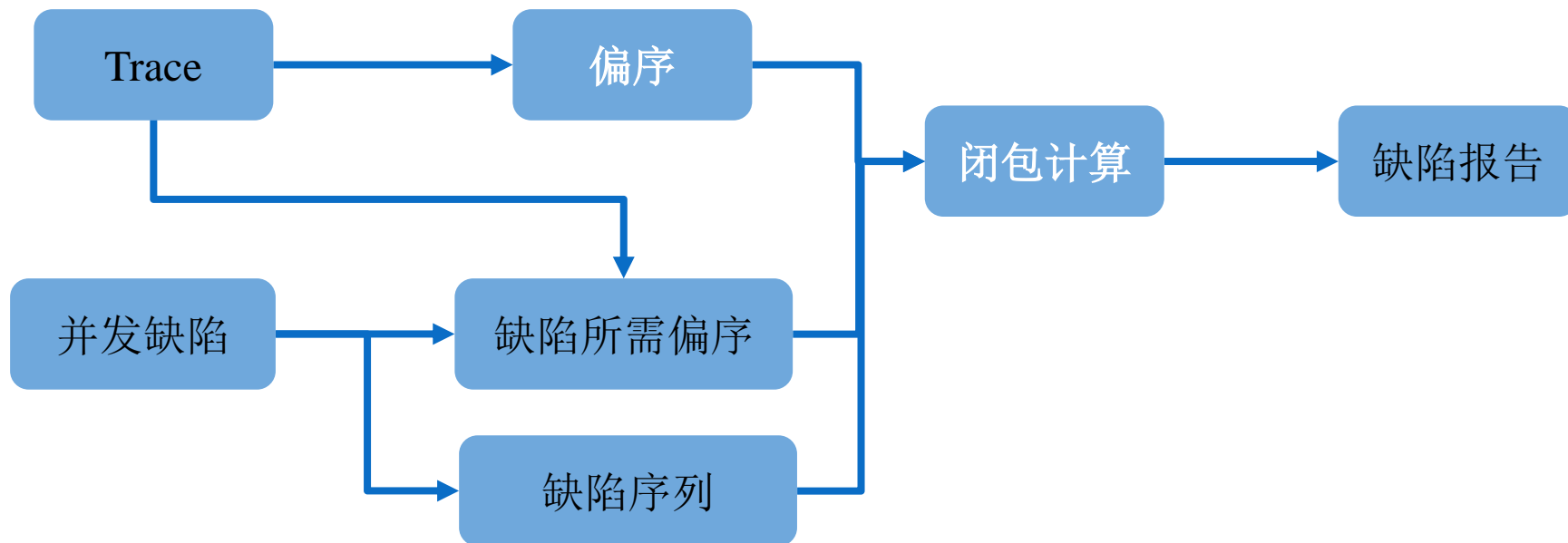
偏序关系图

	t_1	t_2	t_3
1		$acq(l_2)$	
2		$w(x)$	
3		$rel(l_2)$	
4			$w(y)$
5			$acq(l_2)$
6			$w(p)$
7			$w(x)$
8			$rel(l_2)$
9		$acq(l_1)$	
10		$r(x)$	
11		br	
12		$r(p)$	
13		$acq(l_2)$	
14		$r(y)$	
15		$rel(l_2)$	
16		$rel(l_1)$	
17	$acq(l_1)$		
18	$r(x)$		
19	br		
20	$w(p)$		
21	$w(y)$		
22	$rel(l_1)$		

可行序列

动态方法-图算法

- 利用图算法取代向量时钟算法，可以探索更大的线程交错空间
- 算法流程：给定执行序列
 - 首先提取偏序（偏序种类增多，精细度提高）
 - 然后找出潜在的并发缺陷（例如可能构成数据竞争的事件对），构建触发数据竞争的序列片段以及所需的相应偏序
 - 以事件为结点，以偏序关系为有向边，构建有向图
 - 结合执行序列偏序、数据竞争序列片段、数据竞争所需偏序，在图上执行闭包计算，检查是否有环路



动态方法-图算法

- 基于偏序和图算法的并发缺陷检测技术: M2 + SeqCheck
- 偏序: 给定 trace σ
 - Program Order $<_{PO}$: $e_1.tid = e_2.tid$ 且在 σ 中 e_1 发生在 e_2 之前, 则 $e_1 <_{PO} e_2$
 - Observation Order $<_{OO}$: 读事件 e_1 , $e_2 = obs(e_1)$, 则 $e_2 <_{OO} e_1$
 - Lock Order $<_{LO}$: 对同一锁对象的两个 acquire 事件 e_1 和 e_2 , 则 $match(e_1) <_{LO} e_2$ 或者 $match(e_2) <_{LO} e_1$
- Feasible Set (FSet): 给定 trace σ , 用 ε_σ 表示 σ 中的事件的集合, 给定一个事件集合 X 为 ε_σ 的子集, 如果 X 满足以下条件, 则 X 是一个 Fset。
 - Prefix Closed: 如果事件 e 属于 X , 则与事件 e 同线程的所有发生在 e 之前的事件都属于 X
 - Observation Feasible: 任意读事件 e 属于 X , 则 $obs(e) \in X$
 - Lock Feasible:
 - 任意锁释放事件 e 属于 X , 则 $match(e) \in X$
 - 任意锁获取事件 e_1, e_2 属于 X , 如果 $match(e_1)$ 和 $match(e_1)$ 不属于 X , 则 $e_1.var \neq e_2.var$

动态方法-图算法

OO: $e_2 <_{OO} e_5, e_9 <_{OO} e_{10}$

LO: $\{e_3 <_{LO} e_4\}$

Fset:

给定初始集合 $\{e_8, e_9\}$,
可以计算得到 $Fset = \{e_{1-9}\}$ 。

	t_1	t_2
e_1	lock(l)	
e_2	p = 0;	
e_3	unlock(l)	
e_4		lock(l)
e_5		if (p == 0)
e_6		printf(...);
e_7		unlock(l)
e_8		p = 1;
e_9	p = 0	
e_{10}		x = p;

动态方法-图算法

- Trace-closed Partial Order (TCP0): 给定 trace σ , 一个 Fset X , 以及 X 上的偏序集合 P , 如果 P 满足一下条件, 则称 P 为 X 上的 TCP0.
 - P 包含所有 X 上的 Program Order
 - 任意读事件 e 属于 X , 则 $\text{obs}(e) <_P e$
 - 任意属于 X 的针对同一锁的获取事件 e_1 和释放事件 e_2 , 如果 $\text{match}(e_1)$ 不属于 X , 则 $e_2 <_P e_1$
 - Observation-closed: 给定属于 X 的针对共享变量 v 的读事件 e 以及配对的写事件 w , X 中任意对 v 的其他写事件 e' , 则 $w <_P e <_P e'$ 或者 $e' <_P w <_P e$.
 - Lock-closed: 任意属于 X 的针对同一锁的两个释放事件 e_1 和 e_2 , 如果 $\text{match}(e_1) <_P e_2$, 则 $e_1 <_P \text{match}(e_2)$
- 给定一个 Fset X , 以及 X 上的 TCP0 P , 在 X 上对 P 进行闭包, 然后执行线性化, 可以得出 feasible trace。

动态方法-图算法

OO: $e_2 <_{OO} e_5, e_9 <_{OO} e_{10}$

LO: $\{e_3 <_{LO} e_4\}$

Fset:

给定初始集合 $\{e_8, e_9\}$,

可以计算得到 $Fset = \{e_{1-9}\}$ 。

TCPO: (省略PO)

$e_2 <_P e_5$

$e_3 <_P e_4$

$e_2 <_P e_5 <_P e_8$

$e_2 <_P e_5 <_P e_9$

	t_1	t_2
e_1	lock(l)	
e_2	p = 0;	
e_3	unlock(l)	
e_4		lock(l)
e_5		if (p == 0)
e_6		printf(...);
e_7		unlock(l)
e_8		p = 1;
e_9	p = 0	
e_{10}		x = p;

动态方法-图算法

- 构建缺陷序列：给定冲突事件对 (e_1, e_2) ，且在原trace中， e_1 发生在前，令 e'_1, e''_1 分别为发生在 e_1 前后的同线程的紧邻的事件。
 - 构建数据竞争序列： $\langle e'_2, e_1, e_2, e''_1 \rangle$ 以及 $\langle e'_1, e_2, e_1, e''_2 \rangle$
 - 偏序分别为： $e'_2 <_P e_1 <_P e_2 <_P e''_1$ 和 $e'_1 <_P e_2 <_P e_1 <_P e''_2$
- 这种构建方法保证了，冲突事件对中间不会发生其他的事件。
- 对两组序列和偏序分别计算闭包，如果图中不存在环，则 (e_1, e_2) 构成数据竞争。两组序列保证了两个事件的顺序可以交换。

动态方法-图算法

- 给定冲突事件对 (e_8 , e_9)
- 数据竞争序列为: $\langle e_3, e_8, e_9, e_{10} \rangle$ 和 $\langle e_7, e_9, e_8 \rangle$
- 偏序为: $e_3 <_P e_8 <_P e_9 <_P e_{10}$ 和 $e_7 <_P e_9 <_P e_8$ 。
- 闭包计算可知图中不存在环路, 因此 e_8, e_9 构成数据竞争。

Fset: $\{e_{1-9}\}$ 。

TCPO: (省略PO)

$e_2 <_P e_5$

$e_3 <_P e_4$

$e_2 <_P e_5 <_P e_8$

$e_2 <_P e_5 <_P e_9$

	t_1	t_2
e_1	lock(1)	
e_2	p = 0;	
e_3	unlock(1)	
e_4		lock(1)
e_5		if (p == 0)
e_6		printf(...);
e_7		unlock(1)
e_8		p = 1;
e_9	p = 0	
e_{10}		x = p;

动态方法-图算法

OO: $e_2 <_{OO} e_5, e_9 <_{OO} e_{10}$

LO: $\{e_3 <_{LO} e_4\}$

Fset:

给定初始集合 $\{e_9, e_{11}\}$,

可以计算得到 $Fset = \{e_{1-11}\}$ 。

TCPO: (省略PO)

$e_2 <_P e_5$

$e_9 <_P e_{10}$

$e_3 <_P e_4$

$e_2 <_P e_5 <_P e_8$

$e_2 <_P e_5 <_P e_9$

$e_8 <_P e_9 <_P e_{10}$

数据竞争序列为: $\langle e_{10}, e_9, e_{11} \rangle$ 和 $\langle e_3, e_{11}, e_9 \rangle$

偏序为: $e_{10} <_P e_9 <_P e_{11}$ 和 $e_3 <_P e_{11} <_P e_9$ 。

(e_9, e_{11}) 事实上也构成数据竞争

由于偏序 $e_9 <_{OO} e_{10}$ 的存在, 使得闭包计算时出现了环路, 因此漏报。

	t_1	t_2
e_1	lock(l)	
e_2	p = 0;	
e_3	unlock(l)	
e_4		lock(l)
e_5		if (p == 0)
e_6		printf(xx);
e_7		unlock(l)
e_8		p = 1;
e_9	p = 0	
e_{10}		tmp = p; // p++
e_{11}		p = tmp + 1;

动态方法-图算法

- 基于偏序和图算法的并发缺陷检测技术：SeqCheck (FSE 2021)
- 在M2基础上放松了偏序限制，思想为：如果读事件后续不存在分支事件，则该读事件即使的值即时发生变化，也不会影响控制流，此时忽略该读事件对应的观察偏序可以保证结果sound。
- 用 $\text{aftBr}(e)$ 表示同线程中发生在 e 后边的第一个分支事件
- Feasible Set (FSet)：给定trace σ ，用 ε_σ 表示 σ 中的事件的集合，给定一个事件集合 X 为 ε_σ 的子集，如果 X 满足以下条件，则 X 是一个Fset。
 - Prefix Closed：如果事件 e 属于 X ，则与事件 e 同线程的所有发生在 e 之前的事件都属于 X
 - Observation Feasible：任意读事件 e 属于 X ，如果 $\text{aftBr}(e) \in X$ ，则 $\text{obs}(e) \in X$
 - Lock Feasible：
 - 任意锁释放事件 e 属于 X ，则 $\text{match}(e) \in X$
 - 任意锁获取事件 e_1, e_2 属于 X ，如果 $\text{match}(e_1)$ 和 $\text{match}(e_2)$ 不属于 X ，则 $e_1.\text{var} \neq e_2.\text{var}$

动态方法-图算法

- Trace-closed Partial Order (TCP0): 给定 trace σ , 一个 Fset X , 以及 X 上的偏序集合 P , 如果 P 满足一下条件, 则称 P 为 X 上的 TCP0.
 - P 包含所有 X 上的 Program Order
 - 任意读事件 e 属于 X , 如果 $aftBr(e) \in X$, 则 $obs(e) <_P e$
 - 任意属于 X 的针对同一锁的获取事件 e_1 和释放事件 e_2 , 如果 $match(e_1)$ 不属于 X , 则 $e_2 <_P e_1$
 - Observation-closed: 给定属于 X 的针对共享变量 v 的读事件 e 以及配对的写事件 w , X 中任意对 v 的其他写事件 e' , 如果 $aftBr(e) \in X$, 则 $w <_P e <_P e'$ 或者 $e' <_P w <_P e$ 。
 - Lock-closed: 任意属于 X 的针对同一锁的两个释放事件 e_1 和 e_2 , 如果 $match(e_1) <_P e_2$, 则 $e_1 <_P match(e_2)$
- 给定一个 Fset X , 以及 X 上的 TCP0 P , 在 X 上对 P 进行闭包, 然后执行线性化, 可以得出 feasible trace。

动态方法-图算法

OO: $e_2 <_{OO} e_5, e_9 <_{OO} e_{10}$

LO: $\{e_3 <_{LO} e_4\}$

Fset:

给定初始集合 $\{e_9, e_{10}\}$,
可以计算得到 $Fset = \{e_{1-10}\}$ 。

TCPO: (省略PO)

$e_2 <_p e_5$

~~$e_9 <_p e_{10}$~~

$e_3 <_p e_4$

$e_2 <_p e_5 <_p e_8$

$e_2 <_p e_5 <_p e_9$

~~$e_8 <_p e_9 <_p e_{10}$~~

	t_1	t_2
e_1	lock(l)	
e_2	p = 0;	
e_3	unlock(l)	
e_4		lock(l)
e_5		if (p == 0)
e_6		printf(xx);
e_7		unlock(l)
e_8		p = 1;
e_9	p = 0	
e_{10}		x = p++;

SeqCheck优化后, 计算闭包时可以移除一部分偏序, 因此能够检测出数据竞争。

动态方法-图算法

➤ 图算法的优势：

- 运行效率较高：时间复杂度通常是多项式的
- 准确度高：可以保证结果sound（即不存在误报）
- 可扩展性较强：可以拓展到大型应用程序上
- 缺陷覆盖率高：相比于流式算法，基于图的算法不需要固定执行前缀，在计算时大量缩减了偏序，在有一定量trace的情况下，可以达到极高的缺陷覆盖率。

➤ 图算法的不足：

- 必须在现有执行的相同的控制流下检测并发缺陷，限制了缺陷覆盖率，即便SeqCheck一定程度上放宽了条件，仍然存在大量的限制。

7. 最新技术与研究趋势

最新技术与研究趋势

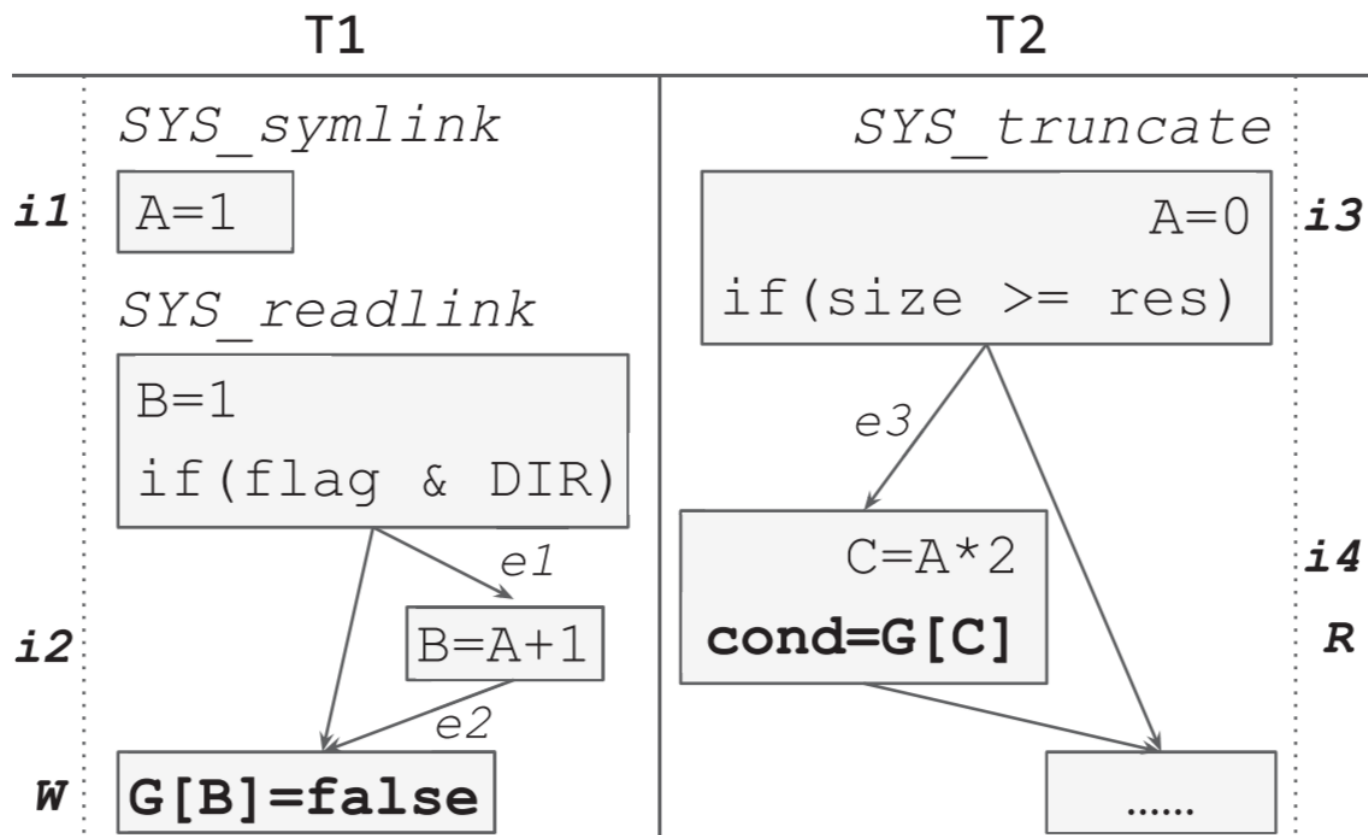
- 最新的技术主要集中在两个方向:
 - 集成并发特性的模糊测试
 - 在图算法基础上的改进与优化

并发导向的模糊测试

- KRACE (S&P 2020)
- 主要思想：在运行程序获取覆盖率的同时，记录线程相关信息
- 具体做法：
 - 记录分支覆盖率（与AFL相同），同时记录对于线程共享变量的并发读写依赖关系的覆盖率
 - 将一条读/写指令标记为 $\langle ix, iy \rangle$ ， ix 标识指令的位置， iy 标识其所属的线程ID
 - 如果发现一条读取指令 $\langle ix2, iy2 \rangle$ 读取另一条写入指令 $\langle ix1, iy1 \rangle$ 写入的值，且这两条指令属于不同线程（ $iy1 \neq iy2$ ），称这两条指令存在并发读写依赖，并记录 $ix1 \rightarrow ix2$ 的依赖边
 - 使用有向图记录所有的依赖边，通过观察一次程序运行是否向有向图中插入新的依赖边，判断此次运行是否触发了新的运行路径。

并发导向的模糊测试

- KRACE (S&P 2020)



并发导向的模糊测试

- KRACE (S&P 2020)

<table><tr><th>T1</th><th>T2</th></tr><tr><td>A=1</td><td></td></tr><tr><td>B=A+1</td><td></td></tr><tr><td></td><td>A=0</td></tr><tr><td></td><td>C=A*2</td></tr><tr><td>①</td><td>B=2, C=0</td></tr><tr><td></td><td><nil></td></tr></table>	T1	T2	A=1		B=A+1			A=0		C=A*2	①	B=2, C=0		<nil>	<table><tr><th>T1</th><th>T2</th></tr><tr><td>A=1</td><td></td></tr><tr><td></td><td>A=0</td></tr><tr><td>B=A+1</td><td></td></tr><tr><td></td><td>C=A*2</td></tr><tr><td>②</td><td>B=1, C=0</td></tr><tr><td></td><td>i3→i2</td></tr></table>	T1	T2	A=1			A=0	B=A+1			C=A*2	②	B=1, C=0		i3→i2	<table><tr><th>T1</th><th>T2</th></tr><tr><td>A=1</td><td></td></tr><tr><td></td><td>A=0</td></tr><tr><td></td><td>C=A*2</td></tr><tr><td>B=A+1</td><td></td></tr><tr><td>③</td><td>B=1, C=0</td></tr><tr><td></td><td>i3→i2</td></tr></table>	T1	T2	A=1			A=0		C=A*2	B=A+1		③	B=1, C=0		i3→i2
T1	T2																																											
A=1																																												
B=A+1																																												
	A=0																																											
	C=A*2																																											
①	B=2, C=0																																											
	<nil>																																											
T1	T2																																											
A=1																																												
	A=0																																											
B=A+1																																												
	C=A*2																																											
②	B=1, C=0																																											
	i3→i2																																											
T1	T2																																											
A=1																																												
	A=0																																											
	C=A*2																																											
B=A+1																																												
③	B=1, C=0																																											
	i3→i2																																											
<table><tr><th>T1</th><th>T2</th></tr><tr><td></td><td>A=0</td></tr><tr><td></td><td>C=A*2</td></tr><tr><td>A=1</td><td></td></tr><tr><td>B=A+1</td><td></td></tr><tr><td>④</td><td>B=2, C=0</td></tr><tr><td></td><td><nil></td></tr></table>	T1	T2		A=0		C=A*2	A=1		B=A+1		④	B=2, C=0		<nil>	<table><tr><th>T1</th><th>T2</th></tr><tr><td></td><td>A=0</td></tr><tr><td>A=1</td><td></td></tr><tr><td></td><td>C=A*2</td></tr><tr><td>B=A+1</td><td></td></tr><tr><td>⑤</td><td>B=2, C=2</td></tr><tr><td></td><td>i1→i4</td></tr></table>	T1	T2		A=0	A=1			C=A*2	B=A+1		⑤	B=2, C=2		i1→i4	<table><tr><th>T1</th><th>T2</th></tr><tr><td></td><td>A=0</td></tr><tr><td>A=1</td><td></td></tr><tr><td>B=A+1</td><td></td></tr><tr><td></td><td>C=A*2</td></tr><tr><td>⑥</td><td>B=2, C=2</td></tr><tr><td></td><td>i1→i4</td></tr></table>	T1	T2		A=0	A=1		B=A+1			C=A*2	⑥	B=2, C=2		i1→i4
T1	T2																																											
	A=0																																											
	C=A*2																																											
A=1																																												
B=A+1																																												
④	B=2, C=0																																											
	<nil>																																											
T1	T2																																											
	A=0																																											
A=1																																												
	C=A*2																																											
B=A+1																																												
⑤	B=2, C=2																																											
	i1→i4																																											
T1	T2																																											
	A=0																																											
A=1																																												
B=A+1																																												
	C=A*2																																											
⑥	B=2, C=2																																											
	i1→i4																																											

并发导向的模糊测试

- KRACE (S&P 2020)
- 种子变异: 属性变异、增、删、线程替换、合并多个种子



并发导向的模糊测试

- KRACE (S&P 2020)

- 种子变异:

Seed 1

```
mkdir(|p-1|, 0777)
mknod(|p-2|, 0333, 0)
open(|p-1|, O_DIR..., 0777) = <fd1>
dup2(<fd1>, |fd2|) = <fd3>
close(<fd1>)
symlink(|p-1|, |p-3|)
close(<fd3>)
```

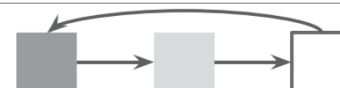
Seed 2

```
create(|p-1|, 0777) = <fd1>
link(|p-1|, |p-2|)
write(<fd1>, [...buffer...], 2036)
open(|p-2|, O_PATH..., 0777) = <fd2>
truncate(|p-1|, 5736)
fsync(<fd2>)
```

Combined Seed

```
create(|p-1|, 0777) = <fd1>
mkdir(|p-2|, 0777)
mknod(|p-3|, 0333, 0)
open(|p-2|, O_DIR..., 0777) = <fd2>
link(|p-1|, |p-4|)
write(<fd1>, [...buffer...], 2036)
dup2(<fd2>, |fd3|) = <fd4>
open(|p-4|, O_PATH..., 0777) = <fd5>
truncate(|p-1|, 5736)
close(<fd2>)
symlink(|p-2|, |p-5|)
close(<fd4>)
fsync(<fd5>)
```

Seed 2 thread
shuffling rotation:



子

并发导向的模糊测试

- Period (ICSE 2022)
- 主要思想：先算出调度方案，然后在对应程序点**主动添加延时**，以探索不同线程交错方案

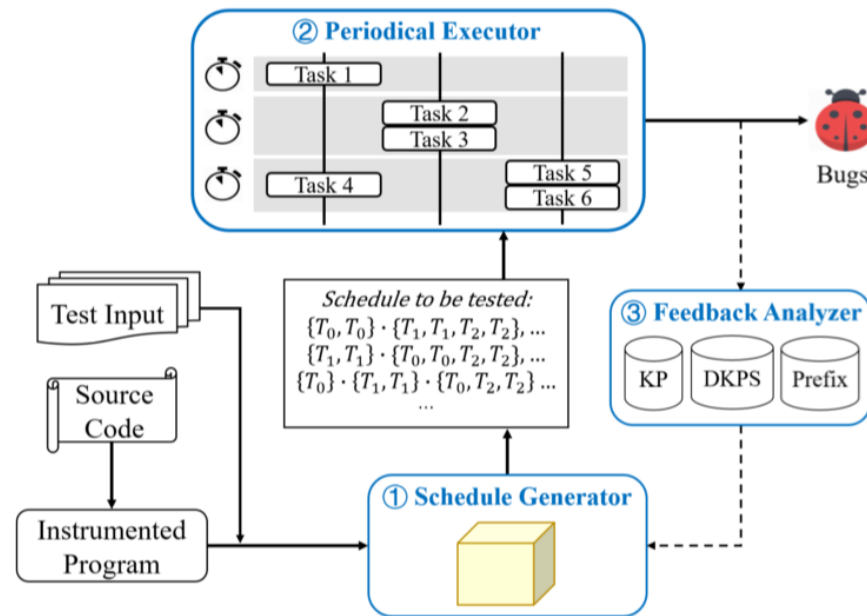
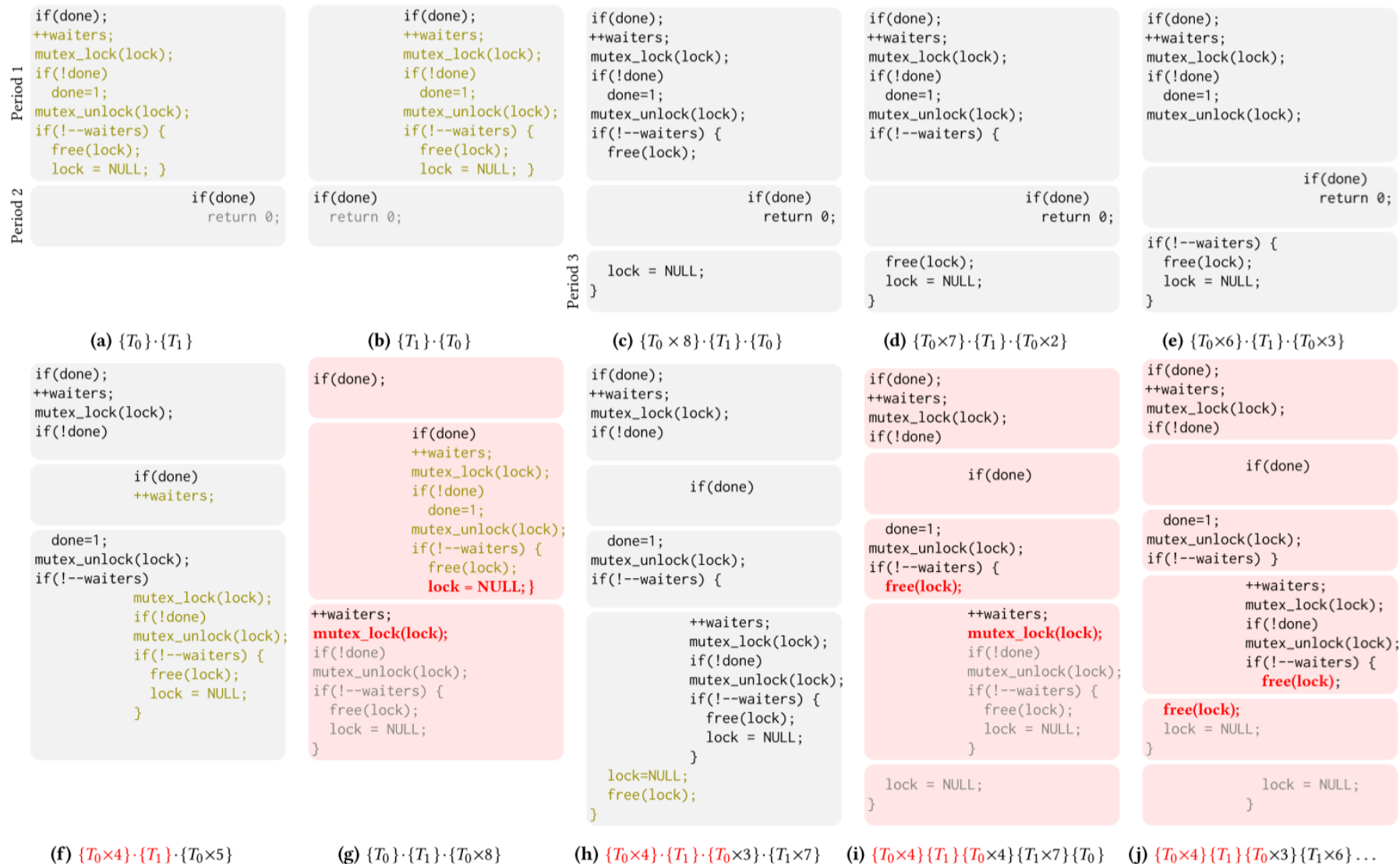


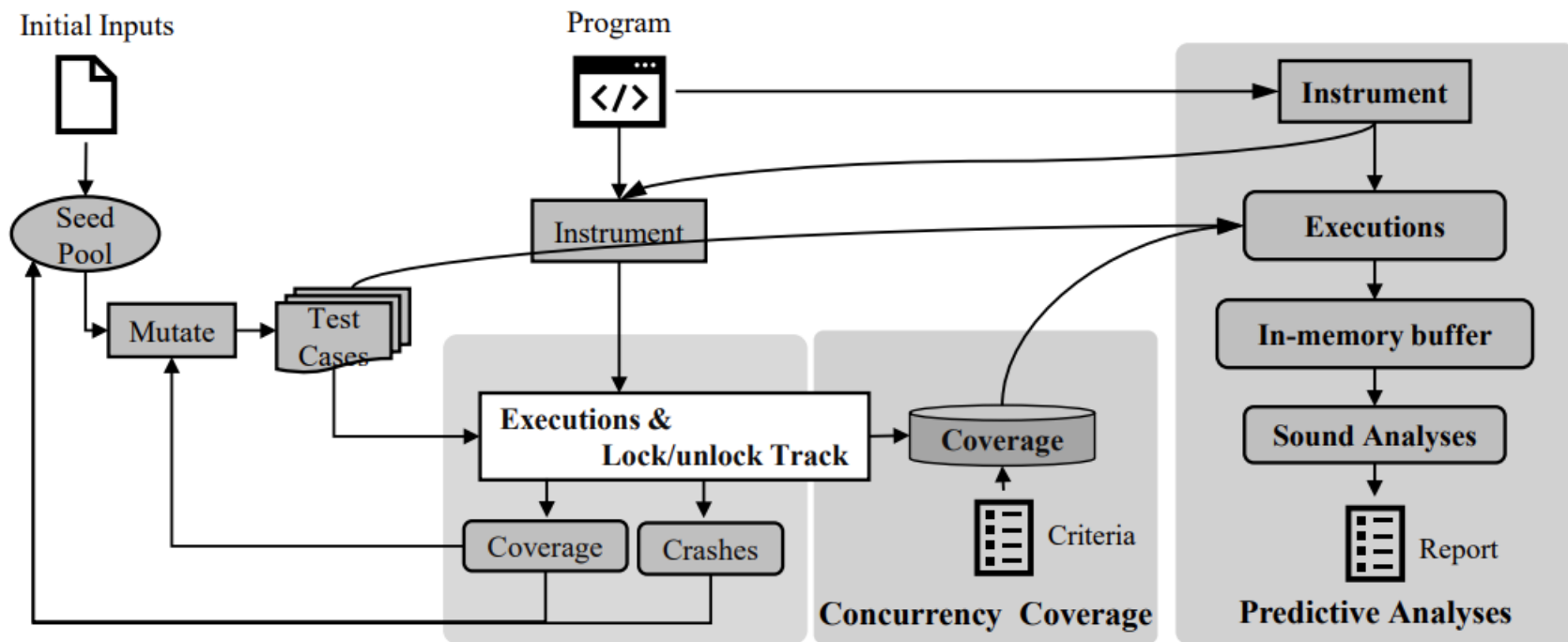
Figure 2: The workflow of PERIOD.

并发导向的模糊测试



并发导向的模糊测试

- PredFuzz (COMPSAC 2023)
- 主要思想：考虑**锁的覆盖率**，结合**模糊测试**和**预测性漏洞检测**



并发导向的模糊测试

- PredFuzz (COMPSAC 2023)
- 主要思想：考虑锁的覆盖率，结合模糊测试和预测性漏洞检测

1. <code>int* p;</code>	18. <code>void t1()</code>
2. <code>Lock* A;</code>	19. <code>{</code>
3. <code>Lock* B;</code>	20. <code>Lock(A);</code>
4. <code>Lock ls[2];</code>	21. <code>... //operations on</code>
5. <code>void main(int arg)</code>	22. <code>if(p != NULL)</code>
6. <code>{</code>	23. <code>free(p);</code>
7. <code> //assert (arg>=0);</code>	24. <code>unlock(A);</code>
8. <code> p = malloc(arg*sizeof(int));</code>	25. <code>}</code>
9. <code> ls[0] = new Lock();</code>	26. <code></code>
10. <code> ls[1] = new Lock();</code>	27. <code>void t2()</code>
11. <code></code>	28. <code>{</code>
12. <code> A=ls[0];</code>	29. <code>Lock(B);</code>
13. <code> B=ls[(arg / 20) % 2];</code>	30. <code>... //operations on B</code>
14. <code></code>	31. <code>if(p != NULL)</code>
15. <code> fork(t1);</code>	32. <code>free(p);</code>
16. <code> fork(t2);</code>	33. <code>unlock(B);</code>
17. <code>}</code>	34. <code>}</code>

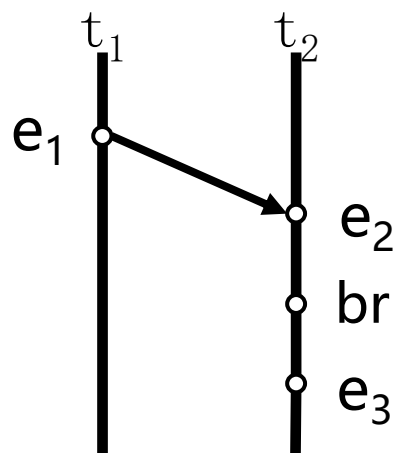
	Keep input?	Con. Vul.?
Input: 0 (0b0000 0000) Coverage: <lock(A), unlock(A), lock(A), unlock(A)>	Yes (AFL++) Yes (PredFuzz)	NO
Input: 1 (0b0000 0001) Coverage: <lock(A), unlock(A), lock(A), unlock(A)>	The same coverage NO (AFL++) NO (PredFuzz)	NO
Input: 32(0b0010 0000) Coverage: <lock(A), unlock(A), lock(B), unlock(B)>	A different one No (AFL++) Yes (PredFuzz)	Yes

最新技术与研究趋势-ToccRace (ICSE'23)


- 传统的基于偏序的算法，必须要求推测出的执行序列与原始执行序列有相同的控制流，因此引入了以下假设：
 - 任意读事件都有可能影响控制流
- 在以上假设基础上，为保证结果sound，所有方法均要求，读事件获取到的值必须保持不变。
- 事实上：
 - 仅有部分读事件会影响控制流
 - 即便控制流发生了变化，存在一些事件，必然发生且其行为不发生变化
- ToccRace针对以上两点进行优化

最新技术与研究趋势-ToccRace (ICSE'23)

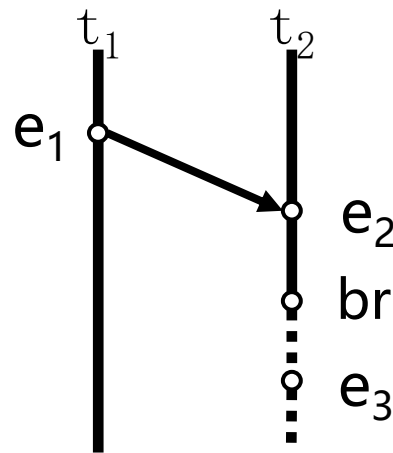
	t_1	t_2
e_1	$p = 0;$	
e_2		$\text{if } (p = 0)$
e_3		$\text{printf}(xx);$
		$p = 1;$



Observed trace
 $(e_2 \text{ reads from } e_1)$
 $(e_1, e_3) : \text{no race}$

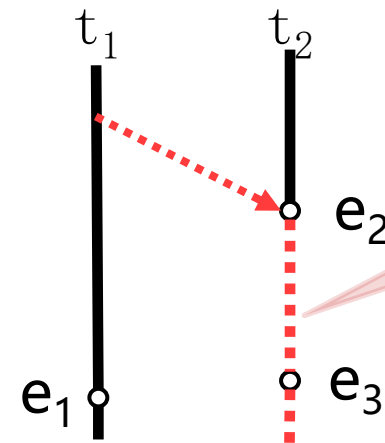
Inference

Same Control-flow

	t_1	t_2
e_1	$p = 0;$	
e_2		$\text{if } (p = 0)$
e_3		$\text{printf}(xx);$
		$p = 1;$



Inferred trace
 $(e_2 \text{ reads from } e_1)$
 $(e_1, e_3) : \text{no race}$
 W-R Event/Value Consistency

	t_1	t_2
e_2		$\text{if } (p = 0)$
e_3		$\text{printf}(xx);$
e_1	$p = 0;$	$p = 1;$



Actual trace
 $(e_2 \text{ reads from other events})$
 $(e_1, e_3) : \text{race}$

Different Control-flow

最新技术与研究趋势-ToccRace (ICSE'23)

	t_1	t_2
1	p = 0;	
2		if (p = 0)
3		printf(xx);
4		p = 1; Must Occur

Inference



	t_1	t_2
1		if (p = 0)
2		printf(xx);
3		p = 1;
4	p = 0;	

Observed order

Line 1 and line 4 : no race

Inferred order

Line 3 and 4 : race

Observation

For any control-flow between
line 2 and line 4 :

Line 4 must occur

Inference

Control-flow changes, but :
“p = 1” still occur

最新技术与研究趋势-ToccRace (ICSE'23)

- Fix-point Event: 给定trace σ , 读事件 e_r , 其中 $match(e_r).tid \neq e_r.tid$, 与 e_r 同线程的内存访问事件 e , 如果满足以下条件, 则称 e 是 e_r 的fix-point event。用 $I(e)$ 表示事件 e 对应的指令。
 - 任意读事件 e'_r , $e_r \leq_{PO} e'_r <_{PO} e$, $I(e)$ 对 $I(e'_r)$ 不存在数据依赖和控制依赖。
 - $I(e_r)$ 到 $I(e)$ 之间, 不可能执行任意同步操作相关的指令(锁、信号量等)
- 如果 e 是 e_r 的fix-point event, 则只要 e_r 发生, e 必然发生且其行为不发生变化。
- Equivalent Pair: 给定事件对 (e_1, e_3) , e_3 是 e_2 的fix-point event, 则称 (e_1, e_2) 是 (e_1, e_3) 的equivalent pair。
 - 通过线程调度可以证明: 如果 (e_1, e_2) 可以并发执行, 则 (e_1, e_3) 必然可以并发执行。
 - 间接验证: 如果冲突对的等价对可以并发执行, 则冲突对必然构成数据竞争。

最新技术与研究趋势-ToccRace (ICSE'23)

a为原始程序，b为一次执行的trace。
图中的Fix-point Event关系？

	t_1	t_2		t_1	t_2		t_1	t_2
1	$p = a;$		e_1	$rd(a)$		1		$if (p[0] < 0)$
2			e_2	$wr(p)$		2		$\{ \text{printf}(""); \}$
3		$if (p[0] < 0)$	e_3		$rd(p)$	3		$p[0] = 1;$
4		$\{ \text{printf}(""); \}$	e_4		$rd(p[0])$	4	$p = a;$	
5			e_5		br	5	$b[0] = 0;$	
6	$b[0] = 0;$		e_6	$rd(b)$		6	$p = b;$	
7	$p = b;$		e_7	$wr(b[0])$		7		
8			e_8	$rd(b)$		8		
9			e_9	$wr(p)$		9		
10		$p[0] = 1;$	e_{10}		$rd(p)$	10		
11			e_{11}		$wr(p[0])$	11		

(a) (b) (c)

Fig. 1: An example with two threads. (p, a, b are global pointers; Init: $p \neq a \neq b, p[0] = 1, a[0] = -1$)

最新技术与研究趋势-ToccRace (ICSE'23)

a为原始程序，b为一次执行的trace。
图中的Fix-point Event关系？

e_{10} 为 e_3 的fix-point event
给定冲突对 (e_2, e_{10}) ，其等价对为 (e_2, e_3)

采用M2和SeqCheck算法分别针对 (e_2, e_3)
计算TCPO？

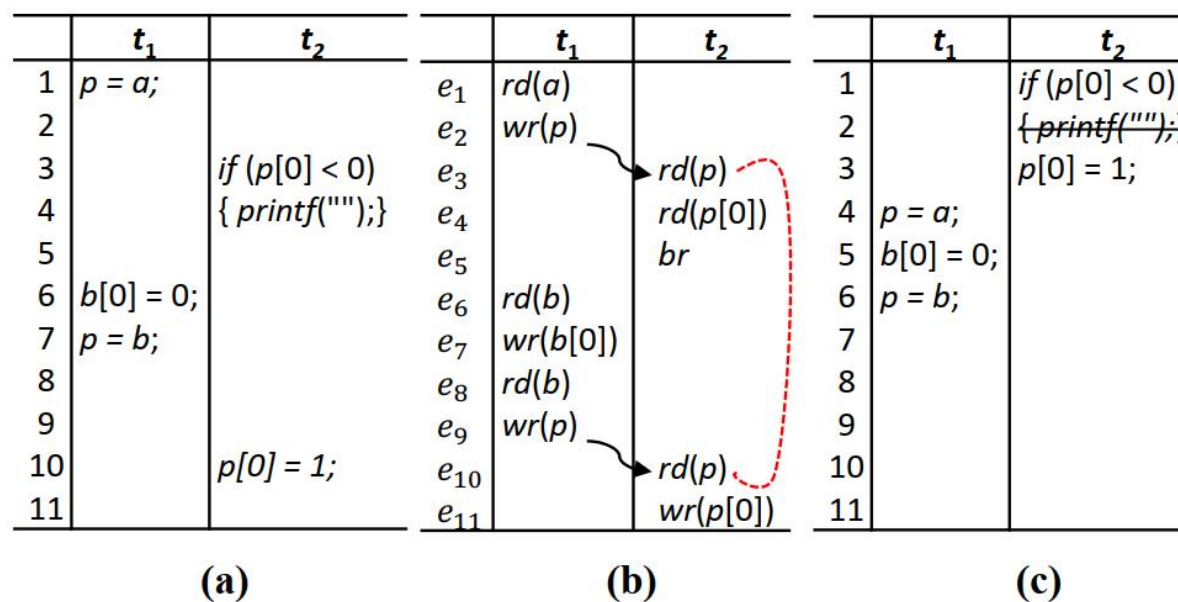


Fig. 1: An example with two threads. (p, a, b are global pointers; Init: $p \neq a \neq b, p[0] = 1, a[0] = -1$)

最新技术与研究趋势-ToccRace (ICSE'23)

a为原始程序，b为一次执行的trace。
图中的Fix-point Event关系？

e_{10} 为 e_3 的fix-point event
给定冲突对 (e_2, e_{10}) ，其等价对为 (e_2, e_3)

采用M2和SeqCheck算法分别针对 (e_2, e_3)
计算TCPO？

M2: $\{e_2 <_P e_3\}$

SeqCheck: $\{\}$

	t_1	t_2		t_1	t_2		t_1	t_2
1	$p = a;$		e_1	$rd(a)$		1		$if (p[0] < 0)$
2			e_2	$wr(p)$		2		$\{ \text{printf}(""); \}$
3		$if (p[0] < 0)$	e_3		$rd(p)$	3		$p[0] = 1;$
4		$\{ \text{printf}(""); \}$	e_4		$rd(p[0])$	4	$p = a;$	
5			e_5		br	5	$b[0] = 0;$	
6	$b[0] = 0;$		e_6	$rd(b)$		6	$p = b;$	
7	$p = b;$		e_7	$wr(b[0])$		7		
8			e_8	$rd(b)$		8		
9			e_9	$wr(p)$		9		
10		$p[0] = 1;$	e_{10}		$rd(p)$	10		
11			e_{11}		$wr(p[0])$	11		

(a) (b) (c)

Fig. 1: An example with two threads. (p, a, b are global pointers; Init: $p \neq a \neq b$, $p[0] = 1$, $a[0] = -1$)

最新技术与研究趋势-ToccRace (ICSE'23)

a为原始程序，b为一次执行的trace。
图中的Fix-point Event关系？

e_{10} 为 e_3 的fix-point event
给定冲突对 (e_2, e_{10}) ，其等价对为 (e_2, e_3)

采用M2和SeqCheck算法分别针对 (e_2, e_3)
计算TCPO？

M2: $\{e_2 <_P e_3\}$

SeqCheck: $\{\}$

→ (e_2, e_3) 可以并发执行
→ (e_2, e_{10}) 构成数据竞争

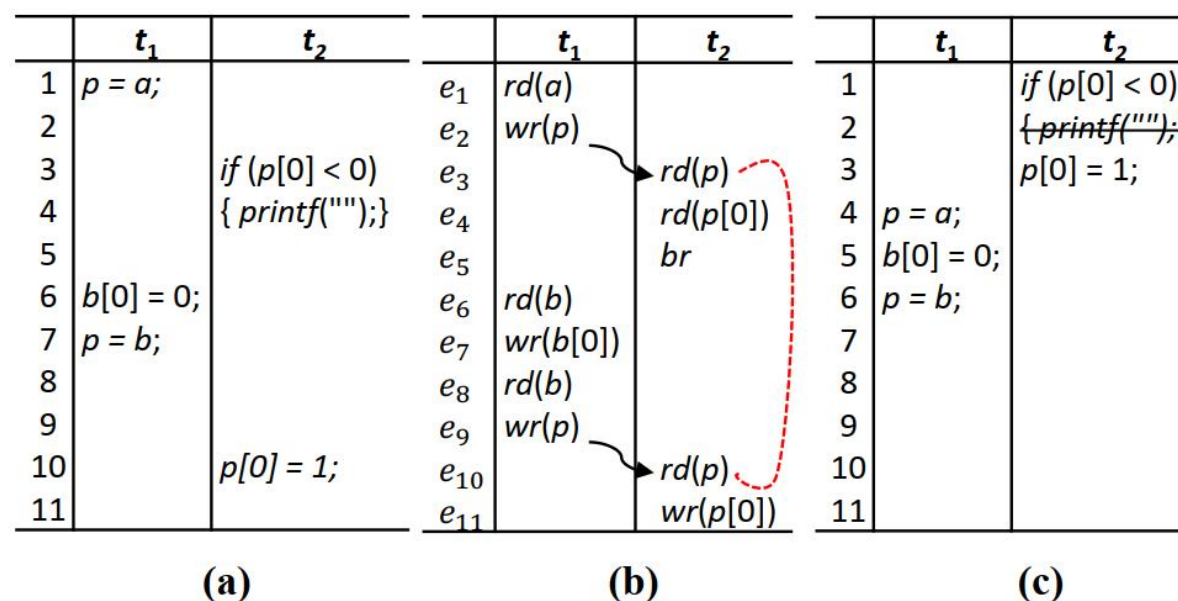
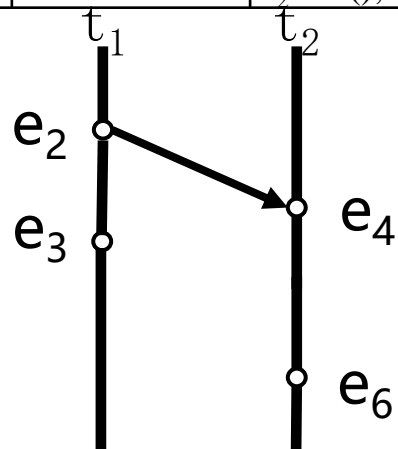


Fig. 1: An example with two threads. (p, a, b are global pointers; Init: $p \neq a \neq b$, $p[0] = 1$, $a[0] = -1$)


最新技术与研究趋势-Eagle (ICSE'24)

	t_1	t_2
e_1	$n_1 = h;$	
e_2	$h = h \rightarrow next;$	
e_3	$n_1.foo();$	
e_4		$n_2 = h;$
e_5		$h = h \rightarrow next;$
e_6		$n_2.foo();$

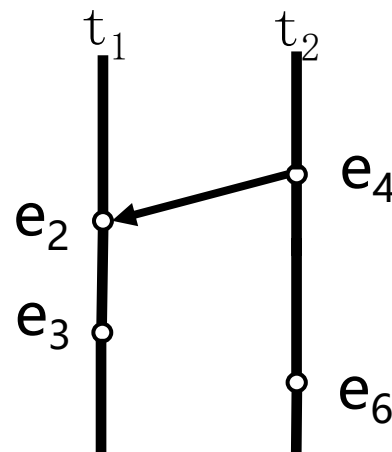


Observed trace

(e_4 reads from $e_2 \Rightarrow n_2 \neq n_1$)
 (e_3, e_6) : unrelated

Inference

 Not
 Considering
 Side Effects

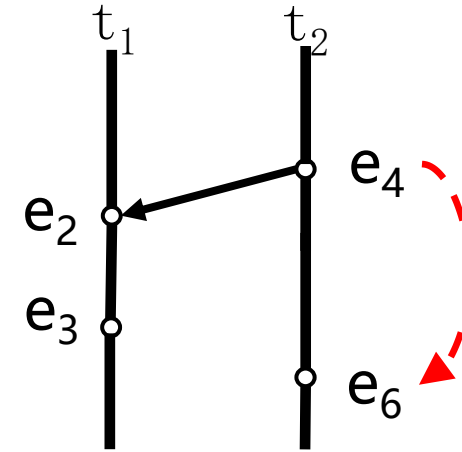
	t_1	t_2
e_1	$n_1 = h;$	
e_2	$h = h \rightarrow next;$	
e_3	$n_1.foo();$	
e_4		$n_2 = h;$
e_5		$h = h \rightarrow next;$
e_6		$n_2.foo();$



Inferred trace

(e_4 **not** reads from e_2)
 (e_3, e_6) : **unrelated**
ERROR!

	t_1	t_2
e_1	$n_1 = h;$	
e_2	$h = h \rightarrow next;$	
e_3	$n_1.foo();$	
e_4		$n_2 = h;$
e_5		$h = h \rightarrow next;$
e_6		$n_2.foo();$



Actual trace

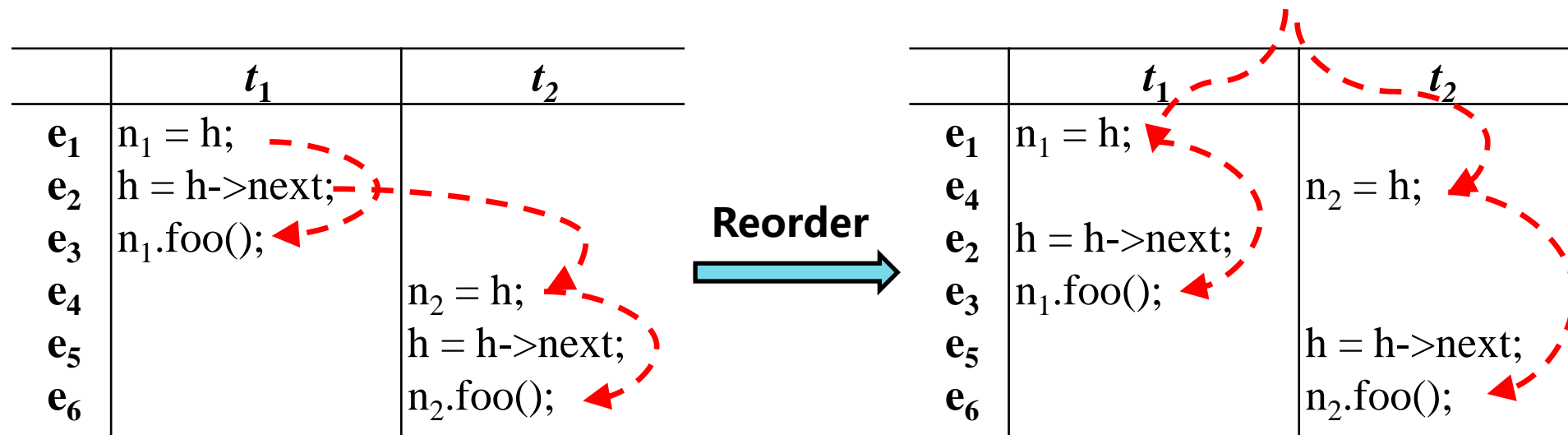
(e_4 **not** reads from $e_2 \Rightarrow n_2 = n_1$)
 (e_3, e_6) : **related & race**

Challenges : Identify side effects & Adjust pointer flow on read-write changes

最新技术与研究趋势-Eagle (ICSE'24)

- 传统方法中，冲突事件对是根据trace中的内存对象的地址静态构建的。
 - 给定一个trace σ ，从属不同线程的两个内存事件 e_1 和 e_2 ，且两个事件中至少一个为写事件，如果 $e_1.var \neq e_2.var$ ，则 (e_1, e_2) 不构成冲突对。
 - 但是在实际运行中，可能存在其他trace ρ ，在 ρ 中， $e_1.var = e_2.var$ ， (e_1, e_2) 构成冲突对。
 - 传统方法无法鉴别以上潜在的冲突对，因此会产生漏报。
-
- Eagle通过分析trace的数据流信息，为共享变量的传递构建数据链，对于在 σ 中不构成冲突对的事件 e_1 和 e_2 ，如果二者的数据链有交集，则二者在其他trace中可能构成冲突对。
 - Eagle通过以上方法，寻找到更多潜在的冲突对，实现了更好地缺陷检测效果。

最新技术与研究趋势-Eagle (ICSE'24)



Observed order

Line 3 and line 6 : no race

Inferred order

Line 3 and 6 : race

Observation

e_1 has side effect on e_3
 e_4 has side effect on e_6
 e_3 and e_6 are unrelated

Inference

e_1 and e_4 read the same value, thus
(1) n_1 and n_2 point to the same object,
(2) e_3 and e_6 are related

并发测试与分析技术总结

各类技术比较（相对）

方法	执行效率	覆盖率	精确性	大规模扩展性
模糊测试	高	低	sound	一般
随机扰动	低	低	sound	一般
数据流分析	高	高	较高	强
符号执行	低	高	高	差
模型检验	极低	高	Sound	极差
LockSet	高	高	差	强
约束求解	低	高	sound	差
流式算法	高	较高	sound	强
图算法	高	高	sound	较强

Q & A

蔡彦

计算机科学国家重点实验室
中国科学院软件研究所

yancai@ios.ac.cn