CS492B Analysis of Concurrent Programs

Code Coverage-based Testing of Concurrent Programs

Prof. Moonzoo Kim

Computer Science, KAIST

Coverage Metric for Software Testing

- A coverage metric defines a set of test requirements on a target program which a complete test should satisfy
  - A *test requirement* (a.k.a., *test obligation*) is a condition over a target program
  - An execution *covers* a test requirement when the execution satisfies the test requirement
  - The *coverage level* of a test (i.e., a set of executions) is the ratio of the test requirements covered by at least one execution to the number of all test requirements
- A coverage metric is used for assessing progress of a test
  - Measure the quality of a test (to assess whether a test is sufficient or not)
  - Detect missing cases of a test (to find next test generation target)

Code Coverage Metric and Test Generation

- A code coverage metric derives test requirements from the elements of a target program code
  - Standard methodology in testing sequential programs
  - E.g. branch/statement coverage metrics

- Many test generation techniques for sequential programs aim to achieve high code coverage fast
  - Empirical studies have shown that a test achieving high code coverage tends to detect more faults in the sequential program testing domain

Concurrency Code Coverage Metric

- Many concurrency coverage metrics have been proposed, which are specialized for concurrent program tests
  - Derive test requirements from synchronization operations or shared memory access operations
- A concurrency coverage metric is a good solution to alleviate the limitation of the random testing techniques

- **Is a test achieving higher concurrency coverage better for detecting faults?**
- **How can we generate concurrent executions to achieve high concurrency coverage?**
- **How can we overcome the limitations of existing concurrency coverage metrics?**

Part I.
The Impact of Concurrent Coverage Metrics on Testing Effectiveness

Code Coverage for Concurrent Programs

- Test requirements of code coverage for concurrent programs capture different thread interaction cases
- Several metrics have been proposed
  - Synchronization coverage:
    ***blocking*, *blocked*, *follows*, *synchronization-pair*,** etc.
  - Data access coverage:
    ***PSet*, *all-use*, *LR-DEF*, *access-pair*, *statement-pair*,** etc.

```
01: int data ;
 …
10: thread1() {         20: thread2() {
11: lock(m);            21:    lock(m);
12: if (data …){        22:    data = 0;
13:  data = 1 ;                ...
       ...              29:    unlock(m);
18: unlock(m);                 ...
       ...
```

***Sync.-Pair*:**
{(11, 21),
(21,11), … }

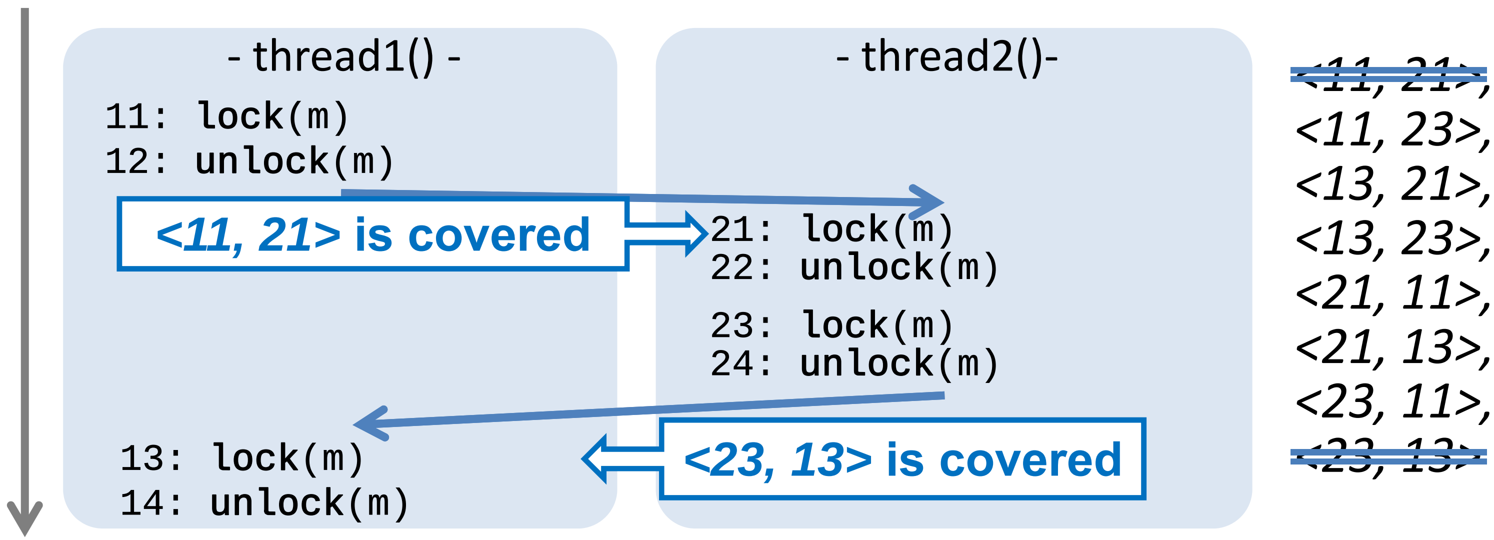
***Stmt.-Pair*:**
{(12, 22),
(22,13), … }

Concurrent Coverage Example – *"follows" Coverage*

- Structure: a requirement has **two code lines of lock operations** $<l_1, l_2>$
- Condition: $<l_1, l_2>$ is covered when 2 different threads hold a lock consecutively at two lines l_1 and l_2

```
10:thread1() {
11: lock(m) ;
12: unlock(m) ;
13: lock(m) ;
14: unlock(m) ;
15:}
```

```
20:thread2() {
21: lock(m) ;
22: unlock(m) ;
23: lock(m) ;
24: unlock(m) ;
25:}
```



Is Concurrent Coverage Good for Testing?

- A common belief about concurrent coverage metrics is that

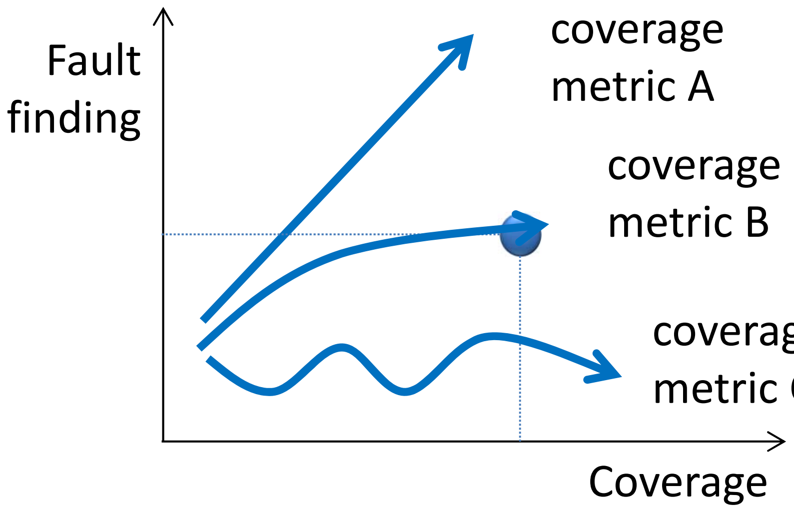
> *"As more test requirements for the metrics are covered, testing becomes more likely to detect faults".*

- A few automated testing techniques for concurrent programs utilize concurrent coverage information
  - Saturation-based testing [Sherman et al., ESEC/FSE 09],
  - Coverage-guided systematic testing [Wang et al., ICSE 11],
  - Coverage-guided thread scheduling [Hong et al., ISSTA 12],
  - Search-based testing w/ concurrent coverage [Krena et al., PADTAD 10]

> Is this hypothesis really true?
>
> - We have to provide empirical evidence

Research Questions 1

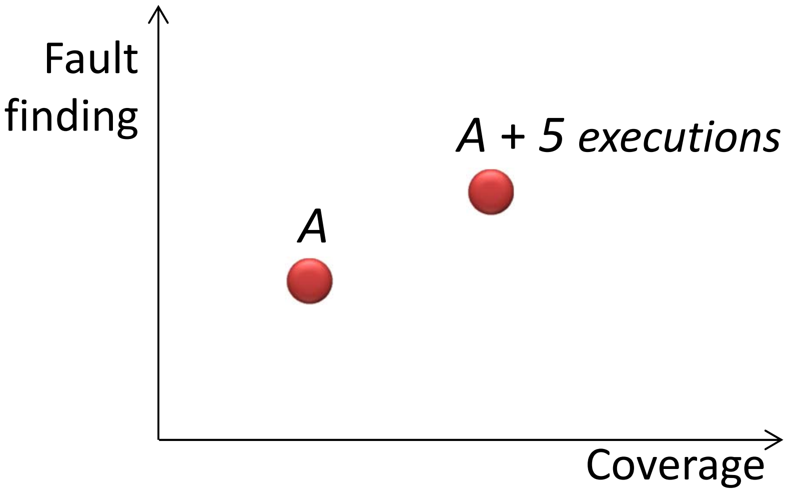
- Does **coverage impact fault finding**?

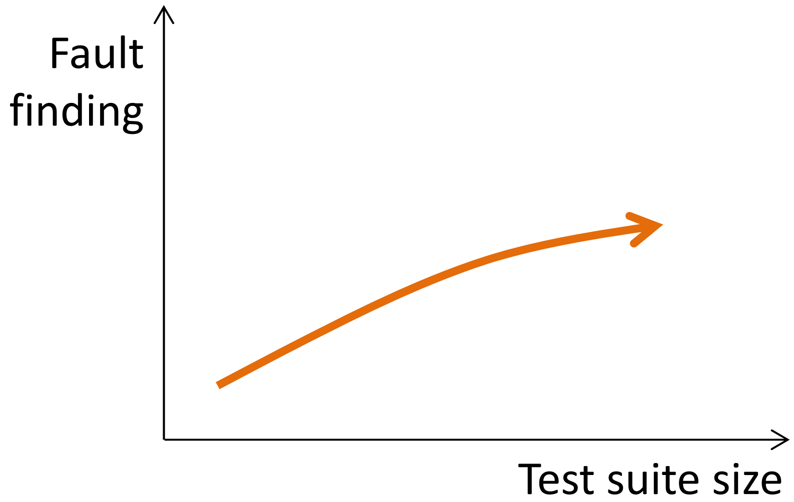


Measure **correlation** of fault finding and coverage to check whether **concurrent coverage is a good predictor of testing effectiveness**

Research Questions 1a

- Does **coverage impact fault finding more than test suite size ?**



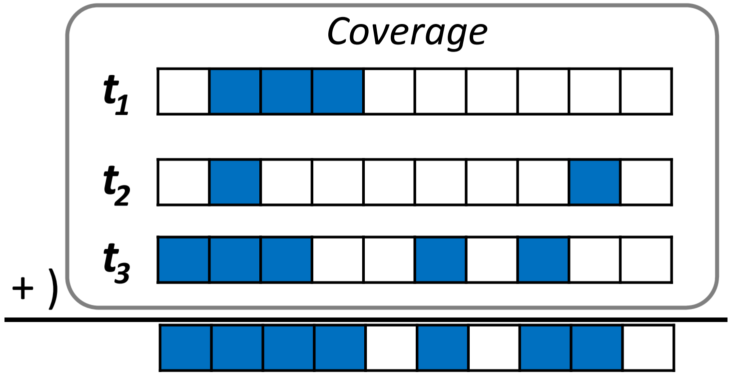


- Because of coverage increase ?
- Because of test size increase?

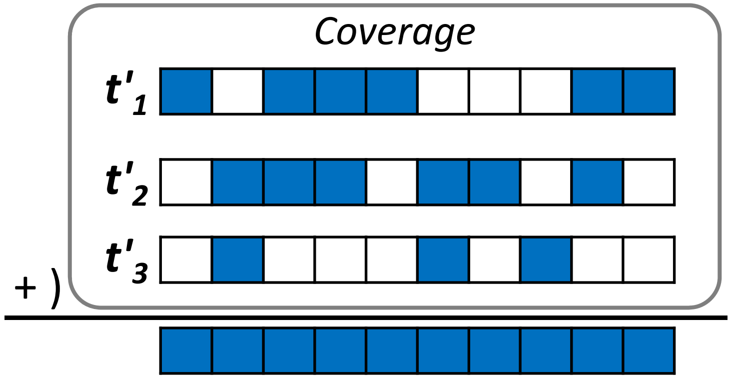
Compare the **correlation** of
fault finding and test suite size

Research Question 2

- Is **testing controlled to have high coverage** more effective than random testing with equal size test suites?



Random test suite:
a test suite having arbitrary three executions



Coverage controlled test suite:
a test suite controlled to have 100% coverage

*Does a **coverage-directed test suite** have **better fault finding ability** than random test suite of equal size?*

Concurrent Coverage Metric Studied

- Study 8 concurrent coverage metrics
  - Select basic & representative metrics from 20 existing metrics
  - The selected coverage metrics are classified with respect to (1) *type of constructs* and (2) *number of code element*

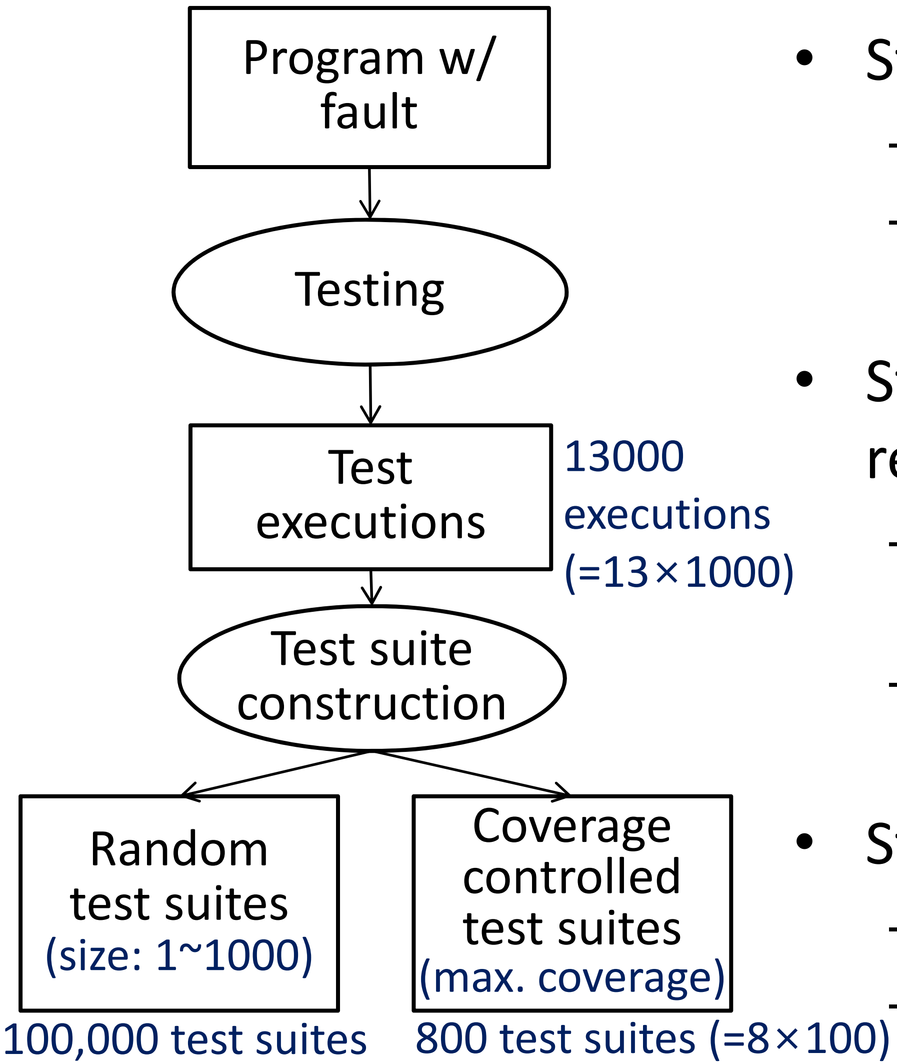
	Synchronization operation	**Data access Operation**
Singular	*blocking*, *blocked* [Edelstein et al., 2012]	*LR-Def* [Lu et al., FSE 07]
Pairwise	*blocked-pair, follows* [Trainin et al, PADTAD 09], *sync-pair* [Hong et al., ISSTA 12]	*Pset* [Yu et al, ISCA 09], *Def-Use* [Tasiran et al., ESE 12]

Experiment Subjects

Type	Program	LOC	Num. threads	Faulty versions
Single fault program	Alarmclock	125	4	1
	Clean	51	3	1
	Piper	71	9	1
	Producerconsumer	87	5	1
	Stringbuffer	416	3	1
	Twostage	52	3	1
Mutation testing	ArrayList	5866	29	42
	BoundedBuffer	1437	31	34
	Vector	709	51	88

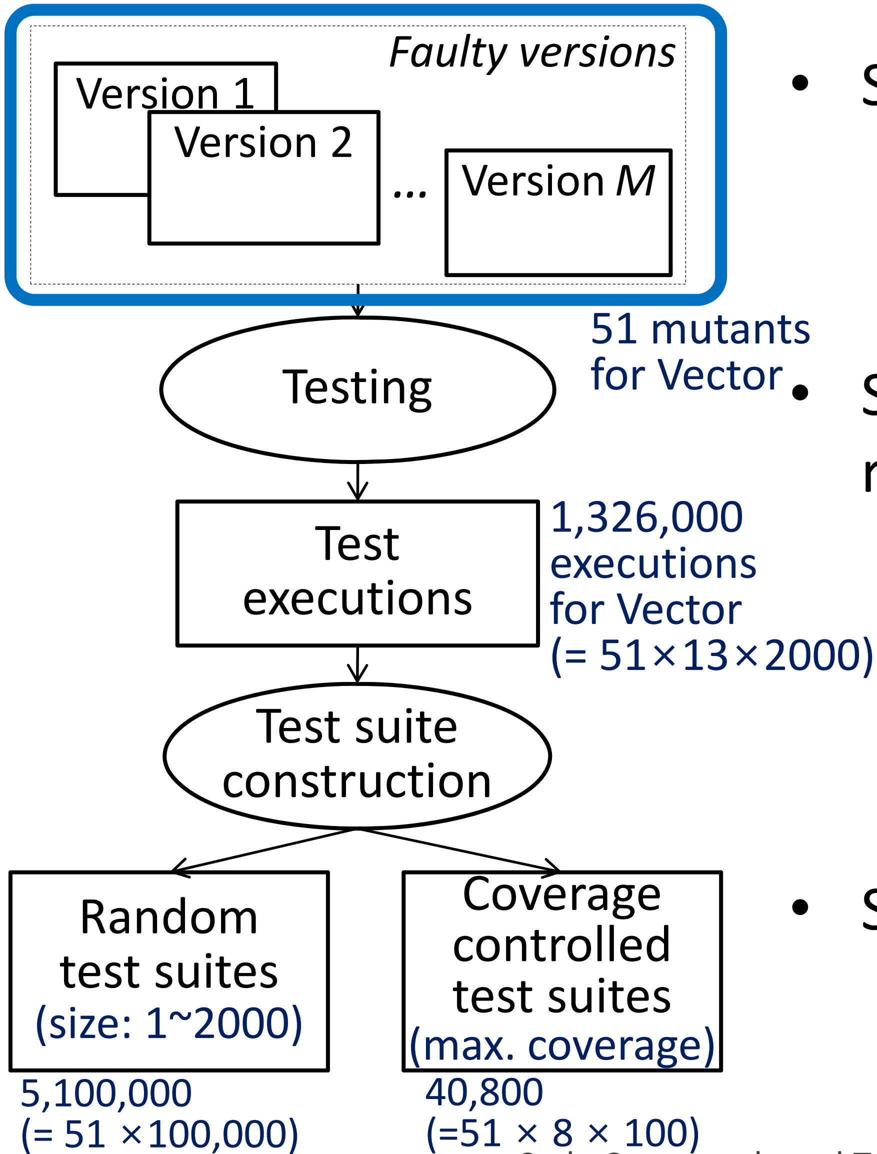
- Single fault programs
  - 6 programs in concurrency bug bench. [Neha et al., PADTAD 09]
  - Each program has a fault with low error density [Dwyer et al., FSE 06]
- Mutation testing
  - Generate 34~88 incorrect versions (valid mutants) for each program
    - Used **concurrent mutation operators** [Bradbury et al., MUTATION 06]
    - Each version is created by applying one mutation operator once

Experiment Process - Single Fault Programs



- Step 1. Generate test executions
  - Use 13 random testing configurations
  - Generate 1000 executions per testing configuration
- Step 2. Construct test suites by resampling test executions
  - Generate 100,000 random test suites of sizes 1 – 1000
  - Generate 100 test suite controlled to achieve maximum overage per metric
- Step 3. Measure metrics for test suites
  - Measure 8 coverage metrics
  - Measure fault finding

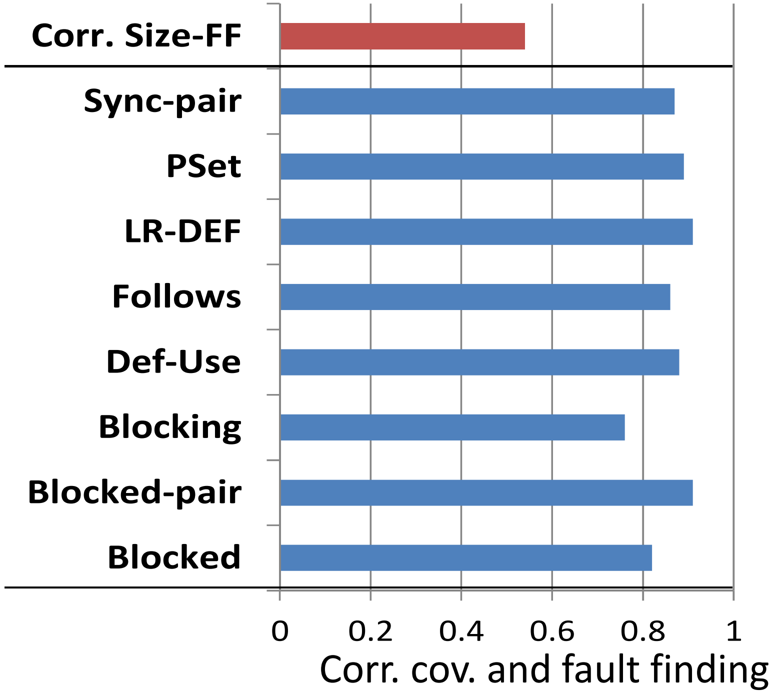
Experiment Process – Mutation Testing

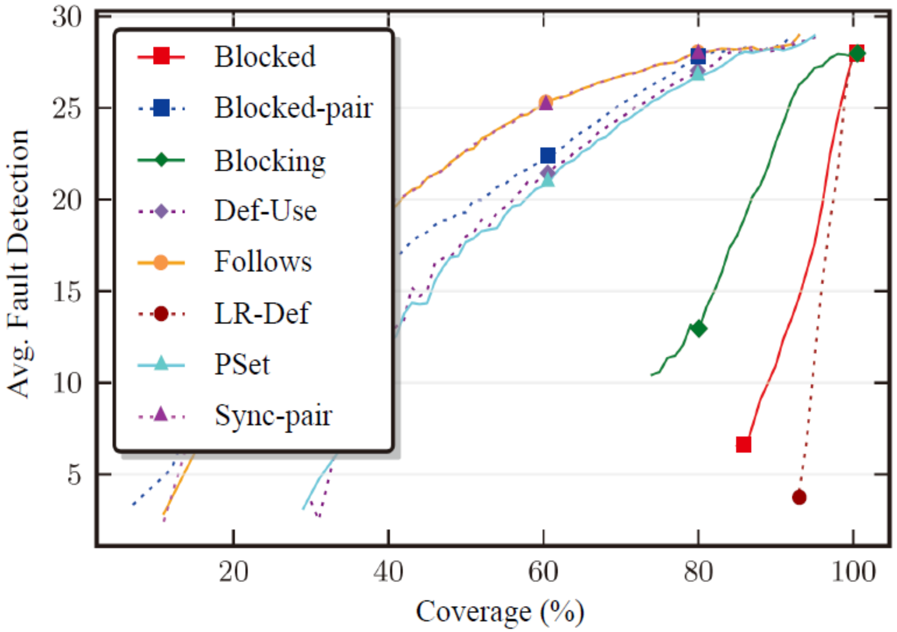


- Step 1. Generate test executions
  - Use 13 random testing configurations
  - Generate 2000 executions per mutant and per testing configuration
- Step 2. Construct test suites by resampling test executions
  - Generate 100,000 random test suites of sizes 1 – 2000 per mutant
  - Generate 100 test suites controlled to achieve maximum coverage per mutant and per coverage metric
- Step 3. Measure metrics for test suites
  - Measure 8 coverage metrics
  - Measure fault finding (mutation score)

RQ 1: Does Coverage Achieved Impact Fault Finding ?

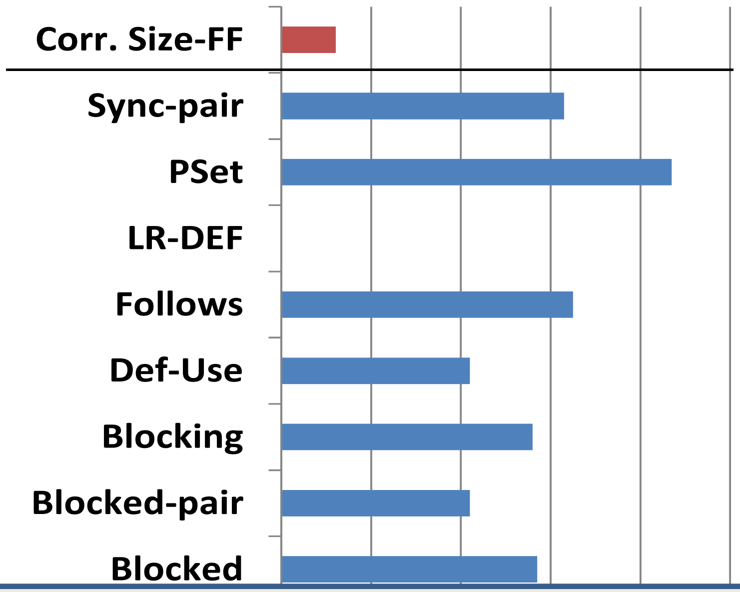
- Compute the **correlations of coverage metrics and fault finding** as well as the **correlations of test suite size and fault finding** by Pearson's *r*
- Results of mutation testing subjects
  - Coverage metrics have stronger correlations than test suite size
  - Ex. *Vector*

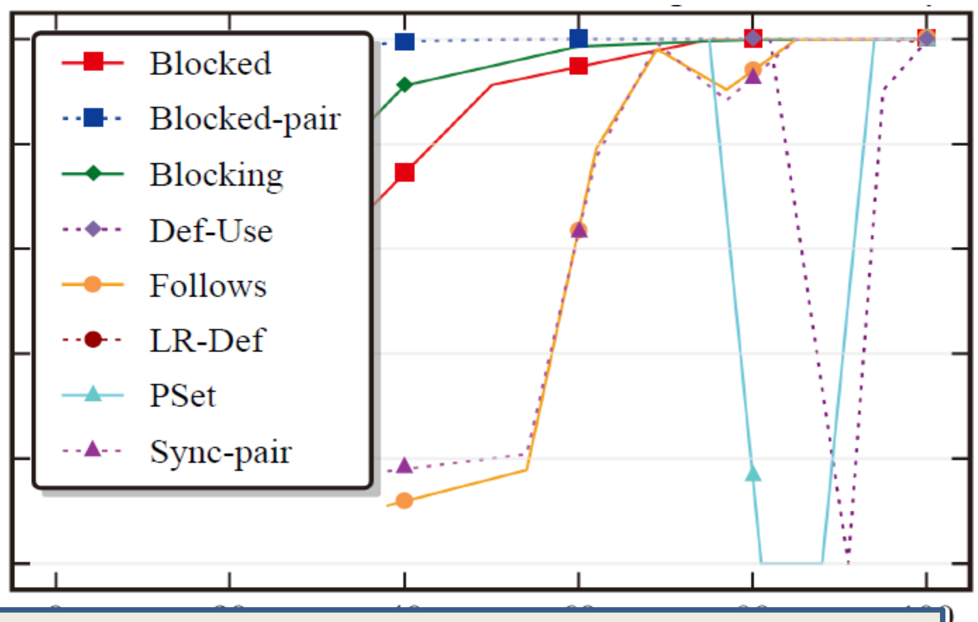




RQ 1: Does Coverage Achieved Impact Fault Finding ?

- Results of single fault subjects
  - There is a coverage metric having high correlation for each subject
  - Ex. *Stingbuffer*





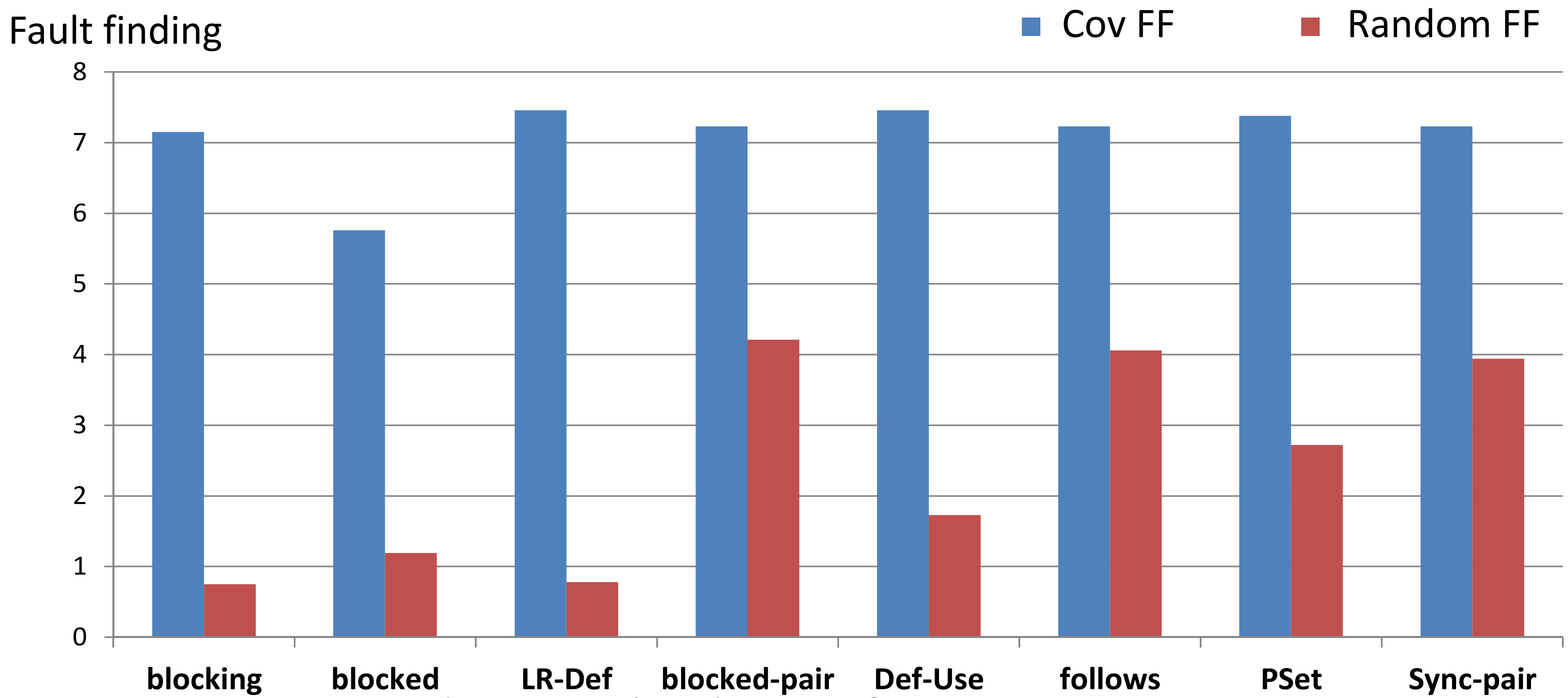
RQ 1: Is concurrent coverage good predictor of test. effectiveness?

➔ **Yes**. The metrics estimate fault finding of a testing properly

RQ 2: Does Coverage Controlled Testing Detect More Faults?

- Compare fault finding of a coverage-controlled test suite w.r.t. a metric *M* and fault finding of random test suite of equal size
- Results of mutation testing
  - Ex. *ArrayList*

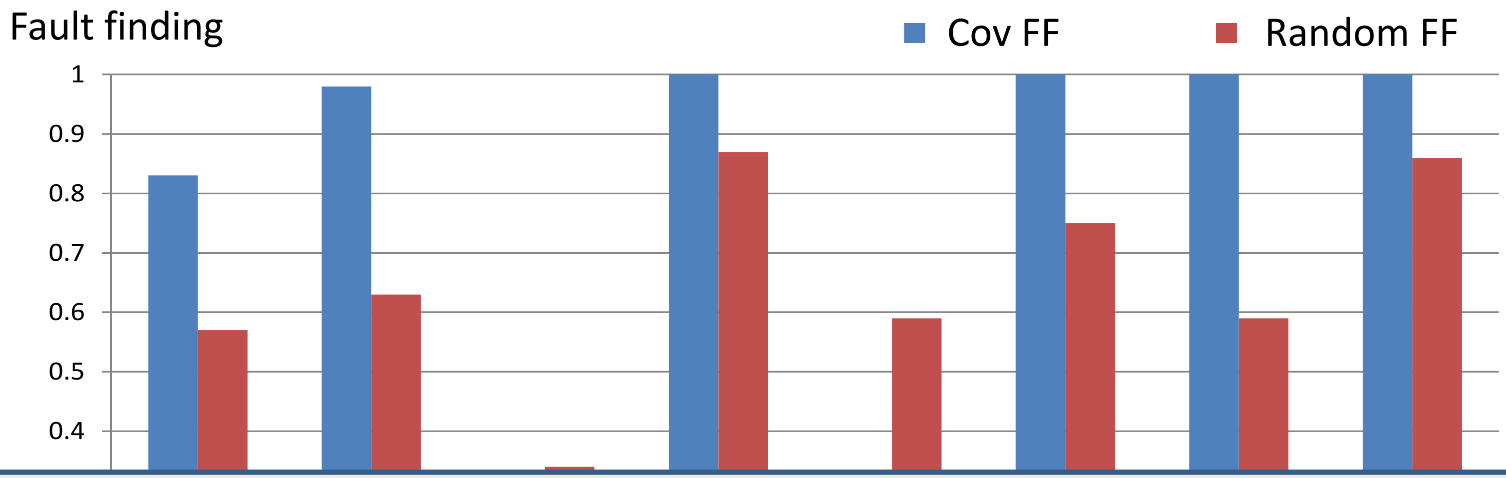
* Cov FF / Random FF: fault finding of controlled test suites/random test suite (0~8.5)



RQ 2: Does Coverage Controlled Testing Detect More Faults?

- Results with single fault programs
  - Generally, controlled test suites have higher fault finding abilities than random ones
  - Coverage metrics have different performances depending on programs
  - Ex. *Stringbuffer*

* Cov. FF / Random FF: fault finding of coverage controlled test suites /random test suite (0~1)



RQ 2: Is concurrent coverage proper for test generation ?
→ **Yes**. Generating test suites toward high coverage can detect more faults than random test generation

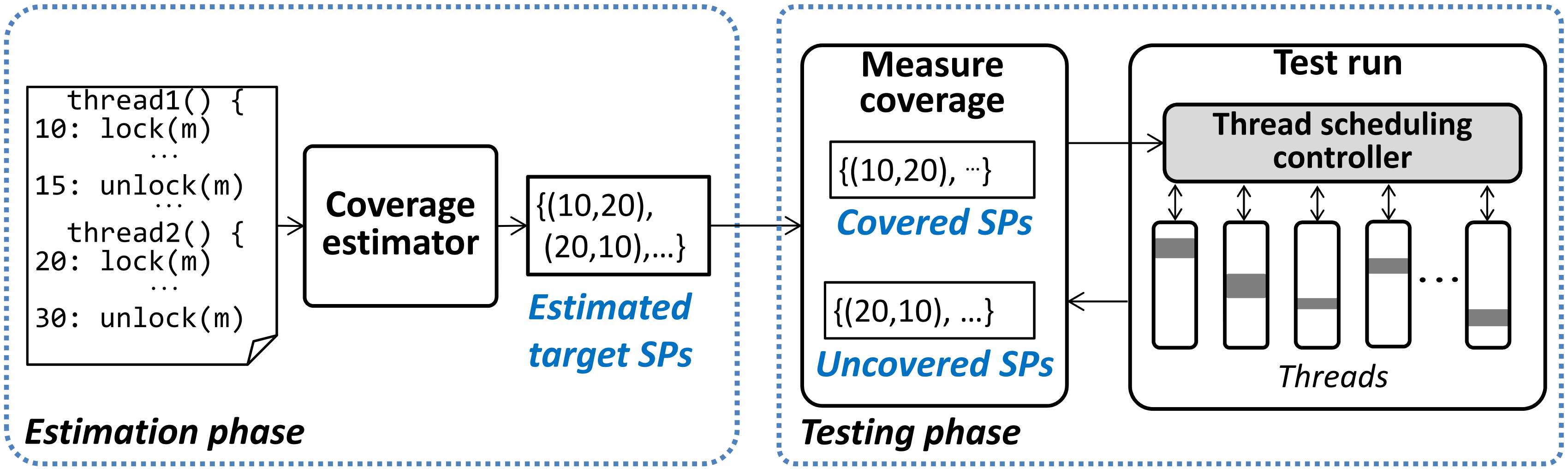
Lessons Learned: Concurrent Coverage is Good Metric

1. Use **concurrent coverage metrics** to improve testing!
   - Good predictor of testing effectiveness
   - Good target for test generation
2. ***PSet*** is the best pairwise coverage metric used alone
   - High correlation with fault finding in general
   - High fault finding for controlled test suites w.r.t. PSet in all subjects
3. ***PSet + follows*** would be better than just a metric alone
   - For some objects, there is a large difference in fault finding depending on metrics
   - A combined metric of data-access based and synchronization-based coverage would provide reliable performance in general

Part II.
Testing Concurrent Programs to Achieve High Synchronization Coverage

Overview

- A testing framework for concurrent programs
  - To achieve **high test coverage fast**
- Key idea
  1. Utilize **coverage** to test concurrent programs systematically
  2. Manipulate **thread scheduler** to achieve high coverage fast



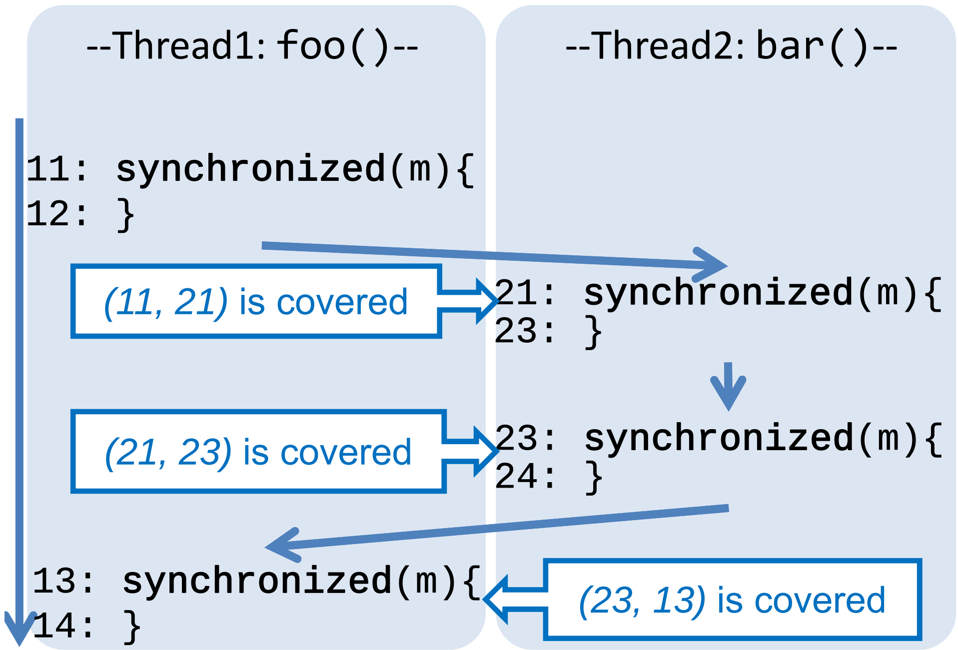
Synchronization-Pair (**SP**) Coverage

```
10:foo() {
11: synchronized(m){
12: }
13: synchronized(m){
14: }
15:}
```

```
20:bar() {
lock(m) ronized(m){
22: }
23: synchronized(m){
24: }
25:}
```

lock(m)

unlock(m)



Def. A pair of code locations **$< l_1, l_2 >$**

is a **SP coverage requirement**, if

(1) l_1 and l_2 are lock statements

(2) l_1 and l_2 hold the same lock m

(3) l_2 holds m right after l_1 releases m

Covered SPs:
(11, 21), (21, 23), (23, 13)

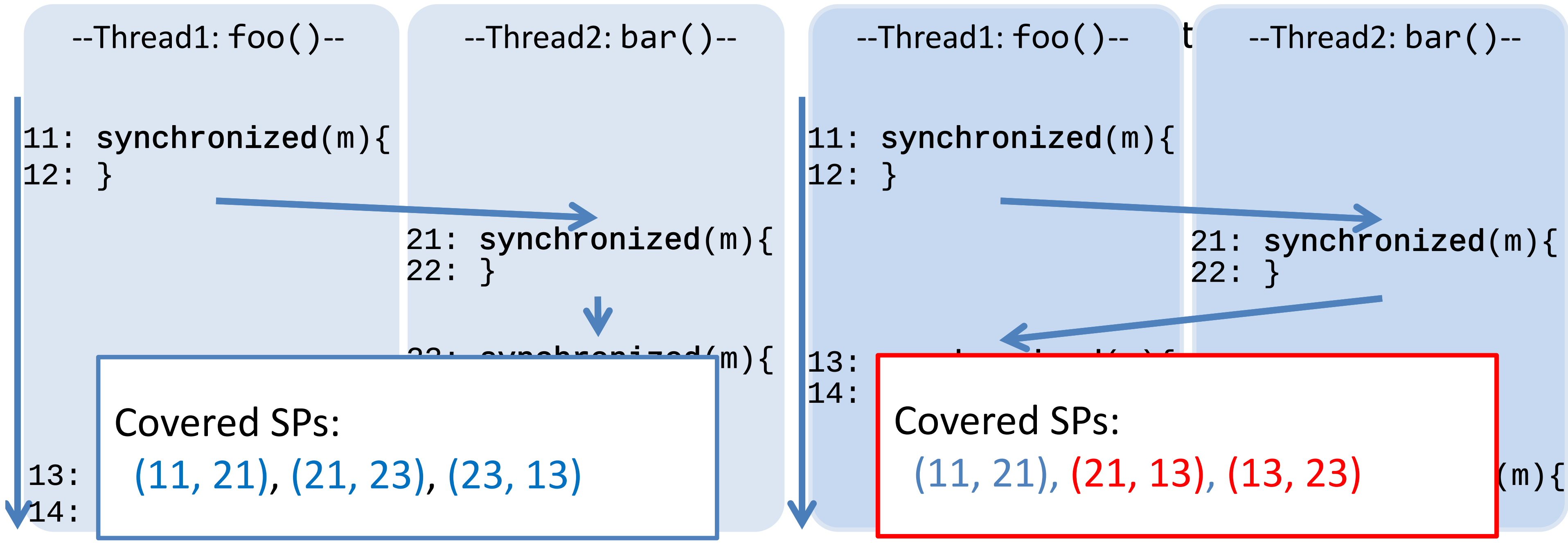
Synchronization-Pair (**SP**) Coverage

```
10:foo() {                 20:bar() {
11:  synchronized(m){      21:  synchronized(m){
12:  }                     22:  }
13:  synchronized(m){      23:  synchronized(m){
14:  }                     24:  }
15:}                       25:}
```

Def. A pair of code locations $<l_1, l_2>$ is a **SP coverage requirement**, if

(1) l_1 and l_2 are lock statements

(2) l_1 and l_2 hold the same lock m

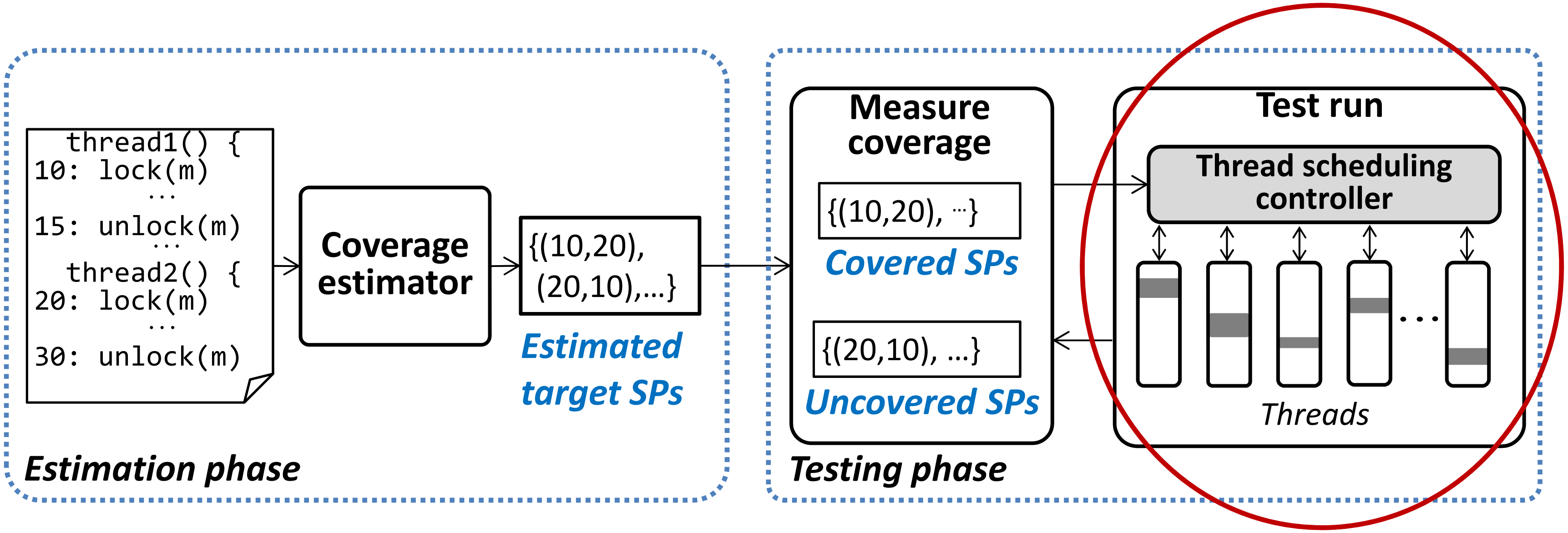


Testing Framework for Concurrent Programs

(1) Estimates SP requirements,

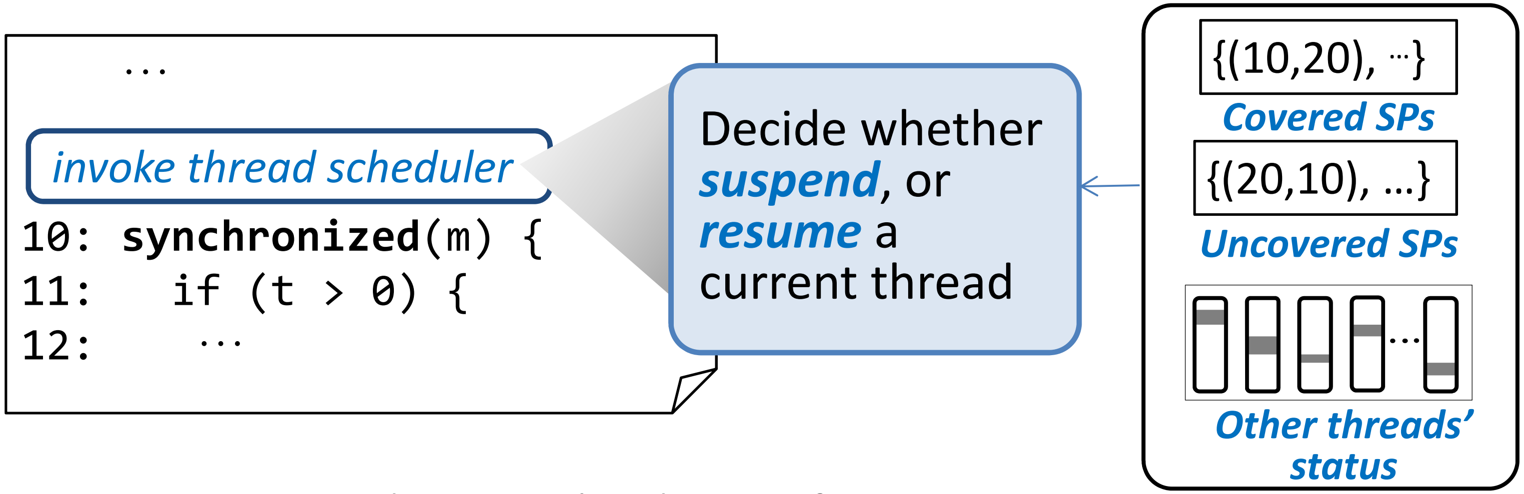
(2) Generates test scheduling by

- monitor running thread status, and measure SP coverage
- suspend/resume threads to cover new coverage req.



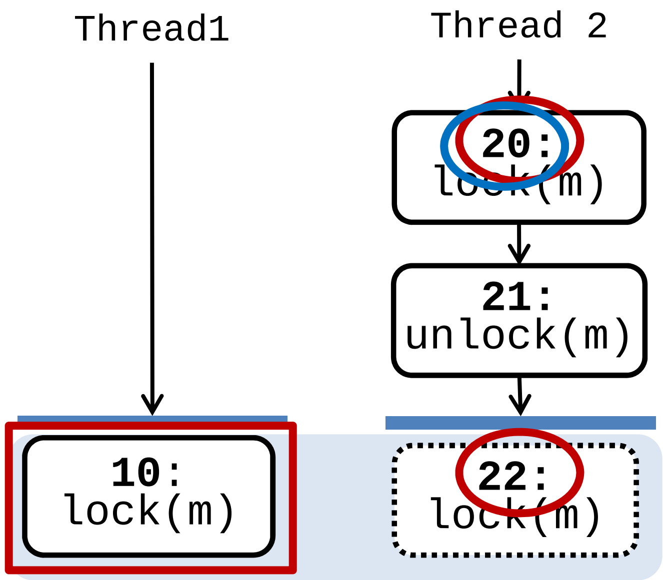
Thread Scheduling Controller

- Coordinates thread executions to satisfy new SP requirements
- Invokes an operation
  (1) **before** every lock operation, and
  (2) **after** every unlock operation
- Controls thread scheduling by
  (1) suspend a thread before a lock operation
  (2) select one of suspended threads to resume using **three rules**



Thread Schedule Decision Algorithm (1/3)

- Rule 1: Choose a thread to cover uncovered SP **directly**



- Covered SPs:

(20,22), (20,10)

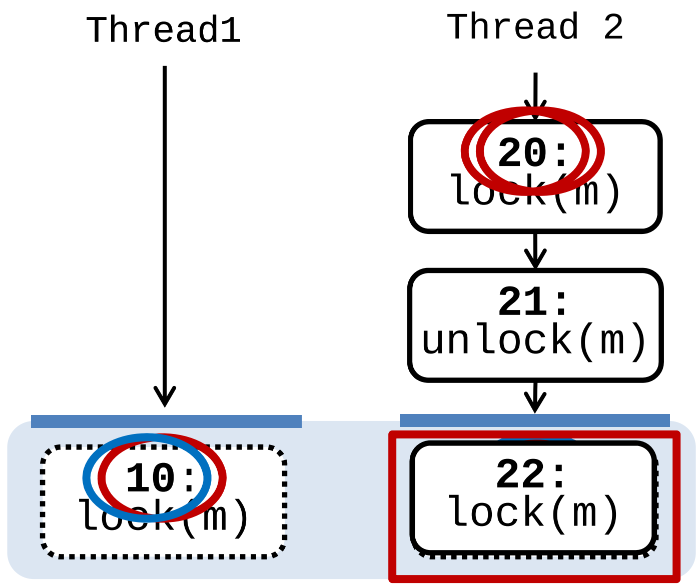
- Uncovered SPs:

(10,22),(10,22),

~~(20,10)~~

Thread Schedule Decision Algorithm (2/3)

- Rule 2: Choose a thread to cover uncovered SP **in next decision**

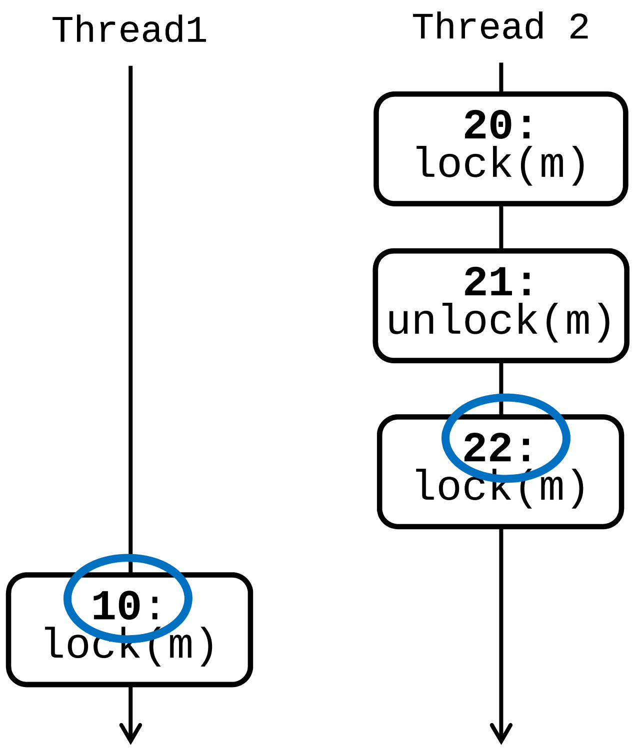


- Covered SPs:
(20,10), (20,22), (10,22)

- Uncovered SPs:
(22,10)

Thread Schedule Decision Algorithm (2/3)

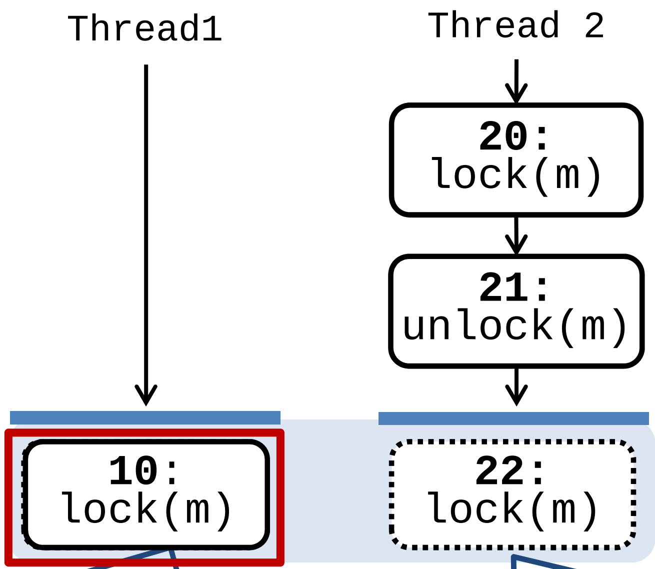
- Rule 2: Choose a thread to cover uncovered SP **in next decision**



- *Covered SPs*:
(20,10),(20,22), (10,22),(22,10)
- *Uncovered SPs*:
~~(22,10)~~

Thread Schedule Decision Algorithm (3/3)

- Rule 3: Choose a thread that is **unlikely to cover uncovered SPs**



- ***Covered SPs:***
  (20,10),(10,20),(20,22),(10,22),(22,10)
- ***Uncovered SPs:***
  (10,60),(70,22),(80,22),(10,50),(22,60)

schedule Thread 1:
Since Thread 2 with line **22** remains under control, more chance to cover (70, **22**), (80, **22**), (**22**, 60)

schedule Thread 2:
Since Thread1 with line **10** remains under control, more chance to (**10**, 60), (**10**, 50)

Empirical Evaluation

- Implementation [Thread Scheduling Algorithm, **TSA**]
  - Used Soot for estimation phase
  - Extended CalFuzzer 1.0 for testing phase
  - Built in Java (about 2KLOC)

- Subjects
  - 7 Java library benchmarks (e.g. Vector, HashTable, etc.) (< 11 KLOC)
  - 3 Java server programs (cache4j, pool, VFS) (< 23 KLOC)

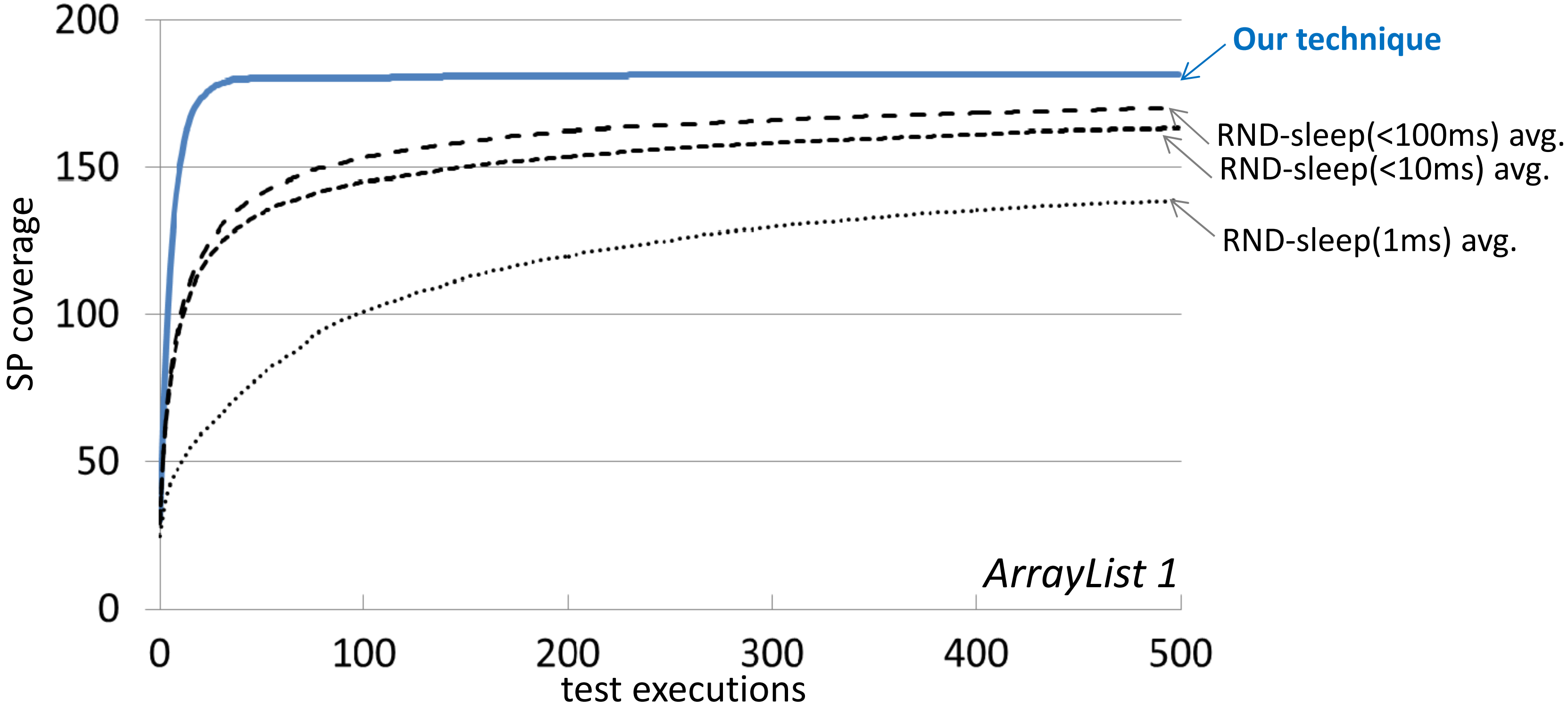
Empirical Evaluation

- Compared techniques
  - We compared TSA to random testing
  - We inserted probes at every read, write, and lock operations
  - Each probe makes a time-delay d with probability p
    - d: sleep(1ms), sleep(1~10ms), sleep (1~100ms)
    - p : 0.1, 0.2, 0.3, 0.4,0.5
  - We use 15 (= 3 x 5) different versions of random testing

- Experiment setup
  - Executed the program 500 times for each technique
  - Measured accumulated coverage and time cost
  - Repeated the experiment 30 times for statistical significance in results

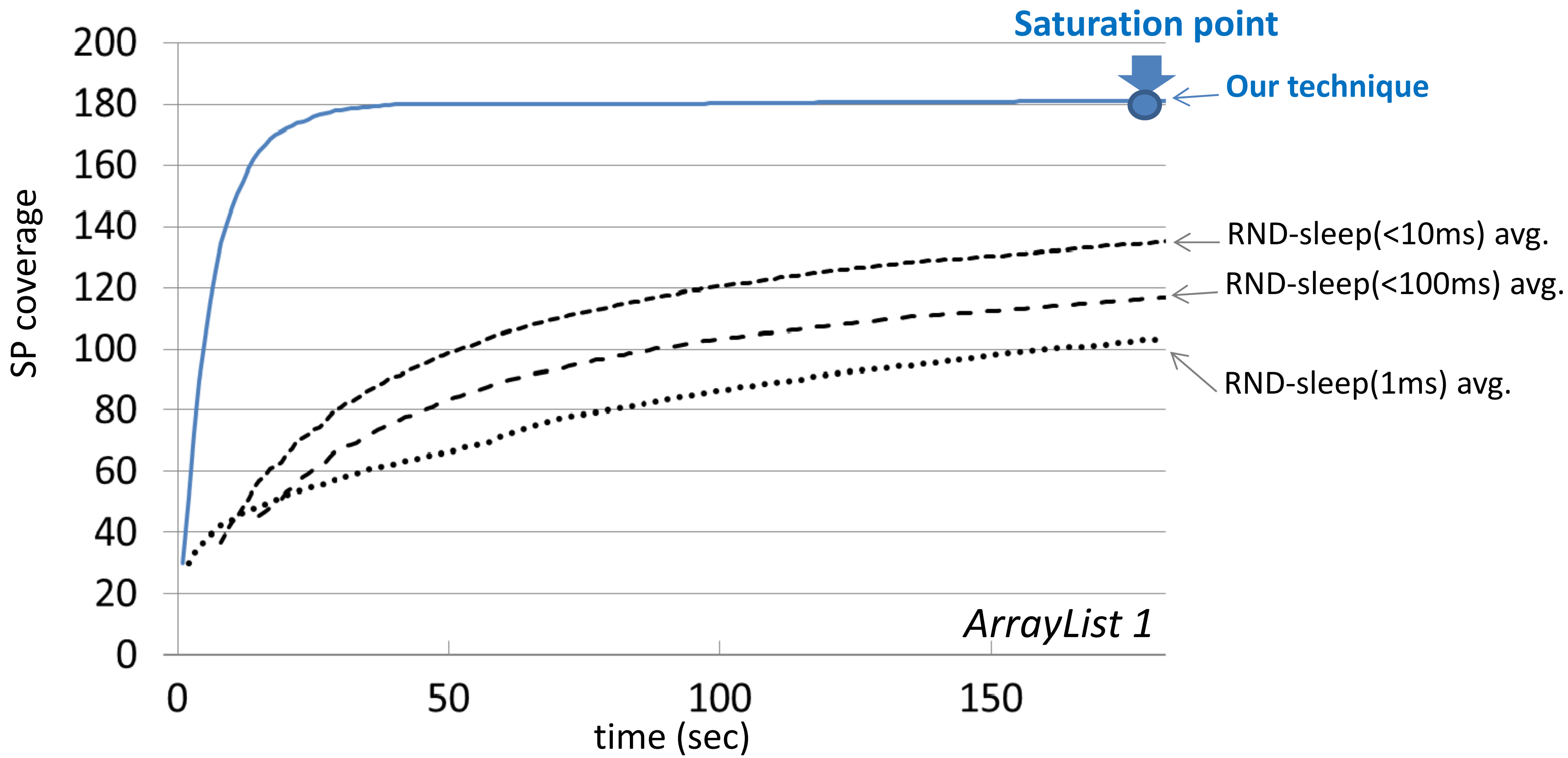
Study 1: Effectiveness

- TSA covers **more SPs** than random testings
  - for accumulated SP coverage after 500 executions



Study 2: Efficiency

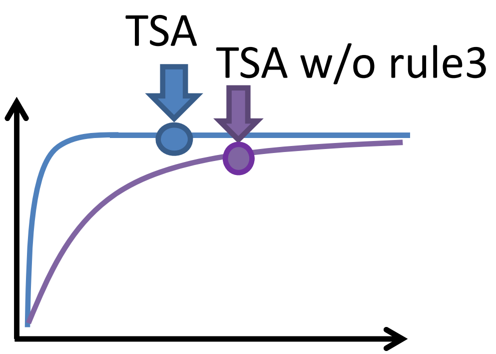
- TSA reaches the saturation point **faster** and **higher**
  - A saturation point is computed by r^2
    (coefficient: 0.1, window size: 120 sec.) [Sherman et al., FSE 2009]



Study 3: Impact of Estimation-based Heuristics (Rule3)

- TSA with Rule3 reaches **higher** coverage at **faster** saturation point
  - Executes the program for 30 minutes, and computed the saturation points
- > 90% of thread scheduling decisions are made by the Rule 3

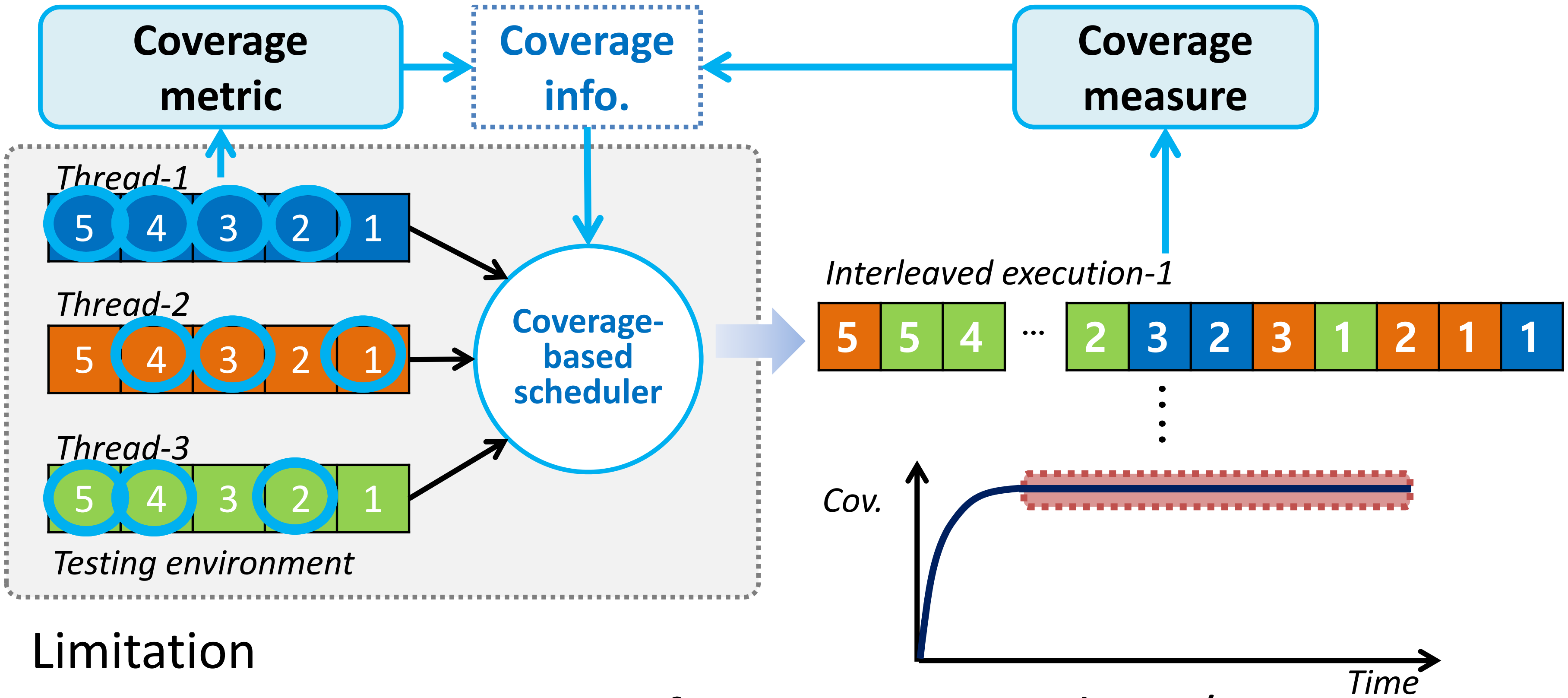
Program	TSA **w/o Rule 3**		TSA **with Rule 3**	
	Coverage	time (sec)	Coverage	time (sec)
ArrayList1	**177.6**	**274.4**	**181.2**	**184.2**
ArrayList2	130.8	246.3	141.4	159.7
HashSet1	151.3	271.5	151.7	172.4
HashSet2	98.0	198.9	120.8	139.3
HashTable1	23.7	120.0	24.0	120.0
HashTable2	539.6	388.8	538.0	165.4
LinkedList1	179.9	278.2	181.2	155.0
LinkedList2	129.9	237.7	141.2	161.2
TreeSet1	151.6	258.4	151.4	191.2
TreeSet2	98.8	237.5	120.5	139.8
cache4j	201.9	205.8	202.2	146.1
pool	T/O	T/O	2950.5	431.1
VFS	246.7	478.2	260.1	493.9



CUVE: Effective and Efficient Testing of Concurrent Programs using Combinatorial Concurrency Coverage

Shin Hong, Yongbase Park, Moonzoo Kim
Software Testing & Verification Group
KAIST

Limitation of Coverage-based Testing Technique

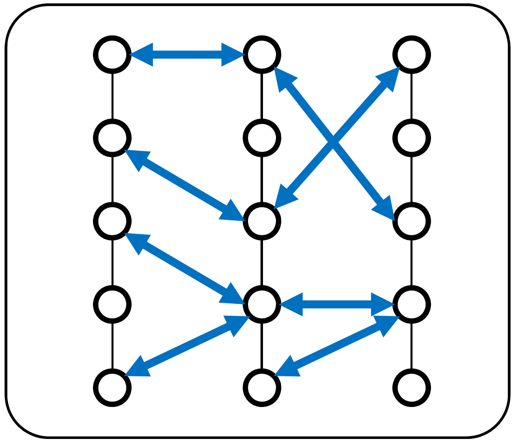


Limitation

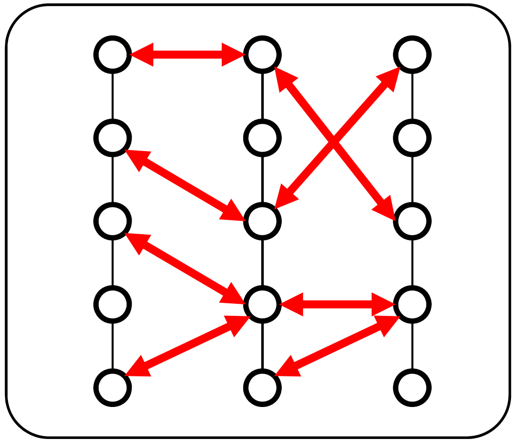
- No coverage metric is perfect as testing predictor/target [Hong et al. ICST 13, Hong et al. STVR]
- Not effective to generate diverse behaviors once a test reaches likely coverage saturation

Overview

- We present *CUVE*, a coverage-based multithreaded program testing technique that uses a new coverage *combinatorial concurrent coverage*
- The experiment results show that CUVE can detect **more concurrency faults** while consuming **less testing time** than conventional techniques



Conventional coverage



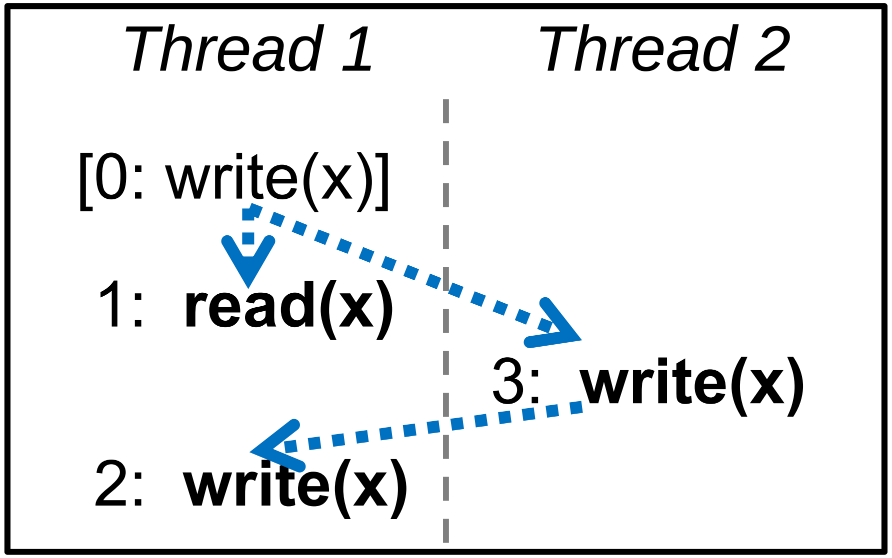
Combinatorial coverage

Combinatorial Concurrent Coverage

- Idea: **the combination of two test requirements** of a metric *M* can capture **more diverse interleavings** than the test requirements by *M*
- A combined test requirement is a combination of two test requirements
  - $CovMetric(ProgCode) = TestReq = \{r_1, r_2, \ldots, r_n\}$
  - $CombConcCov(TestReq) = \{\{r_1, r_2\}, \{r_1, r_3\}, \ldots, \{r_{n-1}, r_n\}\}$
    - An execution covers a combinatorial test requirement $\{r_1, r_2\}$ when the execution covers both r_1 and r_2
  - For n singular requirements, we obtain $C(n,2) = \frac{n \times (n-1)}{2}$ as combined requirements

Why Combinatorial Coverage? (1/3)

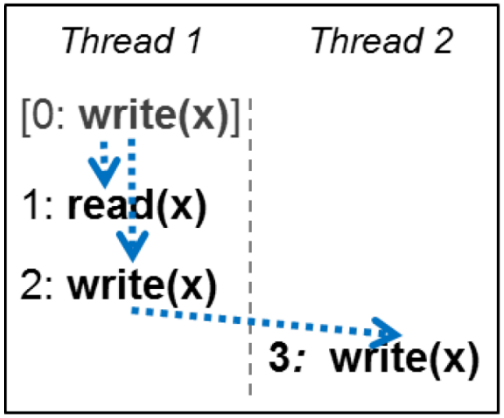
- For detecting atomicity violation errors



Atomicity violation inducing interleaving

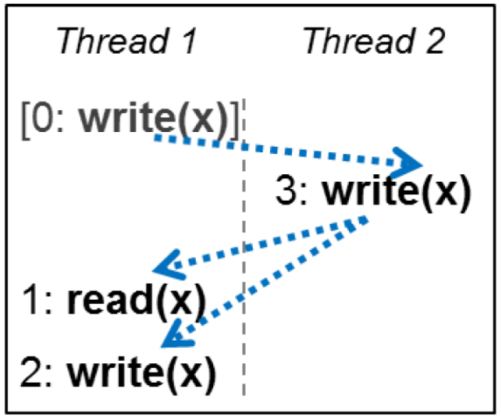
Def-Use TRs**: (0, 1), (0, 3), (3, 2)**

Comb. TRs: **{(0,1), (3, 2)}**



Non-problematic inteleaving-1

Def-Use TRs:
(0, 1), (0, 2), (2, 3)



Non-problematic interleaving-2

Def-Use TRs:
(0, 3), (3, 1), **(3, 2)**

Why Combinatorial Coverage? (2/3)

- Detecting general race error

```
arr[0..1] // array of size 2
len = 2 ;
p = 0 ;

Thread1():
  lock(m) ;
  if (p < len)
    arr[p] = x;
  if (p+1 < len)
    p++ ;
  unlock(m);

Thread2():
  lock(m);
  if (p < len)
    arr[p++] = y;
  unlock(m);

Thread3():
  lock(m);
  z = arr[p];
  if (p > 0)
    p--;
  unlock(m);
```

- No data race detected and no atomicity violation detected
- A test can achieve maximum Sync-Pair coverage without any fault detection

Why Combinatorial Coverage? (3/3)

```
arr[0..1] // array of size 2
len = 2 ;
p = 0 ;
```

```
Thread1():
1   lock(m) ;
2   if (p < len)
3     arr[p] = x;
4   if (p+1 < len)
5     p++;
6   unlock(m);
```

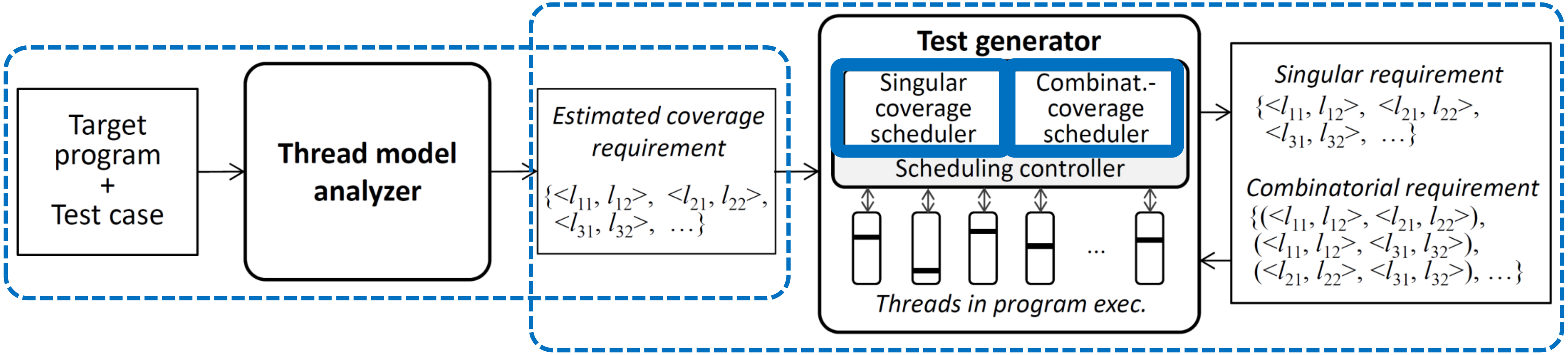
```
Thread2():
11   lock(m);
12   if (p < len)
13     arr[p++] = y;
14   unlock(m);
```

```
Thread3():
21   lock(m);
22   z = arr[p];
```

Array out of index bound

➔ A combined requirement {(1, 11), (11, 21)} determines this fault detecting execution

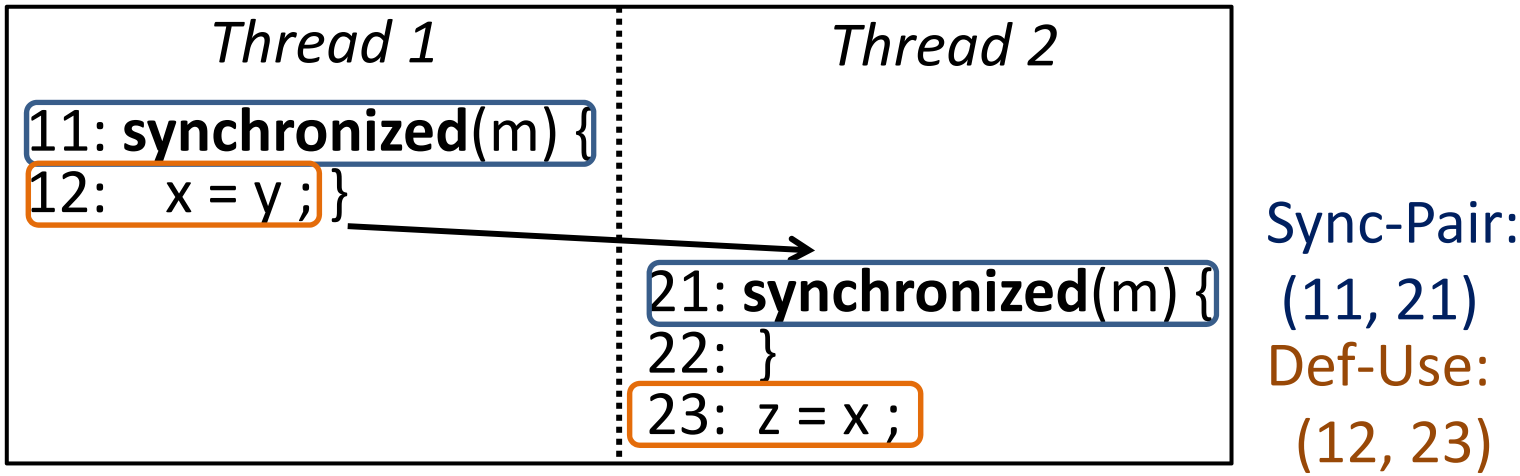
CUVE Framework



- Three testing phases
  (1) Coverage estimation phase ;
  (2) Singular coverage-based testing phase ;
  (3) Combinatorial coverage-based testing phase

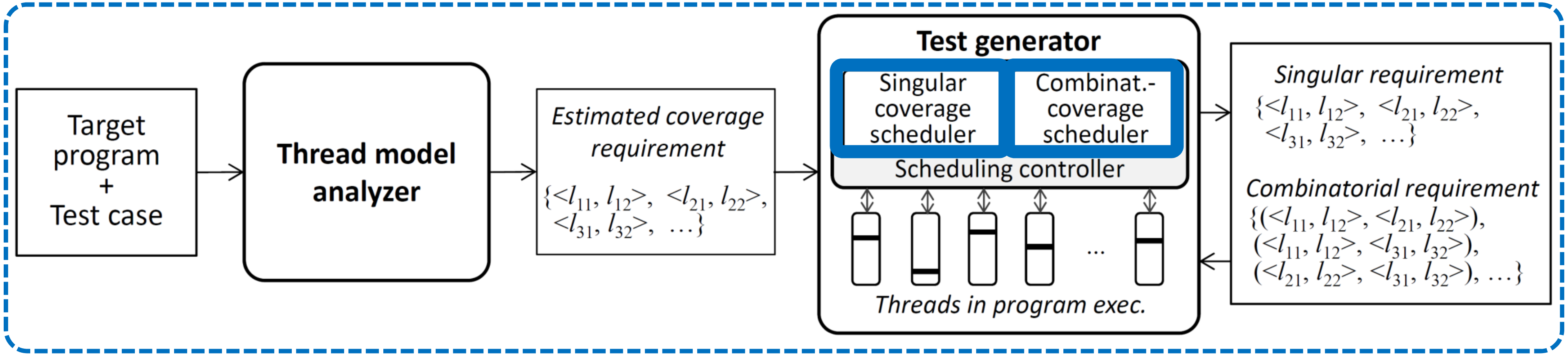
Base Concurrent Coverage Metric

- Use Sync-Pair + Def-Use as a base singular metric
  - Integrate two metrics defining test requirements for different code constructs
    - Sync-Pair: check a consecutive lock contention for two locking
    - Def-Use: check a shared variable writing and its immediate reading
  - Each metric is known to have a high correlation with fault detection



- Generate a combined test requirement for every two singular test requirements (for example, {(11, 21), (12, 23)})

CUVE Framework



- Three testing phases

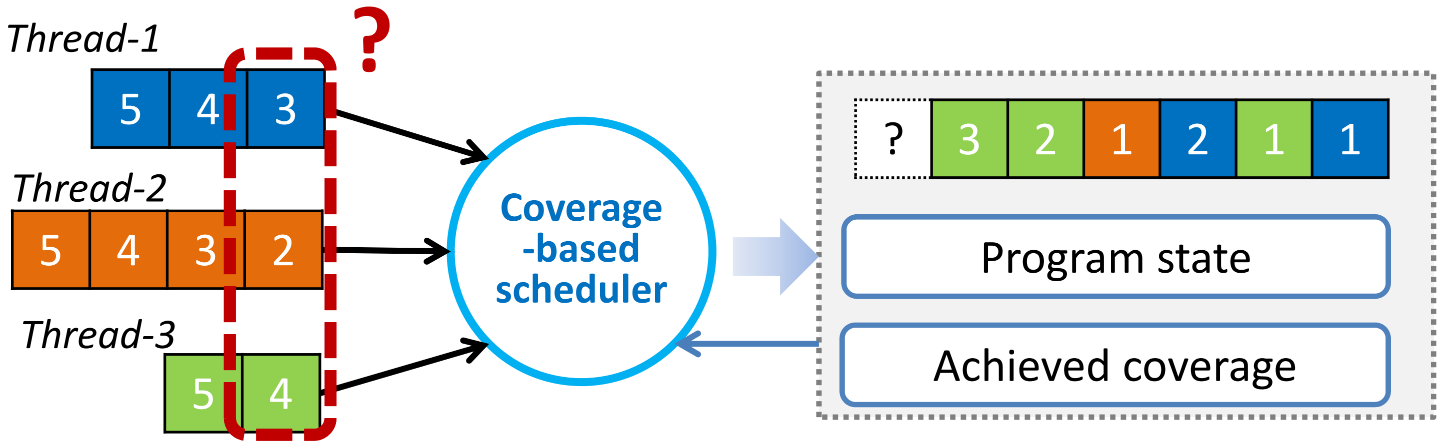
(1) Coverage estimation phase ;

(2) Singular coverage-based testing phase ;

[Hong et al., ISSTA 12]

(3) Combinatorial coverage-based testing phase

Thread Scheduling Algorithm: Greedy Card Player Heuristic



- Rule 1: Choose a thread that **directly covers a largest number of uncovered combined test requirements**
- Rule 2: Choose a thread that covers **a largest number of uncovere d combined requirements in next decision**
- Rule 3: Choose a thread **expected to cover a smallest number of uncovered combined requirements in later step** of this execution

Experiment

- To know
  - Does CUVE detect more diverse faults than the conventional techniques?
  - Does CUVE consume less time to detect faults than the conventional techniques?
  - Does CUVE detect higher coverage than the conventional techniques?
- By comparing CUVE with
  - **RN**: 12 noise injection-based random testing
  - **RS**: randomized scheduler
  - **JPF**: Java Pathfinder (systematic testing)
  - **CUVE-c**: Singular coverage-based testing technique
  - RaceFuzzer (bug-directed testing technique)

Study Object and Mutant Generation

- Generate multiple faulty versions (mutants) by making single syntactic change (mutator) systematically
- Use expression mutation operators and synchronization op
- Th

E.g. RSB: remove a synchronized block

```
synch(a){
   synch(b){
```

```
synch(a){
   synch(b){
```

```
synch(a){
   synch(b){
```

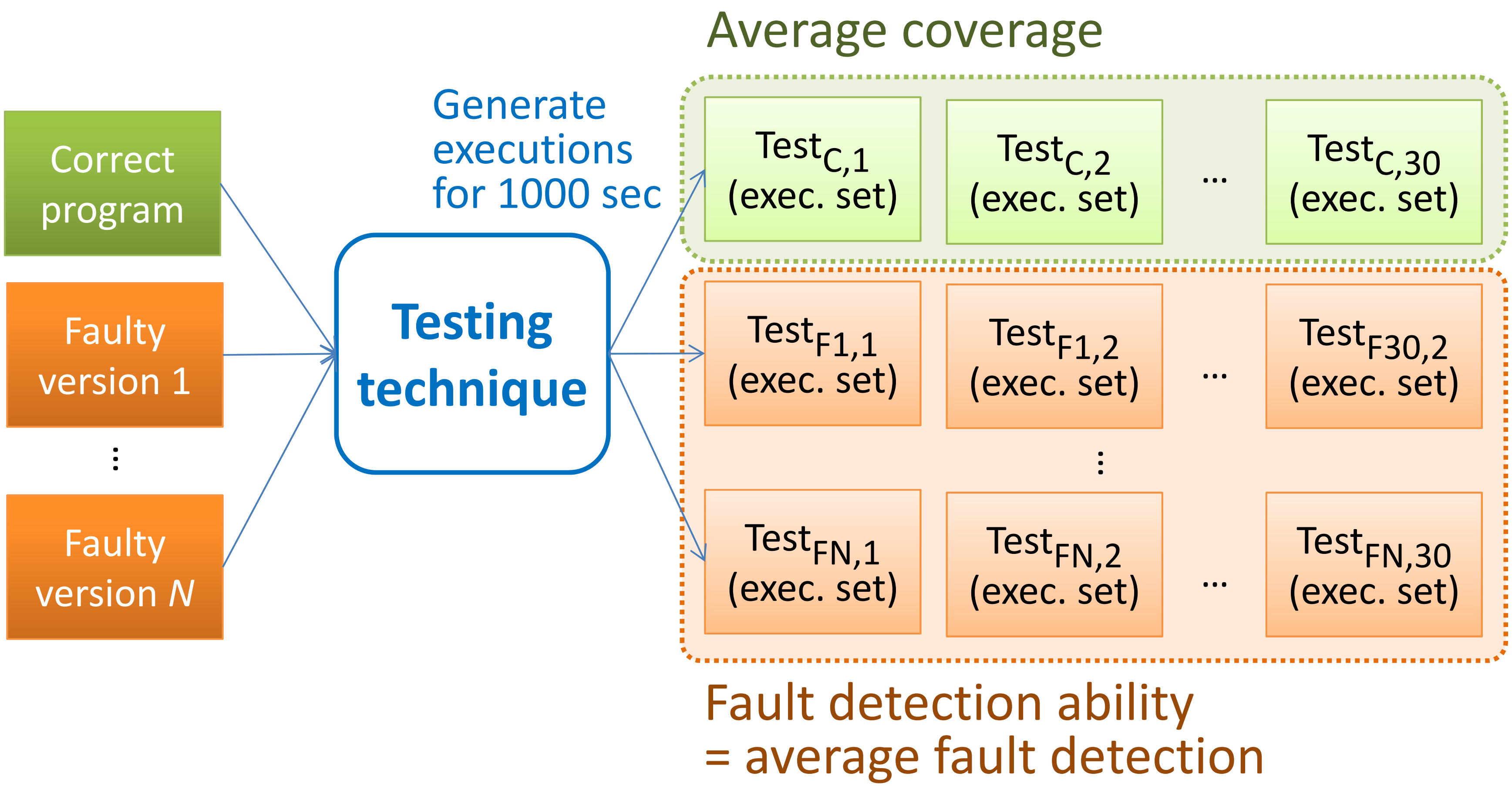
```
synch(a){
   synch(b){
   }
}
```

`(x < y)` ➔ `(x <= y)`

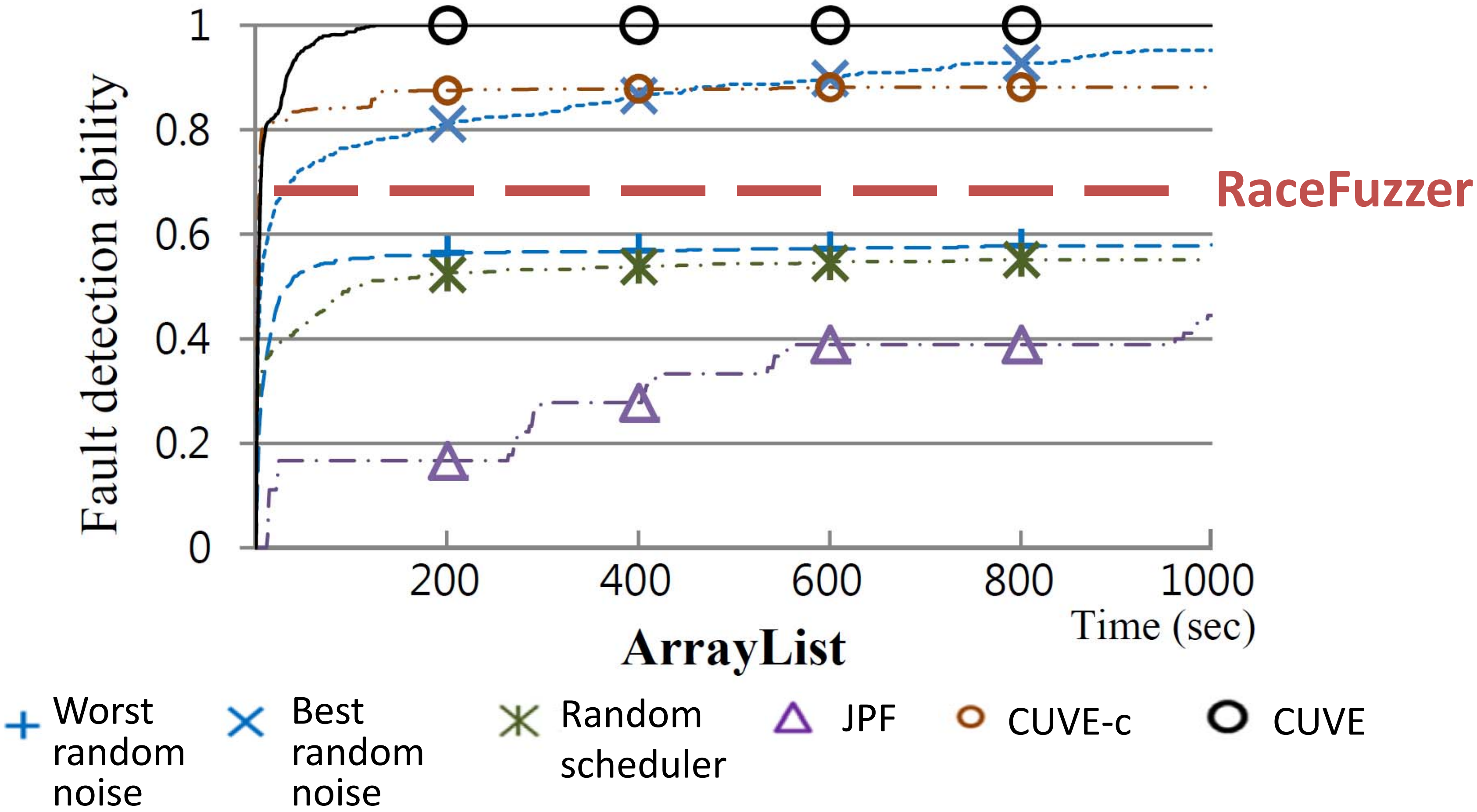
Category	Mutation operator description
Expres mutat opera	Access flag change
Synch nizati mutation operator	Shrink synchronized block
	Split synchronized block

Progarm	Size (LOC)	Number of threads	Number of mutants
ArrayList	3090	27	18 (201)
HashMap	3941	27	12 (300)
TreeSet	4049	22	35 (251)

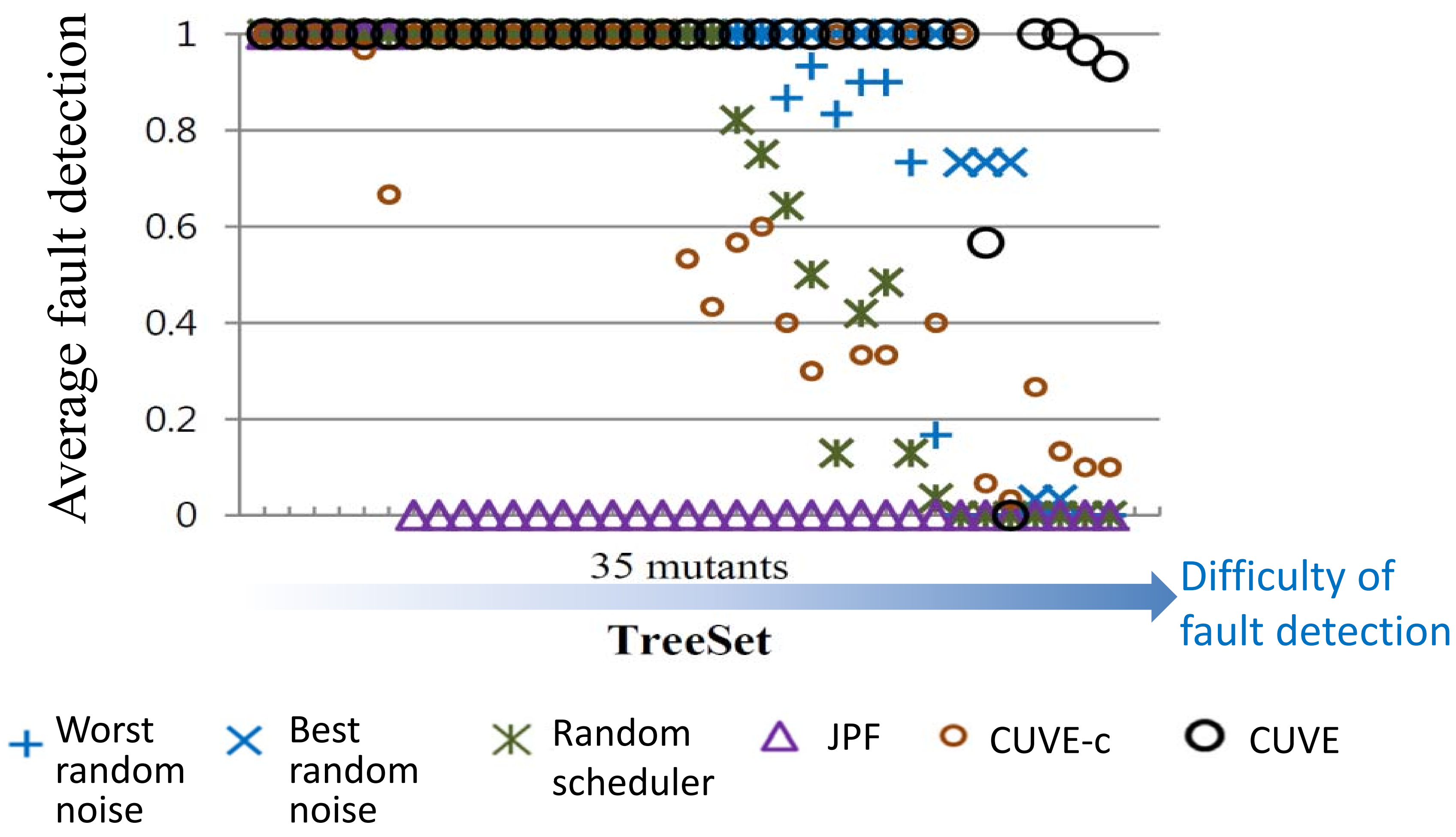
Test Generation



Fault Detection Ability Result

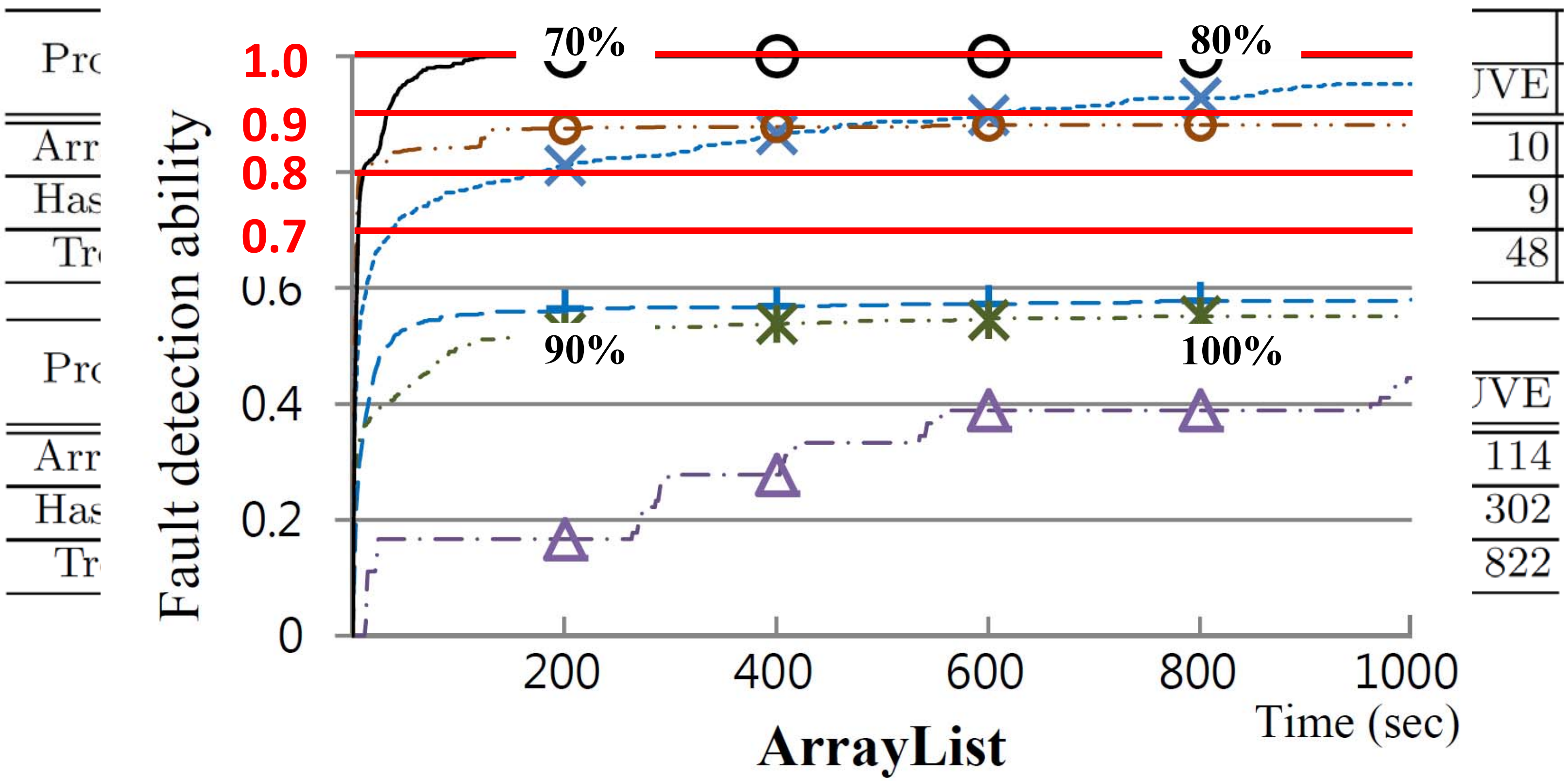


Fault Detection Per Mutant



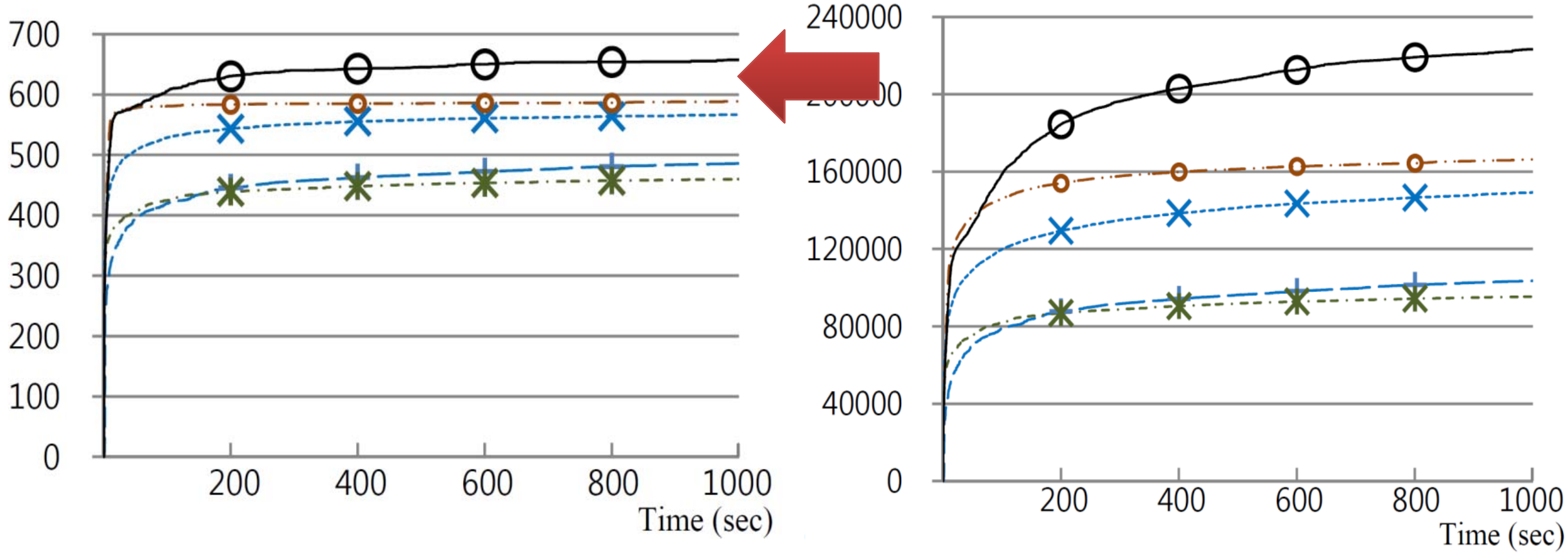
Fault Detection Efficiency

- Time to reach certain level of fault detection ability



Coverage Achievement Result

Singular coverage **Combinatorial coverage**

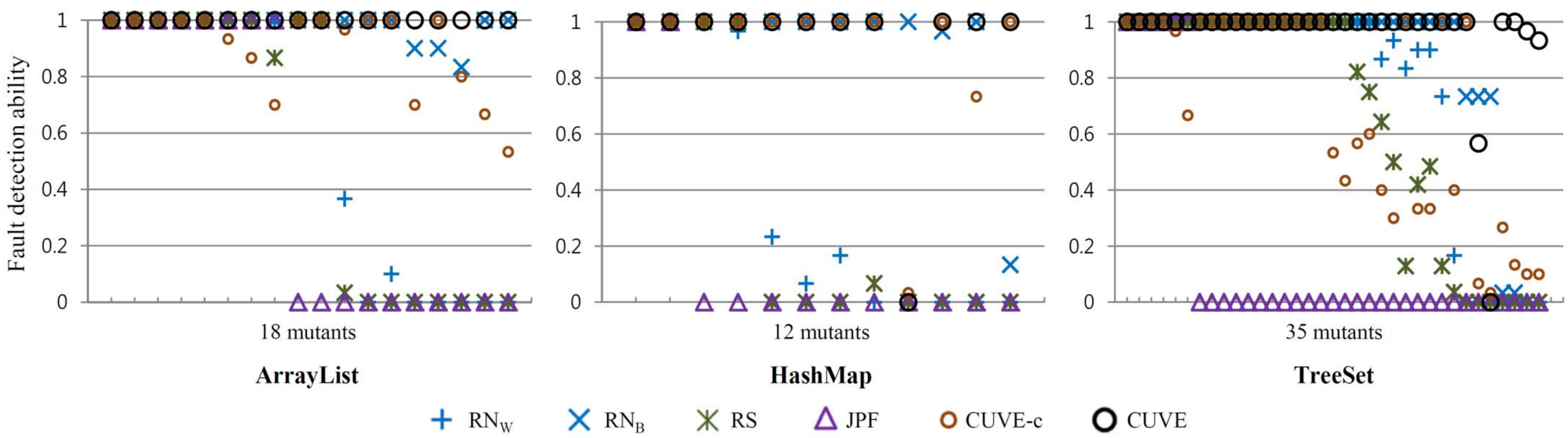


TreeSet

+ Worst random noise
× Best random noise
✳ Random scheduler
o CUVE-c
O CUVE

Mutant Generation Result

- Expression mutation operators generate useful faulty versions that contain concurrency faults
- Generated mutants have diverse difficulties of detecting faults



Summary

- We propose combinatorial concurrent coverage as a useful multithreaded program testing metric
- CUVE generate thread schedules achieving high combinatorial concurrent coverage
- Through the mutation testing, we show that CUVE provides effective and fast conc. fault detections

Future Work

- Use only a core subset of test requirements for generating test generation targets
  - How to create a core test requirement subset?
  - Can we guarantee that such technique can provide safe testing results?
- Use multiple test input values, instead of one
  - What is a 'good' set of test input values?
  - In which order a testing should use test input values?
  - How can we utilize coverage metrics in this case?