코드 커버리지를 이용한 동시성 프로그램 테스트 자동 생성

홍 신 안재민 Matt Staats 김문주

PROVABLE SW LAB

KAIST

Motivation and Overall Research Goal

- *Concurrent programming* becomes popular! So does *concurrency bug*!
  - **37 %** of all open-source C# applications and **87%** of large applications in active code repositories use multi-threading [Okur & Dig FSE 2012]

	Small (1K-10K)	Medium (10K-100K)	Large (>100K)
# of all projects in the study	6020	1553	205
# of projects with multithreading	**1761**	**916**	**178**
# of projects with parallel library uses	412	203	40

- **Research goal** *Develop automated test generation for concurrent programs to detect concurrency bugs effectively & efficiently*
- **Approach** *Utilize concurrent code coverage metrics in automated test generation of concurrent programs*

Approach

- Research challenges: utilize conc. coverage sound and effectively
  - Is achieving high concurrent code coverage useful for testing multithreaded programs?
    - Empirical study on concurrent coverage metrics and their impacts on testing effectiveness [Hong et al. ICST 2013]
  - How to generate high concurrent code coverage achieving test executions fast?
    - Estimation-based thread scheduling algorithm [Hong et al. ISSTA 2012]
  - Is there a better way to use concurrent coverage metric for testing ?
    - Set-coverage metric
    - High set-coverage achieving thread scheduling
    - Set-coverage based distributed test generation (on-going work)

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.
  - Statement-based 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), … }

Impact of Conc. Coverage on Test. Effectiveness

- *Concurrent coverage metrics* have been proposed to support systematic testing of concurrent programs
  - A coverage metric derives test requirements from a target program, which should be satisfied at least in a testing
  - Several distinct concurrent coverage metrics have been proposed

- Intuition behind: as **more test requirements** for the metrics are **satisfied**, the testing process becomes **likely to detect faults**

  However, no empirical evaluation and no quantification in different coverage metrics

Research Questions

- Does a testing achieving higher code coverage detect more faults than one achieving lower code coverage ?
  - RQ1: for given two test suites of equal size, is the test suite with higher coverage in a metric generally more effective ?
    - Does the coverage achieved positively impact the testing effectiveness?
  - RQ2: is the test suite achieving maximum coverage generally more effective than random test suite of equal size?
    - Can we use concurrent coverage as a test reduction target?
    - Is it “safe” to generate testing directed to increase coverage of a metric?

Study Design

- RQ1: for two test suites of equal size, is the test suite with higher coverage in a metric generally more effective ?
- RQ2: is the test suite achieving maximum coverage generally more effective than random test suite of equal size?

- Independent variables
  - Concurrent coverage metrics
    - Existing eight coverage methods
  - Test suite construction
    - Random test suite construction of a given test suite size
    - Greedy selection for a given coverage level of a metric
- Dependent variables
  - Achieved concurrent coverage of a test suite in a metric
  - Test suite size
  - Mutation score (when a target program is a mutation system)
  - Singe fault detection (when a target program is a single fault system)

Concurrent Coverage Metrics Studied

- We selected eight concurrent coverage metrics for the study, that are well-known while ensuring the diversity in the selection
  - A concurrent coverage metric has two key properties:
    - **Type of code element** that the metric is defined over (either synchronizations, or shared data accesses)
    - **Number of code elements** that a test requirement considers (either a single element, or a pair of elements)

	Synchronization operation	Data access operation
Singular	*blocking* [9], *blocked* [9]	*LR-Def* [2]
Pairwise	*blocked-pair* [3], *follows* [3], *sync-pair* [12]	*PSet* [20], *Def-Use* [16]

Experiment Setup

- Conducting our experiment requires us to
  (1) Prepare faulty programs
  (2) Conduct a large number of random test executions
  (3) Record for each execution the test requirements covered for all metrics and fault detection
  (4) Construct test suites by resampling over executions, and

Noise-injection based random testing

- Insert a noise injection probe before every shared variable access, and every lock acquire operation
- Probe makes time delay of a thread execution for *T* sec for a probability *P*
- Use 12 combinations of *T* and *P* and normal program execution
  - *T*: 5 msec, 10 msec, and 15 msec
  - *P*: 0.1, 0.2, 0.3, 0.4

Experiment Setup: Test Suite Construction

- Study for RQ1
  ① Construct a test suite for each coverage point in a metric M,
    - Mutation systems: generate test suites for each mutant

  ② For each constructed test suite, measure test suite size and fault detection
    - Single fault systems
      – Size: # of test execution in a test suite
      – Fault detection: 1 if any exec. in a TS detects an error, 0 otherwise.
    - Mutation systems
      – Size: average # of executions in test suites over mutants
      – Fault detection: # of mutants killed by their test suites

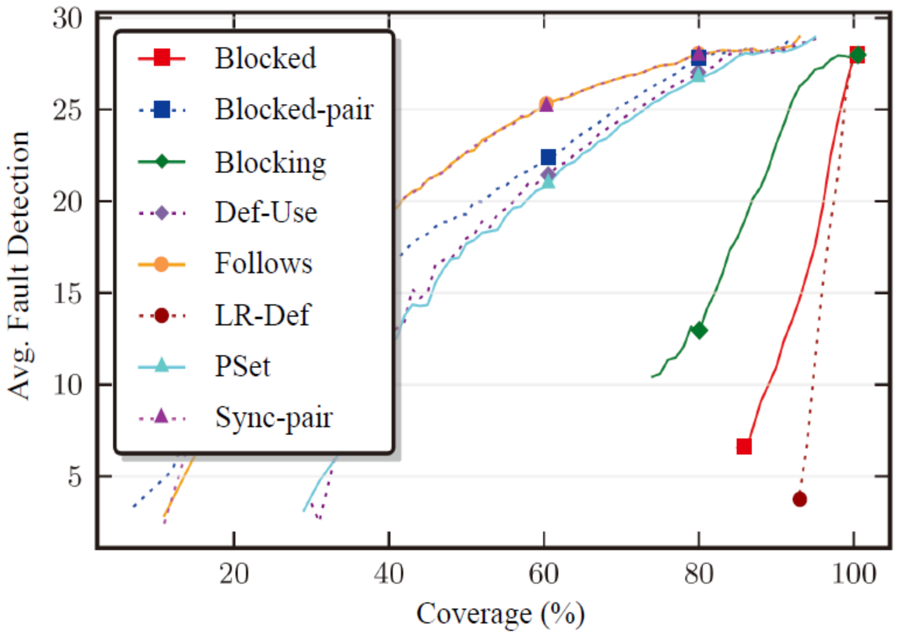
- Study for RQ2
  ① Find the maximum coverage in a metric M

  ② Construct a test suite MAX that achieve maximum coverage whose size is minimum

  ③ Construct a test suite RND whose size is the same as MAX but collects executions randomly

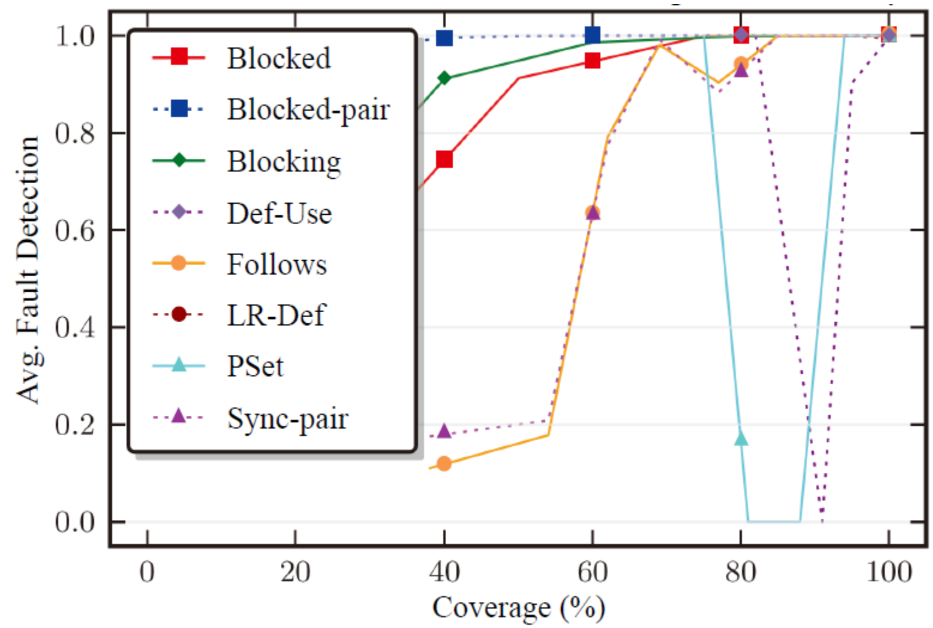
  ④ Measure fault detection of MAX and RND as similar to RQ1 study

Result: Correlations in CV and FF



(a) Coverage vs Fault Detection Effectiveness

Vector



(a) Coverage vs Fault Detection Effectiveness

Stringbuffer

- We measured (1) the correlations between each coverage and testing effectiveness, and (2) the correlations between TS size and testing effectiveness

→ (1) concurrent coverage metrics are moderate to strong predictor of concurrent testing effectiveness

(2) concurrent coverage is often more strongly correlated with testing effectiveness than test suite size

Result: Effectiveness of Maximum Coverage

	follows				LR-Def				PSet			
	MFF	RFF	Cv	Sz	MFF	RFF	Cv	Sz	MFF	RFF	Cv	Sz
ArrayList	7.23	4.06	47.0%	20.2	7.46	0.78	2.68%	1.41	7.38	2.72	28.1%	8.56
BoundedBuffer	4.23	3.95	87.0%	42.7	2.72	2.93	13.3%	2.96	4.38	3.80	32.4%	17.7
Vector	21.0	23.1	56.2%	121	27.8	7.85	5.15%	2.93	27.8	19.7	53.8%	45.5

- MFF: fault detection of maximum coverage test suite

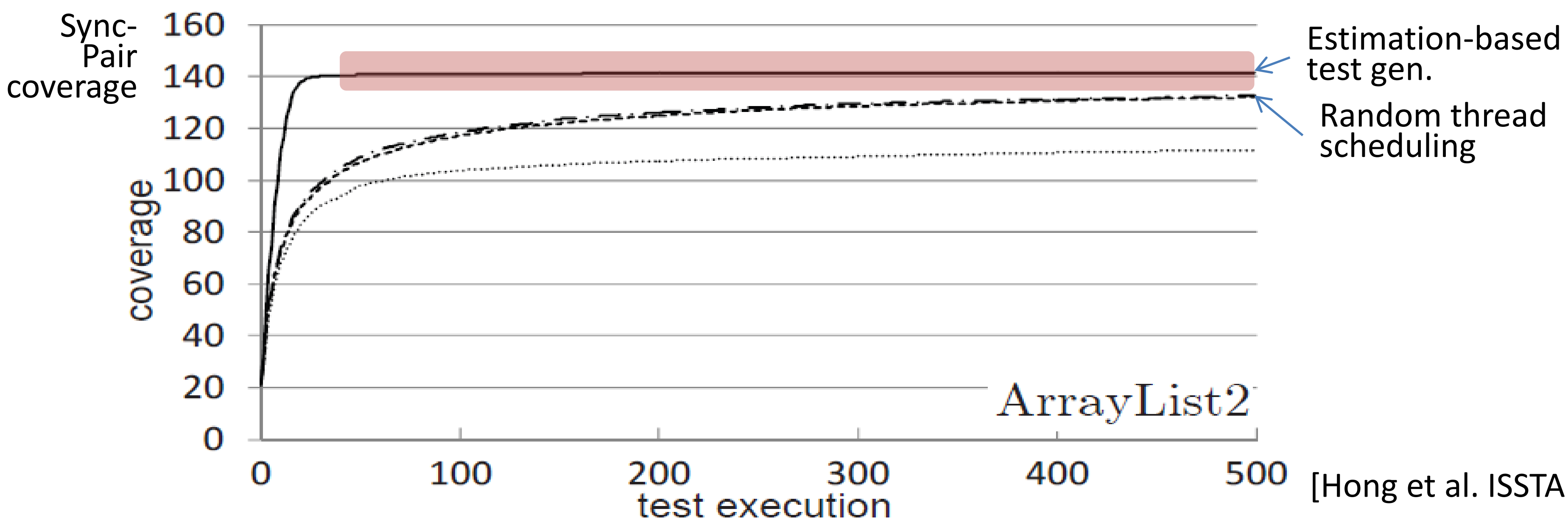
  RFF: fault detection of random test suite of equal size of MFF
- The result implies that achieving high coverage generally yields significant increases in fault detection
  - For example of a mutation system *ArrayList*, increases in average fault detection of 1.7 to 9.5 times (MFF / RFF) at maximum coverage
  - This result implies that that concurrent coverage metrics can be used for directed test generation
- However, in many cases, MFF fails to achieve maximum fault detection achieved by larger test suite of equal coverage
  - For example of ArrayList, maximum fault detection is more than 8

Discussion: Basic Guideline for Practitioner

- Q: Which metric among eight should I use? A: PSet
  - Has generally high correlation with fault detection
  - Achieves always greater correlation with fault detection than test suite size
- Pairwise metrics are preferable for predictors of testing effectiveness
  - The correlation with fault detection for pairwise metrics tends to be higher or equal than that for singular metric
- Pairwise metrics excel as targets for test case generation
- Using *PSet + follows* would be better than just using a metric alone
  - A large difference in fault detection exists depending on the primitive (synchronization/data access) used to define the metrics
  - Metrics excellent in some circumstances perform poorly in others
- No coverage metric is a perfect test generation target !

Set Coverage Testing: Motivation (1/2)

- Testing beyond coverage saturation



[Hong et al. ISSTA 12]

- Limitation of existing concurrent coverage directed test generation
  - Existing coverage criteria does not provide effective guidance after covering all feasible test requirements
  - Existing coverage-guided test generation is no more effective after reaching likely-saturation than random testing

Set Coverage Testing: Motivation (2/2)

- Measuring test requirements covered in an execution provides useful information
  - A set of test requirements derived from a program is a good abstraction of thread interaction cases in the program behavior
- Is there a better way of utilizing coverage metric?
  - In test generation after reaching likely-saturation to avoid redundant test executions
  - In systematic exploration to reach corner case test requirement
  - In distributed testing where plenty of computing resources are available

➔ *Set coverage criteria* of a metric *M*

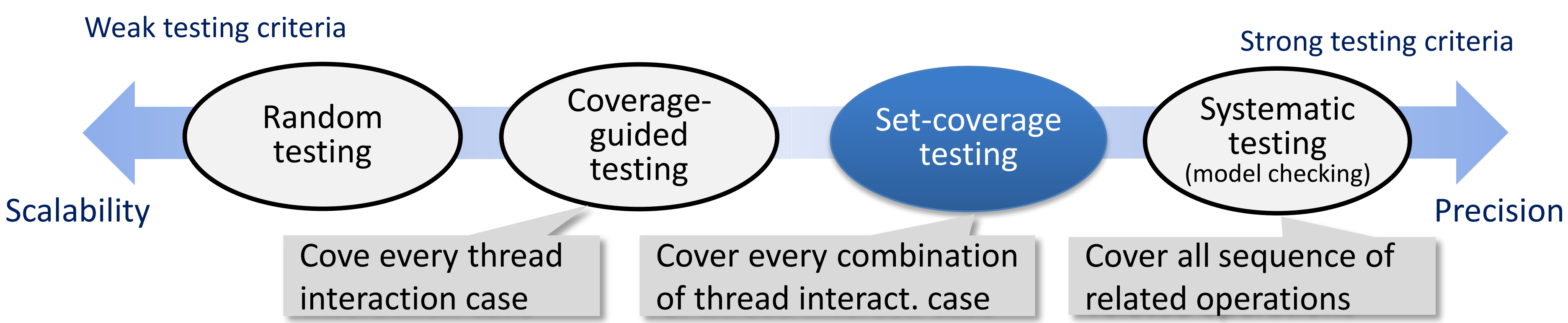
Test all possible combinations of test requirements derived by *M*

Set Coverage Definition

- Set coverage criteria of a metric *M*: for test requirements by *M*, a testing should cover all *combinations* of test requirements
  - A *test requirement set* $\{tr_1, tr_2, \ldots, tr_N\}$ is covered for an execution when there is an execution in a testing that satisfies tr_1, tr_2, .. and tr_N.
  - Set(*N*) coverage: the number of test requirement sets of size *N* covered in a testing
    - Suppose that test requirements $t_1, t_2, \ldots, t_M$ for a program exist
      - Set(2) coverage counts for $\{t_1,t_2\}, \{t_1,t_3\}, \ldots, \{t_{M-1},t_M\}$
      - Set(3) coverage counts for $\{t_1,t_2,t_3\}, \{t_1,t_2,t_4\}, \ldots, \{t_{M-2}, t_{M-1}, t_M\}$
      - Set(1) coverage = conventional coverage
      - Set(*) coverage ≈ Path coverage

Intuition behind Set Coverage

- Set coverage criteria provides simple test generation targets to complex test generation target gradually
- Certain concurrency error scenarios are characterized by sequence of 2~3 thread interactions
  - A subtle program behavior can be triggered after certain thread interactions



Set Coverage Guided Test Generation

- Goal: perform fast Set(1) coverage as existing technique as well as fast & progressive increase of Set(N) coverage after saturation

	Early testing phase		After Set(1) saturation	
	Set(1) cov.	Set(N) cov.	Set(1) cov.	Set(N) cov.
Random thread scheduling	Moderate	Moderate	Progress in low chance	Moderate
Estimation-based test generation	High	High	Not progressive	Low
Model checking	Low	Low	Progressive	High
Set cov. guided test generation	**High**	**High**	**Progressive**	**High**

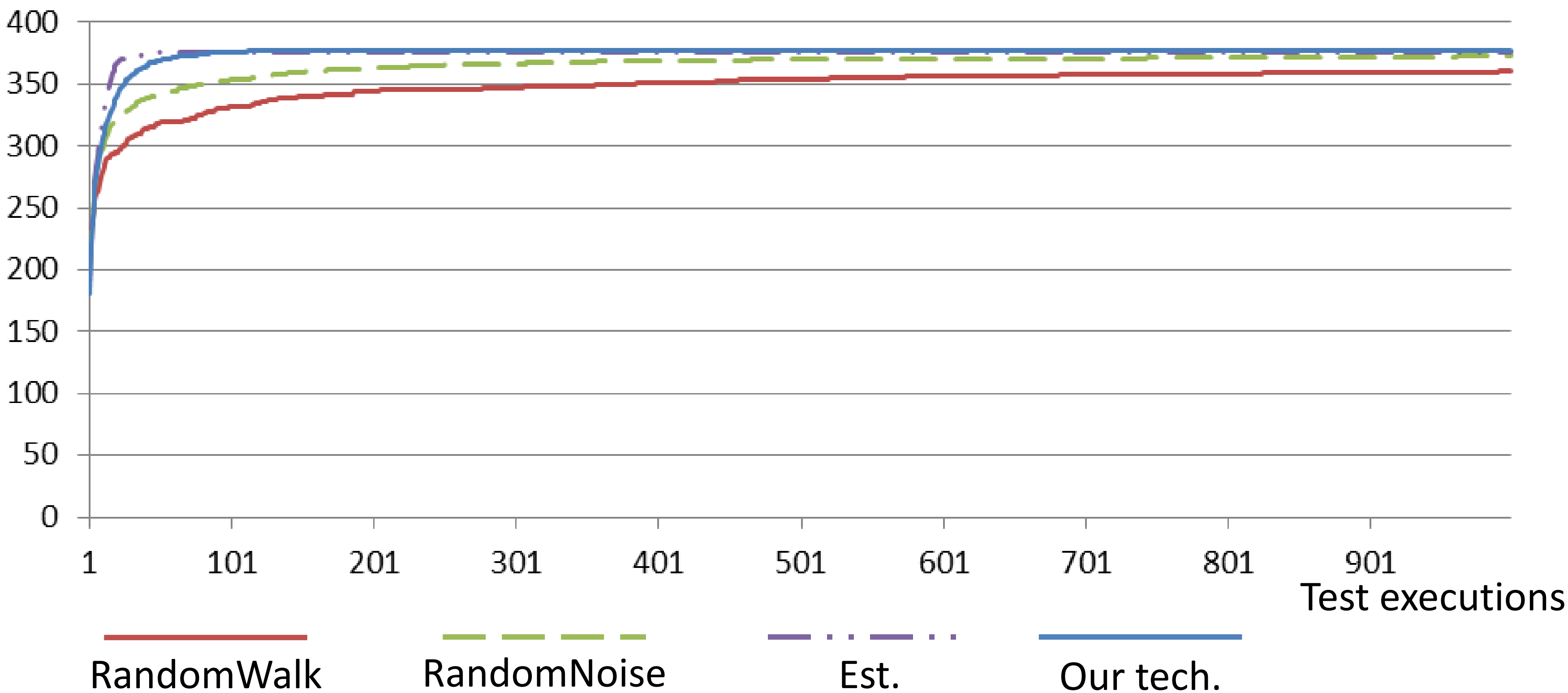
Thread Scheduling Algorithm

- Naïve approach
  - Method: record all possible test requirement sets and check a thread scheduling decision cover unseen test requirement sets
  - Limitation: saving test requirement sets incurs infeasible overhead
    - For example, in testing *ArrayList*, # of PSet +SyncPair test. req. > 300, and # of Set(3) test requirement sets is around 7×10^6
- Idea
  - Conjecture: a testing with high Set(N) coverage covers **Set(2) test requirement sets in many times evenly**
    - A testing with low Set(N) coverage of equal size will cover certain test requirement set of Set(2) more frequently than others
  - Method:
    (1) For each TR set of size 2, count # of test exec. covering the TR set
    (2) Select an operation at a thread scheduling decision to cover most infrequently covered test requirement set of size 2

Preliminary Experiment Result (1/3)

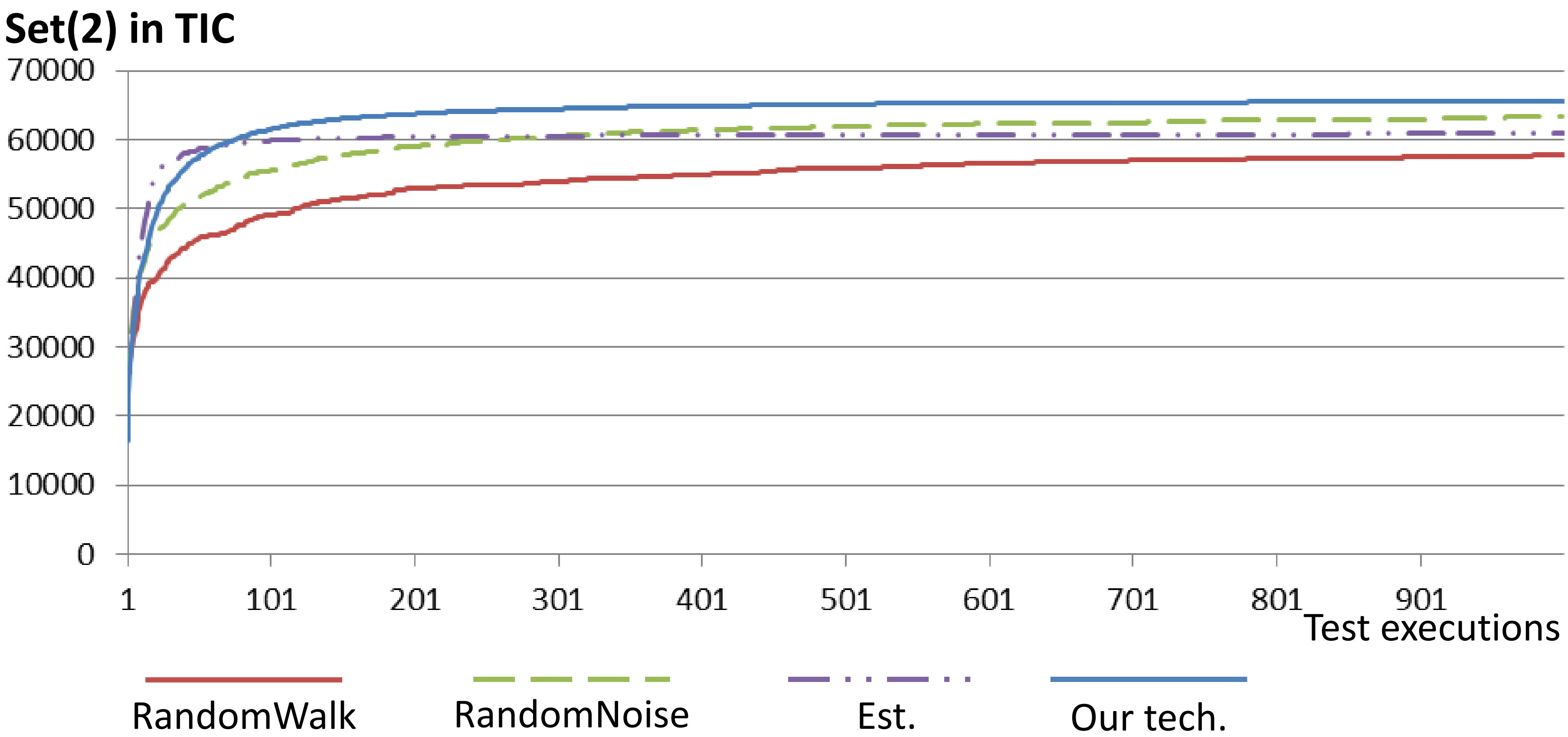
- Comparing set coverage performance of our technique to existing ones
  - Study subject is Java Collection ArrayList with `synchronizedList`
  - Measure in TIC metric (*PSet + follows*)
  - Three different measurements of a single experiment

Set(1) in TIC



Preliminary Experiment Result (2/3)

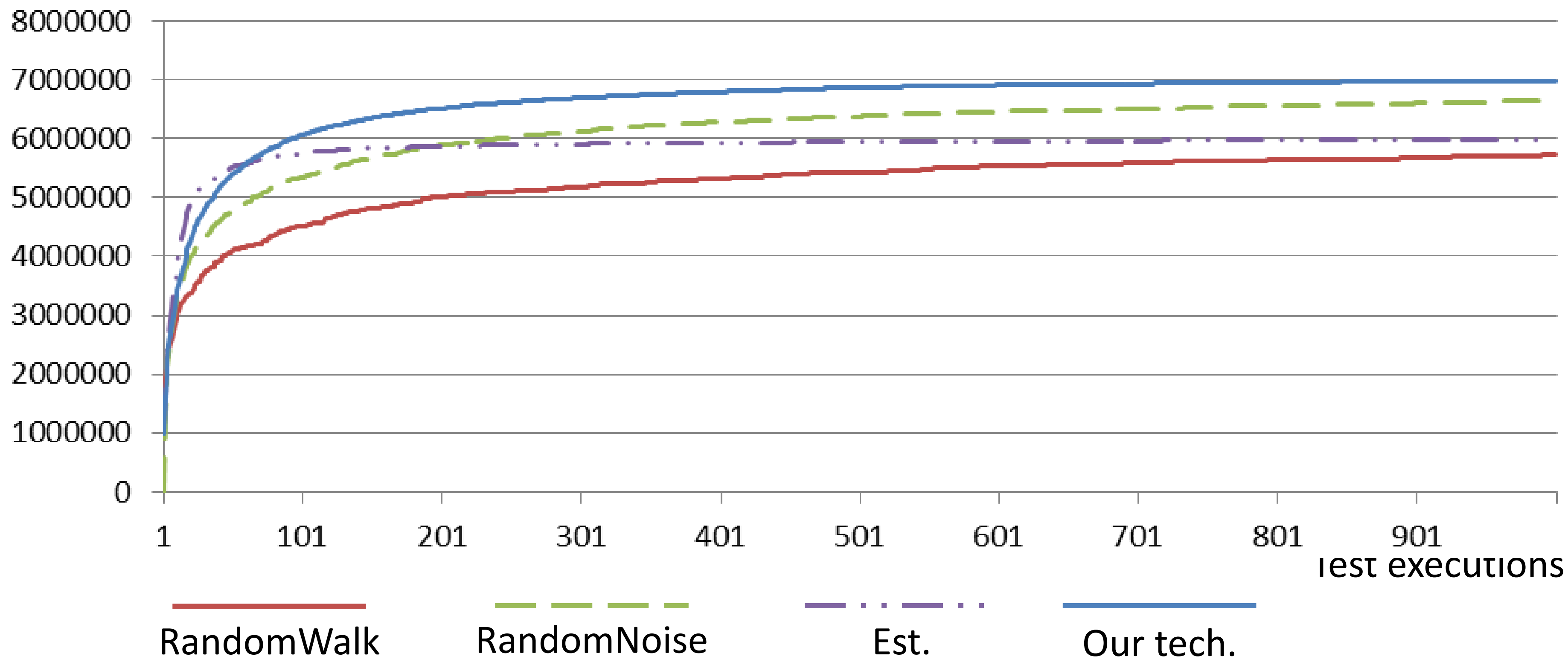
- Comparing set coverage performance of our technique to existing ones
  - Study subject is Java Collection ArrayList with `synchronizedList`
  - Measure in TIC metric (*PSet + follows*)
  - Three different measurements of a single experiment



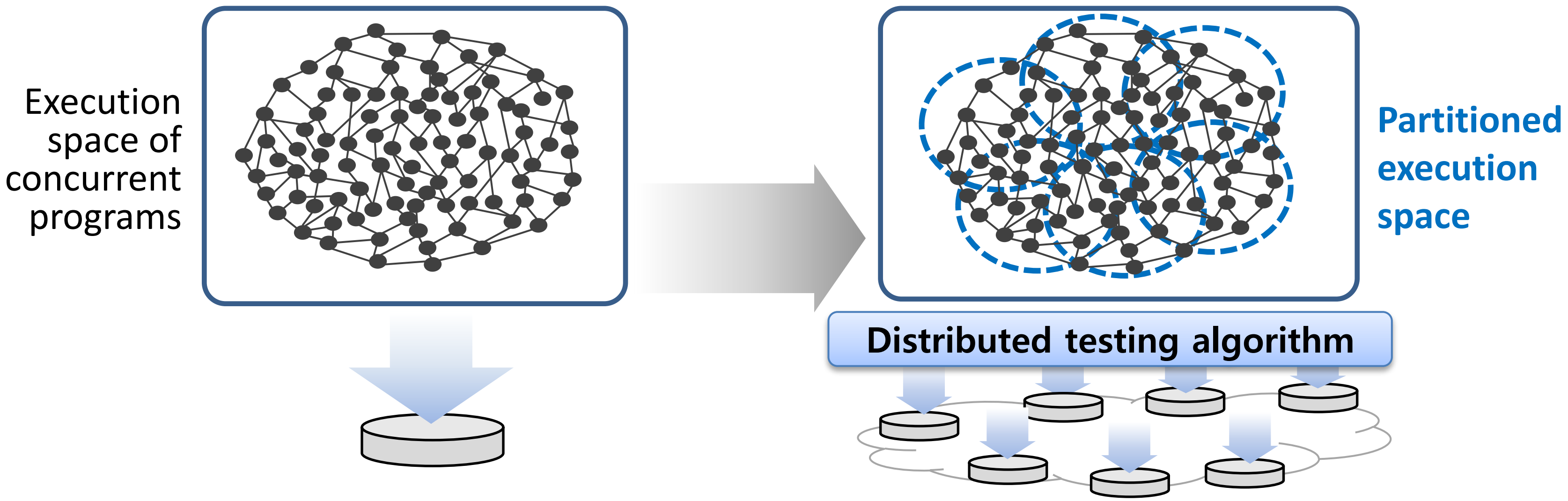
Preliminary Experiment Result (3/3)

- Comparing set coverage performance of our technique to existing ones
  - Study subject is Java Collection ArrayList with `synchronizedList`
  - Measure in TIC metric (*PSet + follows*)
  - Three different measurements of a single experiment

Set(3) in TIC



Distributed Set Coverage Testing: Application



- Utilize distributed computing resources effectively to accelerate test generation!
- Effective distributed testing requires the technique to guarantee
  - Each node should generate non-redundant test executions progressively
  - Test executions generated in different nodes may not overlap

➔ **Use set coverage as a testing task partitioning criteria**

Test Distribution by Scheduling Constraints

- Use *scheduling constraints* to parallelize set coverage testing tasks
  - A scheduling constraint is a propositional formula over test requirements generated by a concurrent coverage metric (e.g. *Pset + Sync-Pair*)
  - A node should generate executions satisfying assigned scheduling constraint
    - Suppose the test requirements for a program are t_1, t_2, …, t_M.
    - A node assigned for a scheduling constraint $f = t_1 \lor (t_2 \land \neg t_3)$ should generate every execution generated by the node must cover either t_1, or t_2 without covering t_3 (, and no other restriction)
  - Scheduling constrains in a testing must satisfy the following two conditions:
    - Each formula assigned for a node should be exclusive to others
    - The disjunction of formulas should cover all test requirement sets

Work in Progress

- Develop an algorithm to generate *good* scheduling constraints
  - Check dependency in test requirements by analyzing program structures
  - Analyze previous execution results to find test requirements appropriate to be in scheduling constraints
- Develop a mechanism of dynamic testing load balancing
- Empirically evaluate benefit of using set coverage as a test generation target

코드 커버리지를 이용한 동시성 프로그램 테스트 자동 생성

홍 신 안재민 Matt Staats 김문주

PROVABLE SW LAB

KAIST