ASPLOS

Selectively Uniform Concurrency Testing

Huan Zhao

Co-Authors: Dylan J. Wolff Umang Mathur Abhik Roychoudhury

{ zhaohuan, wolffd, umathur, abhik } @comp.nus.edu.sg

NUS
National University of Singapore

Overview

Background

Model Controlled Concurrency Testing (CCT)
as a sampling process

Our Work

What makes a sampling process ideal?

Concurrent Programs (1)

Critical digital infrastructure

Disastrous concurrency bugs

Northeastern Blackout (2003)

Therac-25 Radiation Therapy Machine (1980s)

Boeing 787 Dreamliner (2013)

Concurrent Programs (2)

- Non-deterministic behaviors depending on *interleavings*, i.e., ordering of thread executions

- With 2 threads each with 5 **atomic** events, there are 252 possible observable behaviors (value of x)

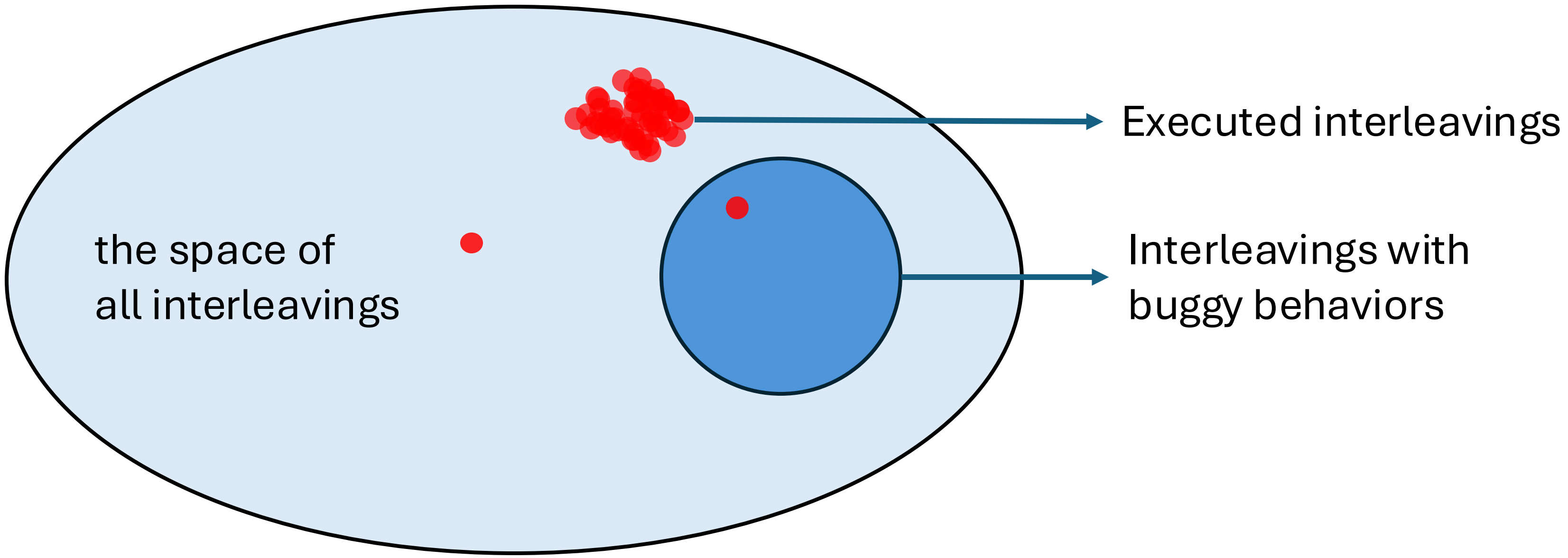
→ Concurrency bugs may only manifest in few interleavings
→ Difficult to expose and reproduce concurrency bugs!

```
void thread_A() {
    x = x<<1;
    x = x<<1;
    x = x<<1;
    x = x<<1;
    x = x<<1;
}
```

```
void thread_B() {
    x = x<<1+1;
    x = x<<1+1;
    x = x<<1+1;
    x = x<<1+1;
    x = x<<1+1;
}
```

Stress Testing

- Repeated and uncontrolled executions with the default OS scheduler
- However, wildly different interleavings are possible when deployed, due to unpredictable changes to the system workload!



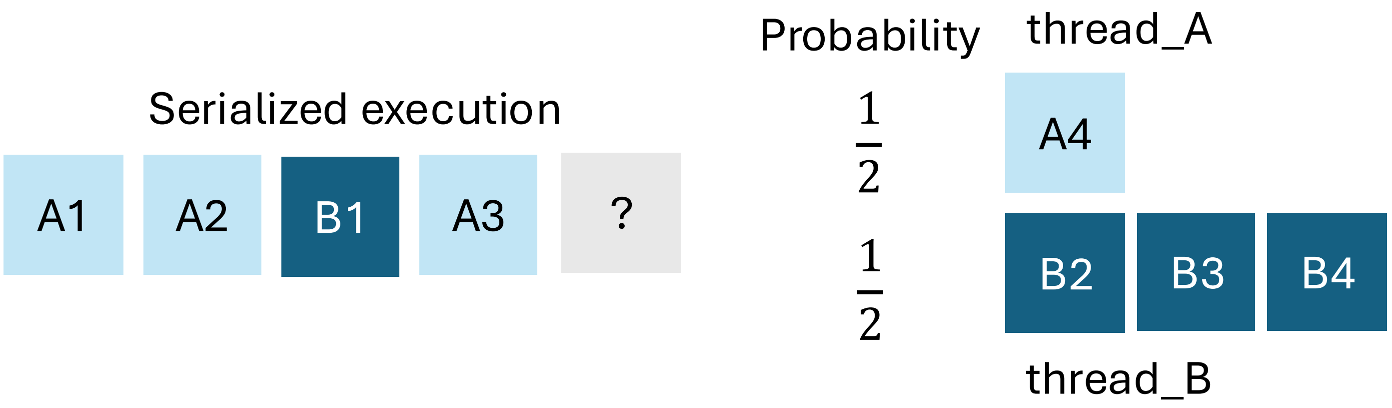
Controlled Concurrency Testing (CCT)

1. *Serialized* program execution

2. *Online* decision of which event to run next

Randomized CCT: Random Walk

1. *Serialized* program execution
2. Choose each thread to run *with equal prob.*

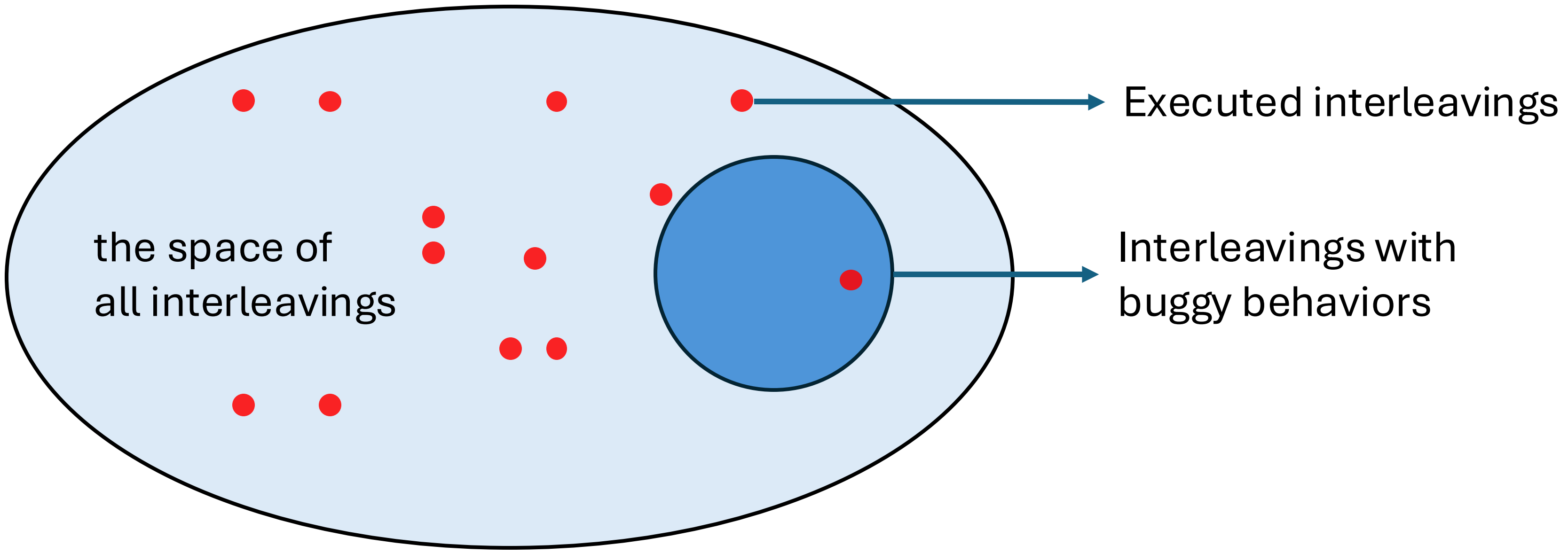



```
void thread_A() {
    x = x<<1;
    x = x<<1;
    x = x<<1;
    x = x<<1;
    x = x<<1;
}
```

```
void thread_B() {
    x = x<<1+1;
    x = x<<1+1;
    x = x<<1+1;
    x = x<<1+1;
    x = x<<1+1;
}
```

CCT as a Sampling Process

- Sampling of interleavings according to a prob. distribution
- Lightweight yet effective schedule generation, often with prob. guarantees
- Reproduces exposed bugs deterministically



What makes

a sampling process ideal?

Uniformity

The **optimal** sampling strategy:

Sample each *program behavior* with equal probability (i.e., uniformly)

→ maximize the *minimum probability* of any behavior being sampled

However, behaviors are program dependent and unknown a priori!

The most fine-grained notion of behaviors: interleavings!

Motivating Example (1)

- Interleavings ←→ (observable) program behaviors, i.e., value of x
- Total number of interleavings / behaviors: (10 choose 5) = 252
- An ideal algorithm samples each value of *x* w.p. 1/252

```
void thread_A() {
    x = x<<1;
    x = x<<1;
    x = x<<1;
    x = x<<1;
    x = x<<1;
}
```

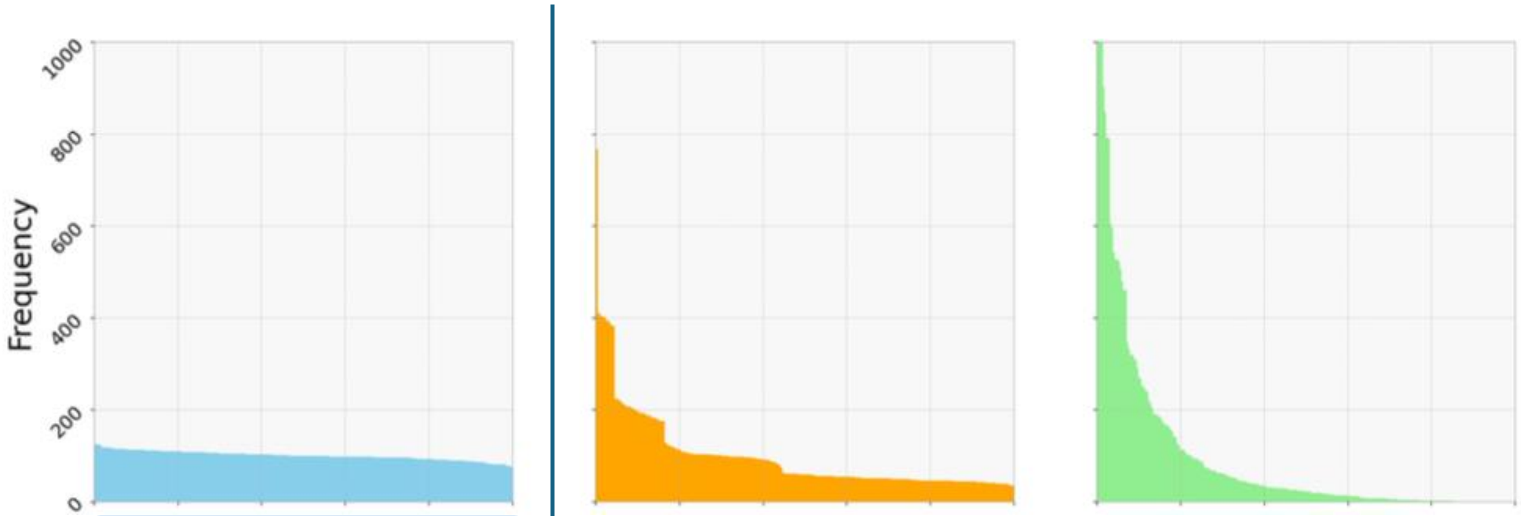
```
void thread_B() {
    x = x<<1+1;
    x = x<<1+1;
    x = x<<1+1;
    x = x<<1+1;
    x = x<<1+1;
}
```

Motivating Example (2)

Total # of int. / behaviors = 252

```
void thread_A() {
    x = x<<1;
    x = x<<1;
    x = x<<1;
    x = x<<1;
    x = x<<1;
}
```

```
void thread_B() {
    x = x<<1+1;
    x = x<<1+1;
    x = x<<1+1;
    x = x<<1+1;
    x = x<<1+1;
}
```



Ideal

Random Walk
Partial Order Sampling [1]

PCT-10 [2]

Existing strategies are *highly* biased!

For RW / POS: 2 events are sampled w.p. 1/32; but 140 events are sampled w.p. 1/512
For PCT-10: 38 interleavings are not witnessed *once* in >25,000 executions in our experiment!

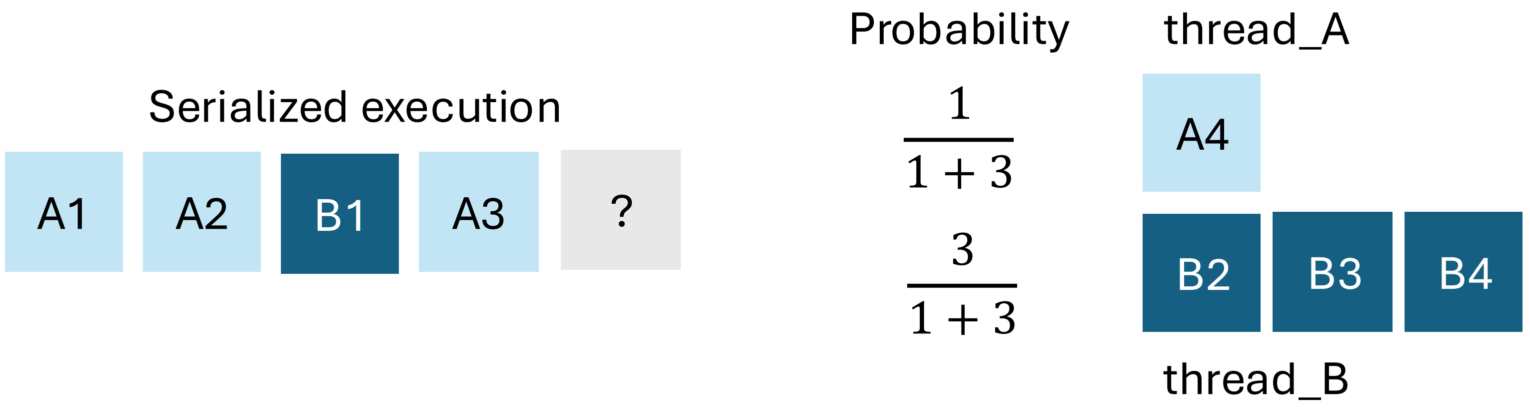
[1] Yuan, Xinhao, et al., 2018. "Partial order aware concurrency sampling." *CAV 2018.*
[2] Burckhardt, Sebastian, et al., 2010. "A randomized scheduler with probabilistic guarantees of finding bugs." *ACM SIGARCH Architecture News* 38(1).

Key Insight 1: Interleaving Uniformity

(a) Interleaving is a proxy of program behaviors;

(b) Interleaving uniformity is desirable,

and achievable with a **weighted** random walk



Selectivity

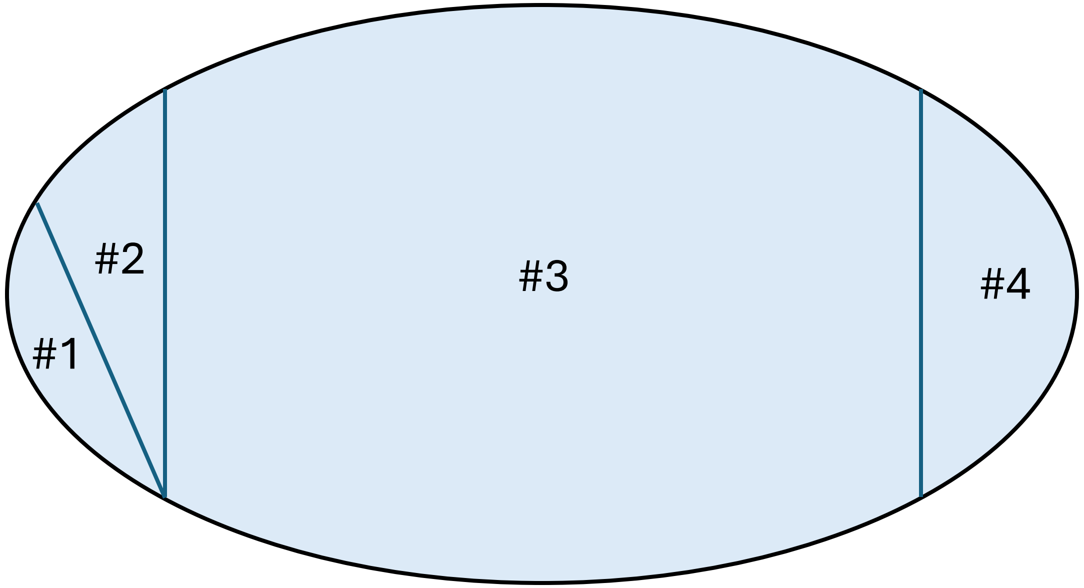
- Interleaving ≠ behavior. Interleaving uniform ≠ behavior uniform.

  e.g., 10 interleavings may result in the same observable behavior (value of x)!

- With interleaving uniformity, most samples result in behavior #3!

- Behaviorally redundant samples!

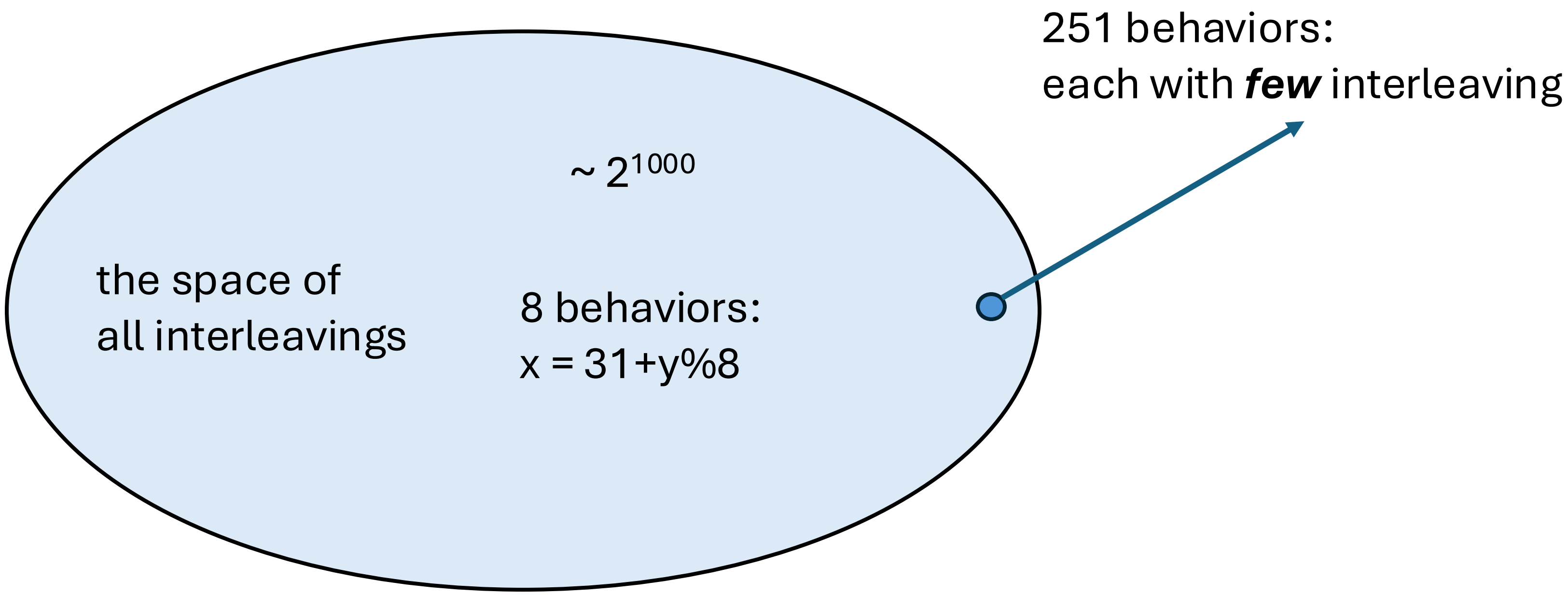
the space of all interleavings,
partitioned by *program behaviors*



Motivating Example (1)

- Interleaving uniformity means that

  most behaviors will almost never be sampled!

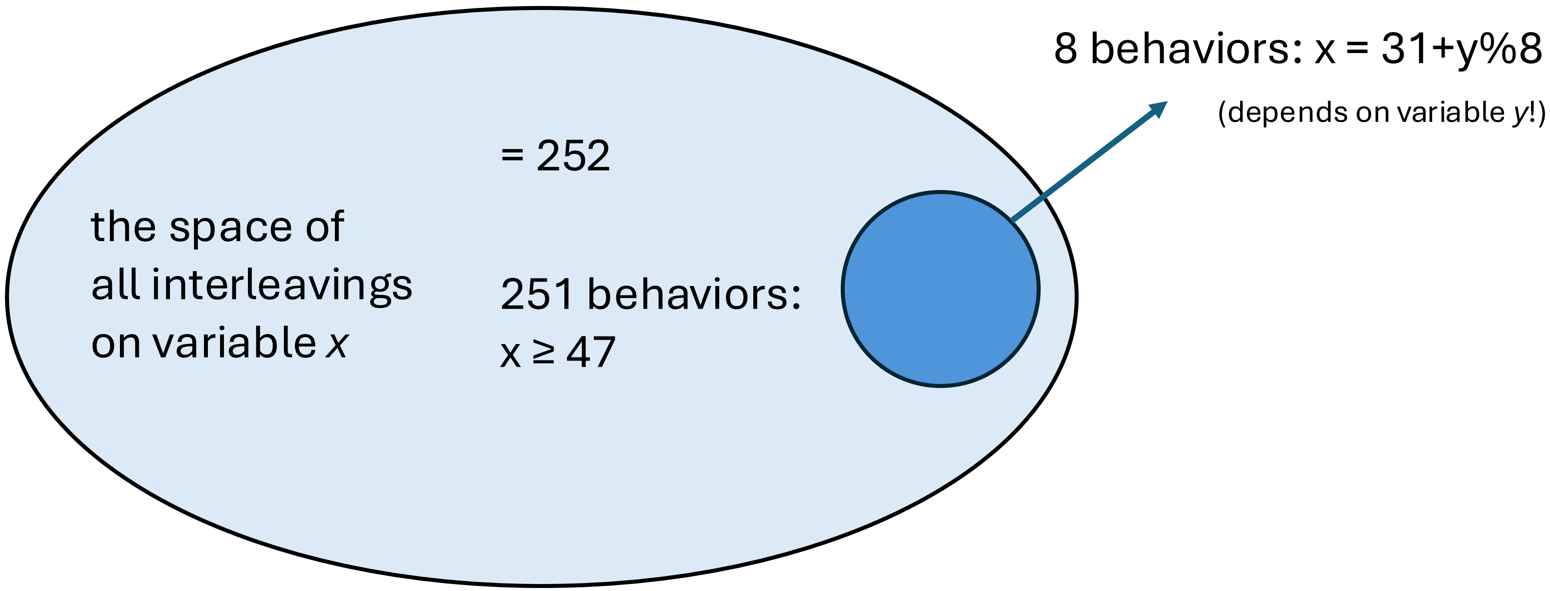



```
void thread_A() {
    x = x<<1;
    x = x<<1;
    x = x<<1;
    x = x<<1;
    x = x<<1;
    y = y<<1;
    // repeat 1000x
    y = y<<1;
}
```

```
void thread_B() {
    y = y<<1+1;
    // repeat 1000x
    y = y<<1+1;
    x = x<<1+1;
    x = x<<1+1;
    x = x<<1+1;
    x = x<<1+1;
    x = x<<1+1+y%8;
}
```

Motivating Example (2)

- The projected interleavings of variable *x* are much more balanced in the behavioral space!




```
void thread_A() {
    x = x<<1;
    x = x<<1;
    x = x<<1;
    x = x<<1;
    x = x<<1;
    y = y<<1;
    // repeat 1000x
    y = y<<1;
}
```

```
void thread_B() {
    y = y<<1+1;
    // repeat 1000x
    y = y<<1+1;
    x = x<<1+1;
    x = x<<1+1;
    x = x<<1+1;
    x = x<<1+1;
    x = x<<1+1+y%8;
}
```

Key Insight 2: Selective uniformity

Uniform sampling of the interleavings of

an appropriate <u>subset of program events</u>

achieves effective *behavioral* exploration

Key Insight 2: Selective uniformity

Uniform sampling of the interleavings of

an appropriate <u>subset of program events</u>

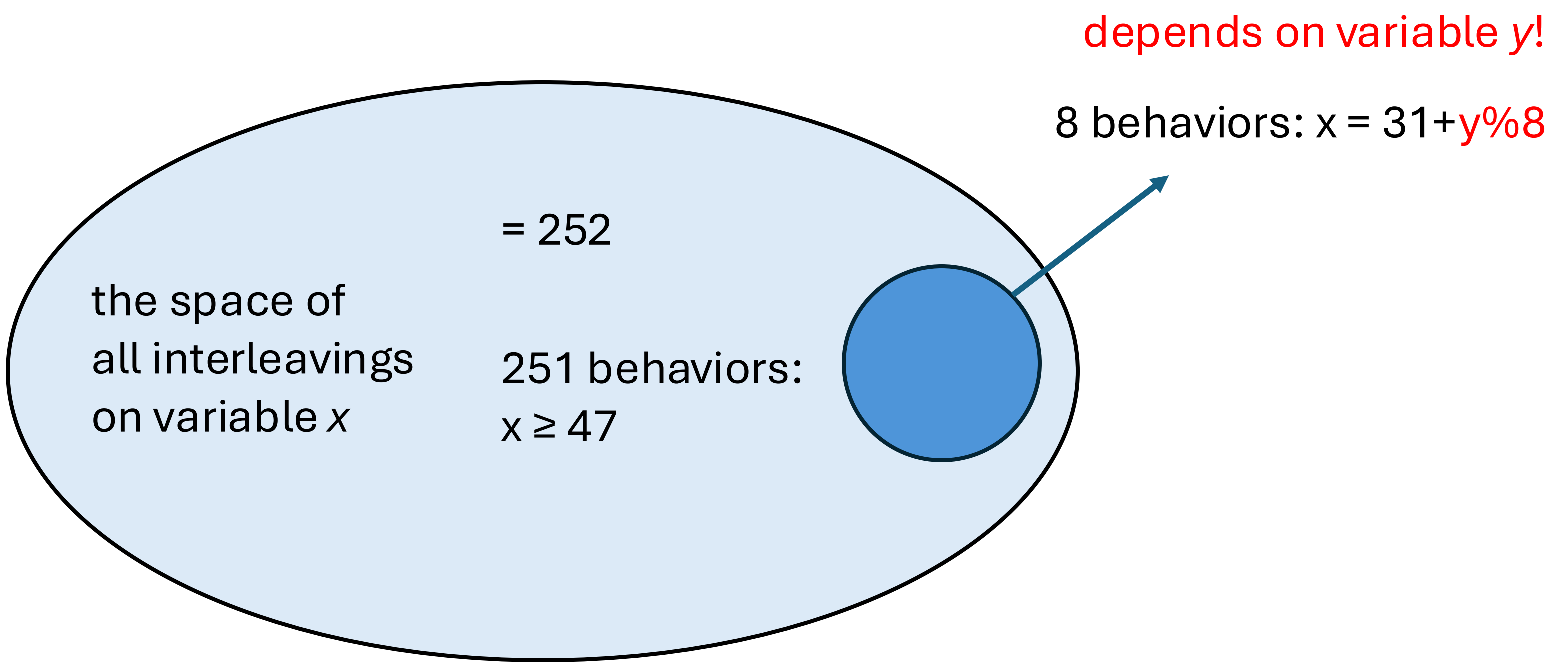
achieves effective *behavioral* exploration

Given a subset of event $\Delta \subseteq \Sigma$, we want to

[Uniformity] sample the interleaving of Δ uniformly

Motivating Example (3)

- At the same time, we should not disable any interleaving of *y*, or between *x* and *y*!




```
void thread_A() {
    x = x<<1;
    x = x<<1;
    x = x<<1;
    x = x<<1;
    x = x<<1;
    y = y<<1;
    // repeat 1000x
    y = y<<1;
}
```

```
void thread_B() {
    y = y<<1+1;
    // repeat 1000x
    y = y<<1+1;
    x = x<<1+1;
    x = x<<1+1;
    x = x<<1+1;
    x = x<<1+1;
    x = x<<1+1+y%8;
}
```

Key Insight 2: Selective uniformity

Uniform sampling of the interleavings of

an appropriate <u>subset of program events</u>

achieves effective *behavioral* exploration

Given a subset of event $\Delta \subseteq \Sigma$, we want to

(1) [Uniformity] sample the interleaving of Δ uniformly

(2) [Completeness] sample any interleaving over Σ with non-zero probability

Selective Uniform Random Walk

Given a subset of event $\Delta \subseteq \Sigma$, we want to

(1) [Uniformity] sample the interleaving of Δ uniformly

(2) [Completeness] sample any interleaving over Σ with non-zero probability

Central Idea:

- Make *eager* decisions about the interleaving of Δ with weighted random walk
- Only schedule Σ - Δ so that the decision on Δ is respected

Constant time per step: O(# of threads)

More details discussed in the paper!

Algorithm 2: `SURW`

Input: Δ; `// set of interesting events`
Input: $n_1, \ldots, n_k$; `// interesting event counts`
$T_{iNext} \leftarrow$ random T_i weighted by n_i;
blocked $\leftarrow \emptyset$;
while $E \leftarrow$ `getEnabled()` $\neq \emptyset$ **do**
    $T_t \leftarrow$ `pickFrom(`E - *blocked*`)`;
    if `nextEvent(`T_t`)` $\in \Delta$ **then**
        if $T_{iNext} == T_t$ **then**
            $n_t \leftarrow n_t - 1$;
            $T_{iNext} \leftarrow$ random T_i weighted by n_i;
            blocked $\leftarrow \emptyset$;
        else
            blocked.add(T_t); **continue**;
    `execute(nextEvent(`T_t`))`;

Evaluation (1)

[RQ1] Is SURW better at exposing bugs compared to other concurrency testing algorithms?

[RQ2] How do the two key components of SURW, uniformity and selectivity, contribute to its effectiveness?

Evaluation (2)

3 established concurrency testing benchmarks in the community [1-3]

Other state-of-the-art algorithms: PCT-3 [4], PCT-10 [4], POS [5]

Baselines: Random Walk, Non-Selective, Non-Uniform

[1] Thomson, Paul, et al., 2016. "Concurrency testing using controlled schedulers: An empirical study." *TOPC* 2.4 (2016).
[2] Meng, Ruijie, et al., 2019. "ConVul: an effective tool for detecting concurrency vulnerabilities." *ASE* 2019.
[3] Liang, Jiashuo, et al., 2023. "RaceBench: A Triggerable and Observable Concurrency Bug Benchmark." *AsiaCCS* 2023.
[4] Yuan, Xinhao, et al., 2018. "Partial order aware concurrency sampling." *CAV 2018.*
[5] Burckhardt, Sebastian, et al., 2010. "A randomized scheduler with probabilistic guarantees of finding bugs." *ACM SIGARCH Architecture News* 38(1).

RQ1 Bug Finding (1)

[Metric 1]. Average **# of bugs** exposed by different algorithms (the higher, the better)

Benchmark	SURW	PCT	POS	Random Walk
SCTBench (max. 37)	**34.90**	30.75	29.25	19.90
		+13%	+19%	+75%
RaceBenchData (max. 1,500)	**944**	461	885	489
		+105%	+7%	+93%

[Metric 2]. Average **# of interleavings** sampled to expose each bug (the lower, the better)

- On 26 / 35 targets, SURW requires the minimum number of schedules
- On 6 / 9 other targets, SURW requires only < 10 schedules on average

RQ1 Bug Finding (2)

Target	SURW	PCT-3	PCT-10	POS	Random Walk
CS/twostage	**8 ± 4**	**13 ± 14**	**9 ± 10**	15 ± 12	464 ± 581
CS/twostage_20	**6 ± 3**	159 ± 151	101 ± 92	156 ± 137	–
CS/twostage_50	**20 ± 17**	1676 ± 1715	692 ± 392	1637 ± 1385	–
CS/twostage_100	$\boxed{\textbf{454 ± 444}}$	7466 ± 831*	5726 ± 2591*	6674 ± 2877*	–
CS/reorder_3	**7 ± 7**	185 ± 199	148 ± 191	86 ± 70	–
CS/reorder_4	**7 ± 6**	554 ± 643	362 ± 213	533 ± 651	–
CS/reorder_5	**10 ± 9**	647 ± 517	1094 ± 1216	2169 ± 2182	–
CS/reorder_10	**17 ± 11**	3225 ± 2426*	4462 ± 3266*	–	–
CS/reorder_20	**6 ± 4**	3005 ± 2680	3297 ± 2877*	–	–
CS/reorder_50	**13 ± 12**	3304 ± 1721*	–	–	–
CS/reorder_100	$\boxed{\textbf{194 ± 214}}$	–	–	–	–
ConVul/CVE-2013-1792	**15 ± 13**	95 ± 83	50 ± 61	39 ± 33	364 ± 289
ConVul/CVE-2016-1972	$\boxed{\textbf{11 ± 8}}$	4902 ± 2391*	2712 ± 2704*	34 ± 34	299 ± 256
ConVul/CVE-2016-1973	**5 ± 3**	10 ± 8	**6 ± 3**	**5 ± 4**	308 ± 333
ConVul/CVE-2016-7911	8 ± 9	20 ± 18	15 ± 15	11 ± 9	**3 ± 2**
ConVul/CVE-2016-9806	**3 ± 2**	7 ± 6	**4 ± 3**	7 ± 5	2209 ± 2065
ConVul/CVE-2017-15265	–	–	–	–	–
ConVul/CVE-2017-6346	15 ± 10	24 ± 18	20 ± 19	10 ± 9	**3 ± 4**

`CS/twostage_100`:
`SURW` in ~450 schedules
vs. >5k schedules

`CS/reorder_100`:
`SURW` in ~200 schedules
vs. >> 200k schedules

`CVE-2016-1972`:
`SURW` in ~10 schedules
vs. PCT >2k schedules

RQ1 Bug Finding (2)

Answer to RQ1

SURW outperforms other sampling algorithms by a large margin in bug finding!

RQ2 Ablation Study (1)

Two ablative versions:

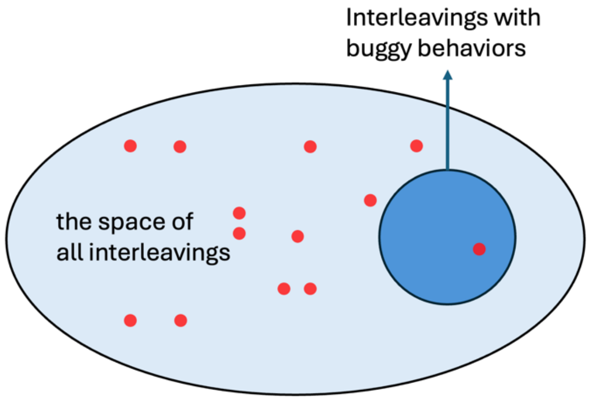
- Non-Uniform (N-U): Selective interesting subset + naïve random walk
- Non-Selective (N-S): Weighted random walk on the entire program

[Metric 1]. Average # of bugs exposed by different algorithms

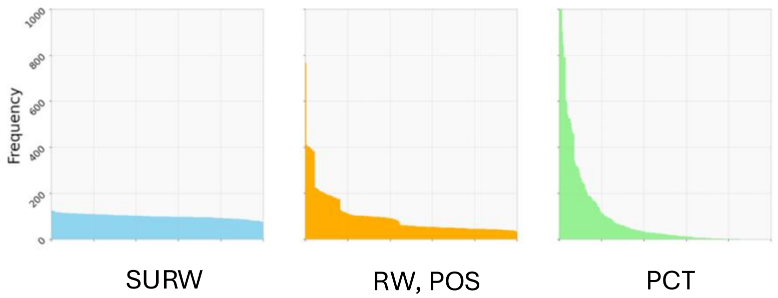
Benchmark	SURW	Non-Uniform	Non-Selective
SCTBench (max. 37)	**34.90**	29.70	30.75
		+18%	+14%

Summary

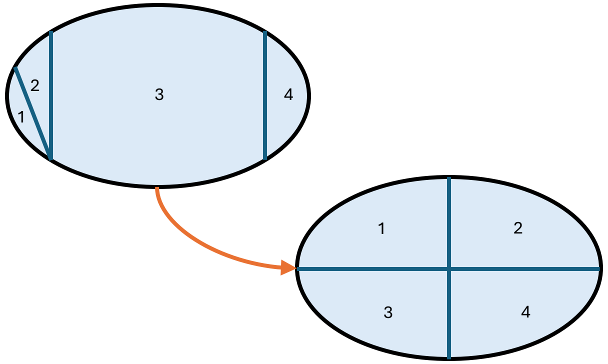
Concurrency testing as sampling



Interleaving-uniformity sampling
via weighted random walk



Effective behavioral exploration
via selective uniformity



Link to Artifact

Concurrency testing as sampling

selectively uniform

Link to Paper