Detecting and Avoiding Concurrency Bugs

Pil Jae Jang

Cyril Agbi

Paper similarities

- Testing Parallel programming is hard to debug due to interleaving bugs
- A viable solution is to better equip programmers to detect and fix these bugs
- Praise Transactional Memory

Learning from Mistakes

— A Comprehensive Study on Real World Concurrency Bug Characteristics

Motivation

Writing correct concurrent programs is difficult ! why?

a. Concurrency bug detection - imperfect
    i. Most of research: single variable, changing lock
b. Concurrent program testing and model checking
    i. Exponential interleaving
c. Concurrent programming language design
    i. TM provides programmers an easier way to specify which code regions should be atomic. But, Not perfect!

Notes

Deals with Four applications

- Not cover all applications

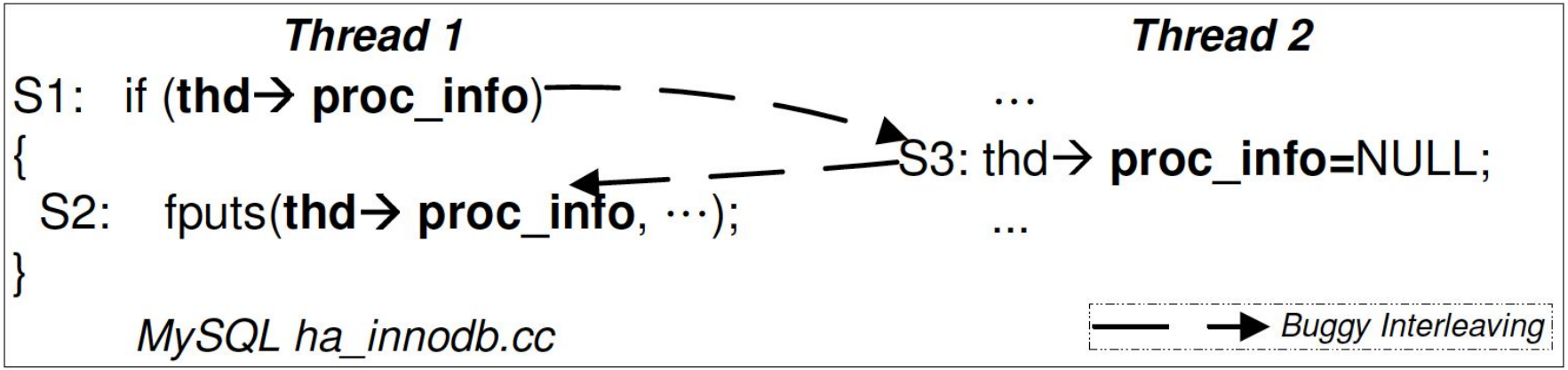
Data Race

- All of Data Races are not bug.
- Ex) benign race such as “while-flag”

Bug pattern study

Application	Total	Atomicity	Order	Other
MySQL	14	12	1	1
Apache	13	7	6	0
Mozilla	41	29	15	0
OpenOffice	6	3	2	1
Overall	74	51	24	2

Atomicity


Thread 1
Thread 2
S1: if (thd→ proc_info)
{
S2: fputs(thd→ proc_info, ⋯);
}
⋯
S3: thd→ proc_info=NULL;
...
MySQL ha_innodb.cc
Buggy Interleaving

Other - rare

Thread 1

```
void  buf_flush_try_page() {
        ...
     rw_lock(&lock);

}
```

MySQL buf0flu.c

Thread 2 ··· Thread n

```
rw_lock(&lock);
```

Monitor thread

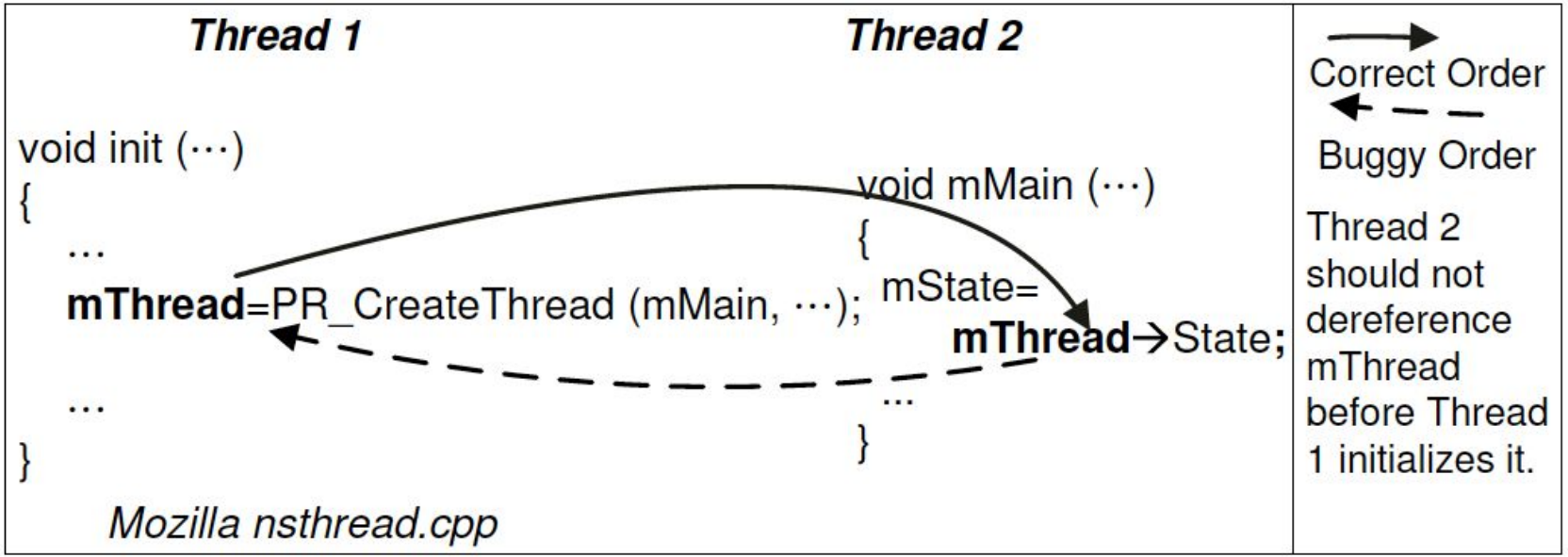
```
void  error_monitor_thread() {

  if(lock_wait_time[i] >
                    fatal_timeout)
     assert(0, "We crash the server;
        It seems to be hung.");
}
```

MySQL srv0srv.c

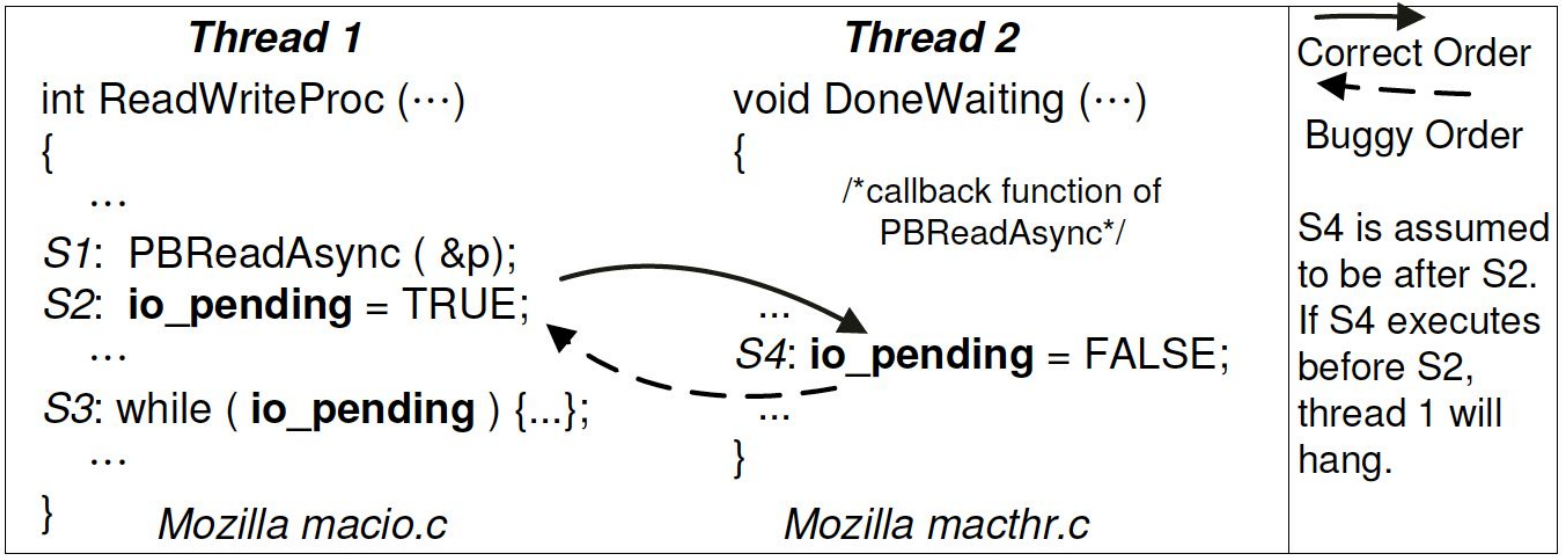
- Originally for deadlock detection
- Ideally need to set **fatal_timeout=infinite** to detect deadlock

Order



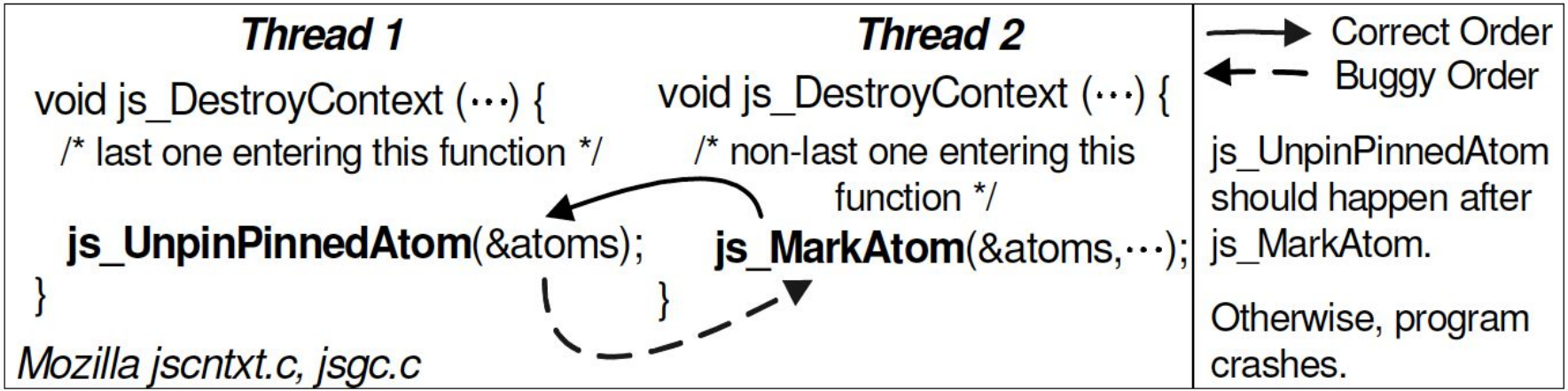
- Between Write and Read

Order



- Between Write and Write
- can hang forever

Order



- Between two groups

Bug manifestation study

How many threads are involved?

Most(101 out of 105) concurrency bugs involves only two threads.

- Increase the workload, then check pairs of threads.
- Few concurrency bugs would be missed.

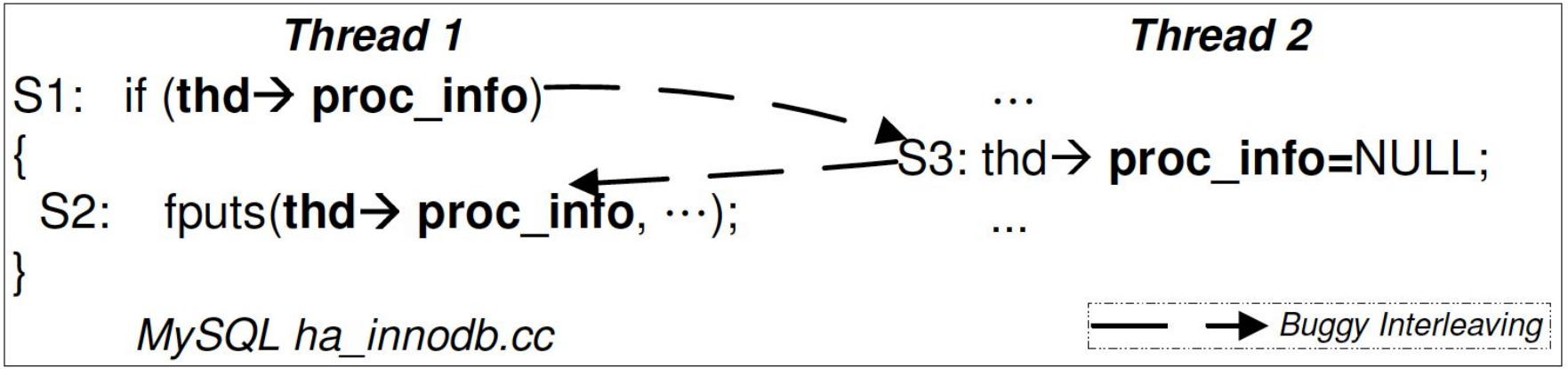
Bug manifestation study

How many variables are involved?

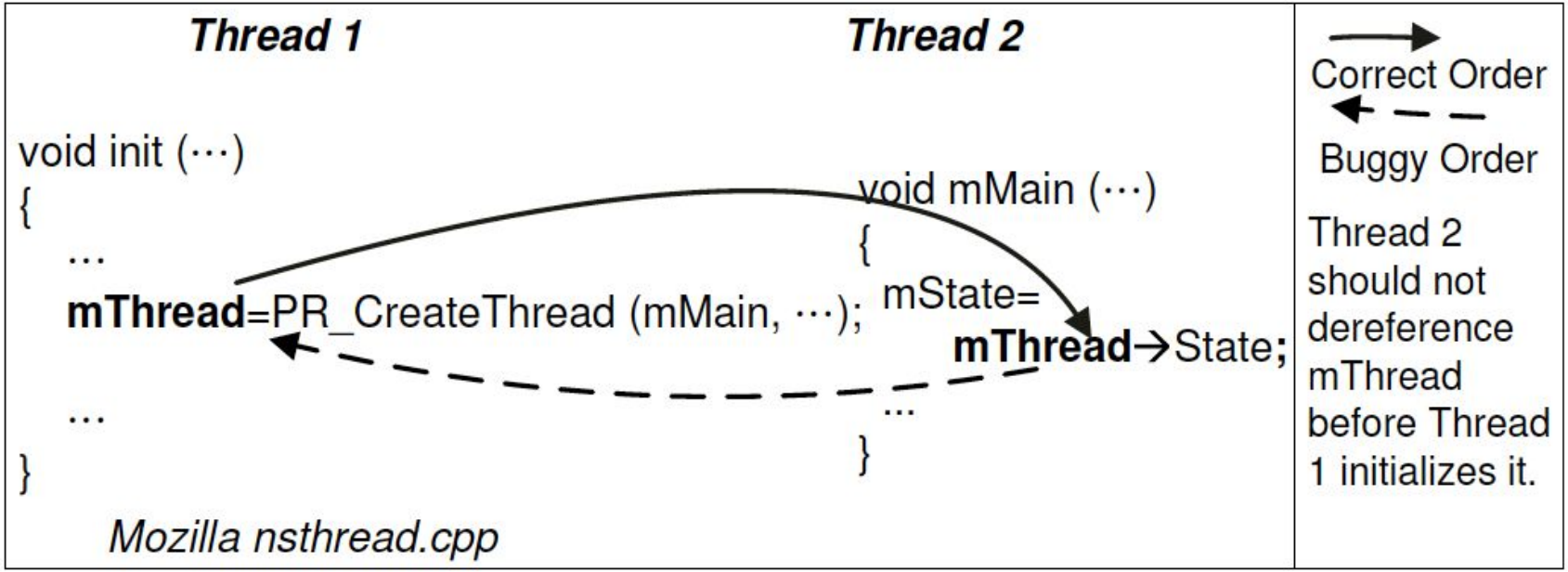
66% -> One variable

34% -> More than one variable

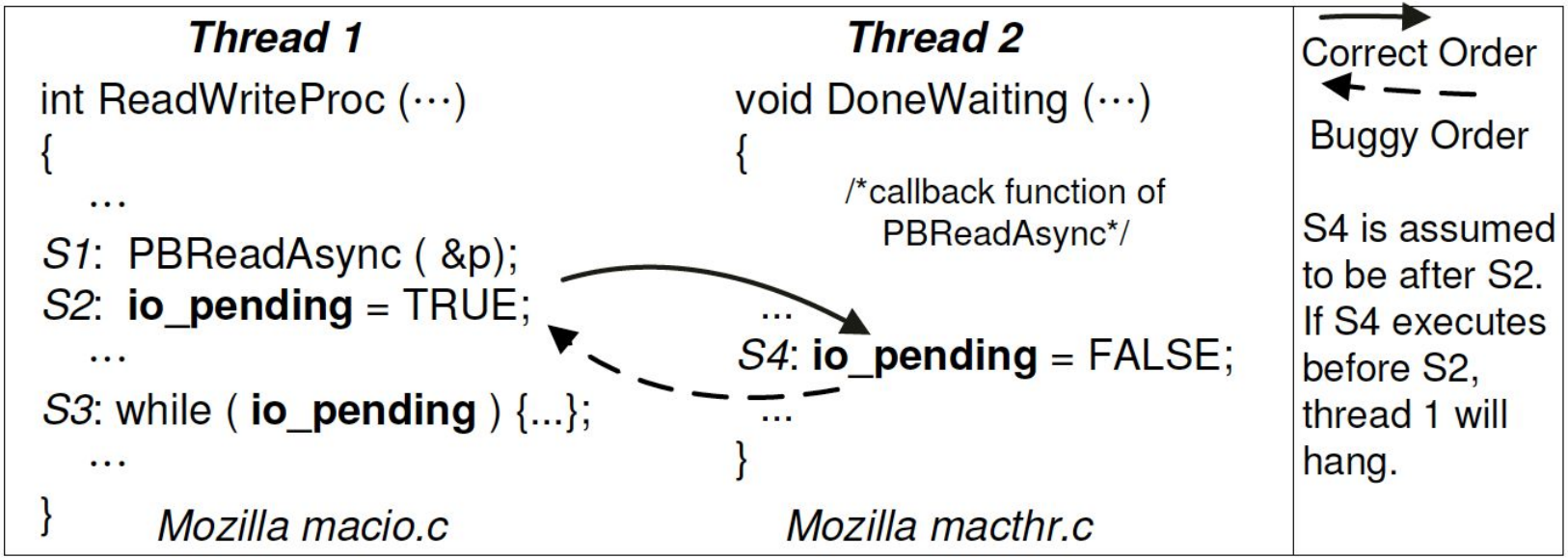
One variable



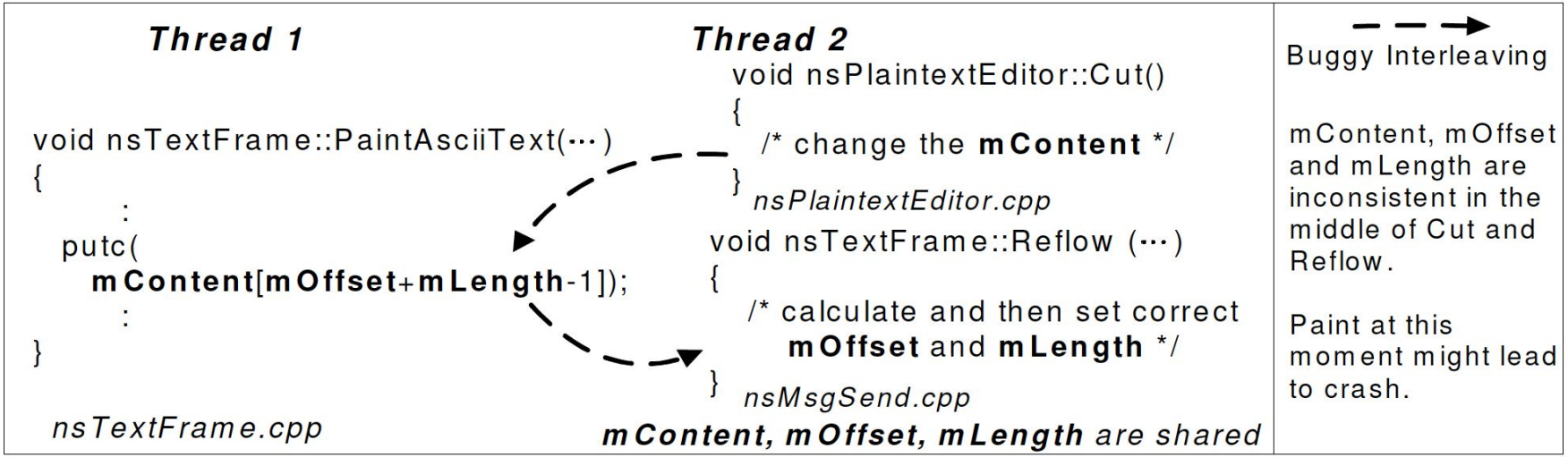
One variable

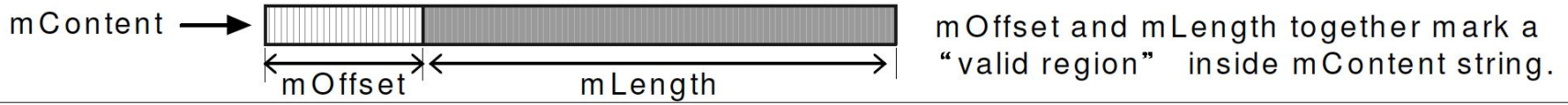


One variable



More than one variable





More than one variable

- The required condition for the bug manifestation is that thread 1 uses the three correlated variables in the middle of thread 2's modification to these three variables.
- We need new concurrency bug detection tools to address multiple variable concurrency bugs.
- Most existing bug detection tools only focus on single-variable concurrency bugs

How many accesses are involved?

Non-deadlock concurrency bugs						
Application	Total	1 acc.*	2 acc.	3 acc.	4 acc.	>4 acc.
MySQL	14	0	2	7	4	1
Apache	13	0	6	5	2	0
Mozilla	41	0	12	18	5	6
OpenOffice	6	0	2	3	1	0
Overall	74	0	22	33	12	7

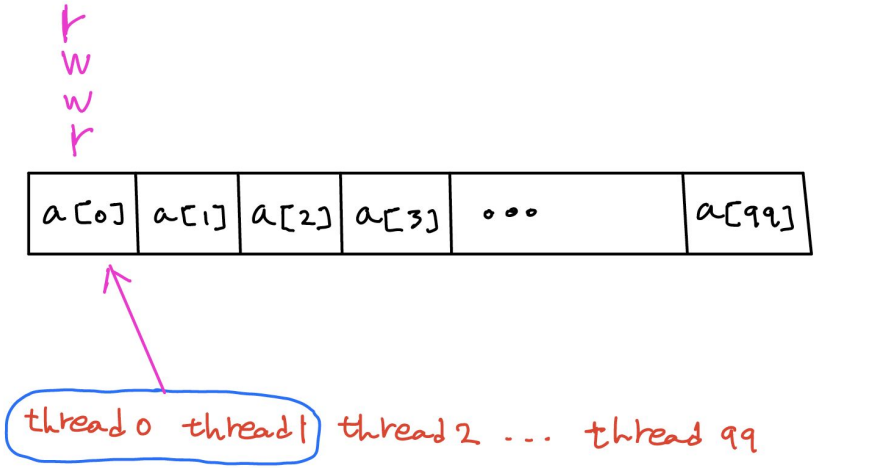
Deadlock concurrency bugs						
Application	Total	1 acc.*	2 acc.	3 acc.	4 acc.	>4 acc.
MySQL	9	4	1	4	0	0
Apache	4	0	0	4	0	0
Mozilla	16	1	2	12	0	1
OpenOffice	2	2	0	0	0	0
Overall	31	7	3	20	0	1

How many accesses are involved?

- Significant implication for concurrent program testing.
  - The challenge in concurrent program testing is that the number of all possible interleavings is **exponential to the number of dynamic memory accesses,** which is too big to thoroughly explore.

- Exploring all possible orders **within every small groups** of memory accesses, e.g. groups of 4 memory accesses.
  - The complexity of this design is only polynomial to the number of dynamic memory accesses, which is a huge reduction from the exponential-sized all-interleaving testing scheme.

Bug manifestation study Take away



Bug fix study

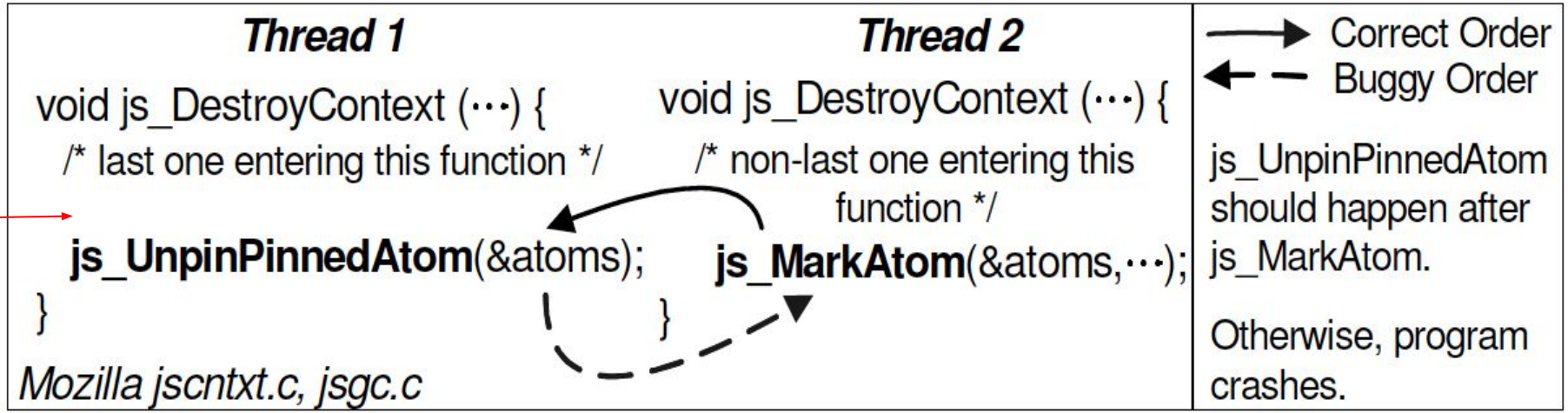
Application	Total	COND	Switch	Design	Lock	Other
MySQL	14	2	0	5	4	3
Apache	13	4	2	3	4	0
Mozilla	41	13	8	9	9	2
OpenOffice	6	0	0	2	3	1
Overall	74	19	10	19	20	6

- Adding Lock cannot enforce order intention.

Bug fix study

(1) Condition check (denoted as COND):

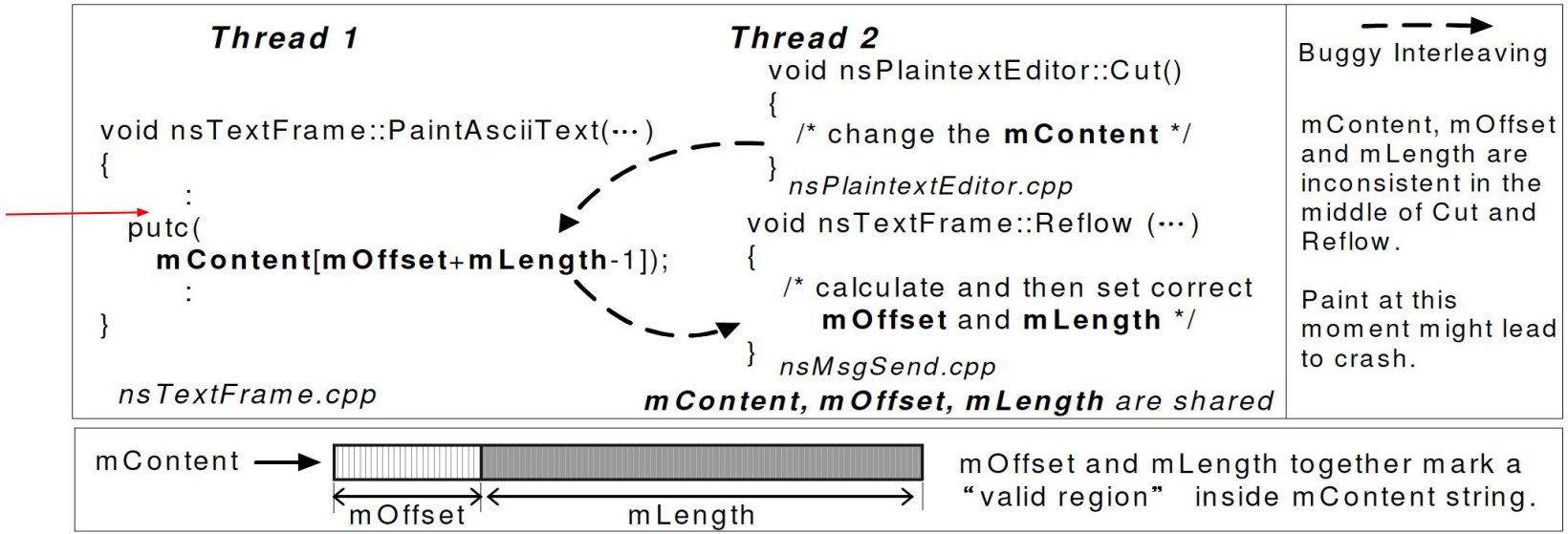
Ex) use while-flag to fix order-related bugs consistency



Bug fix study

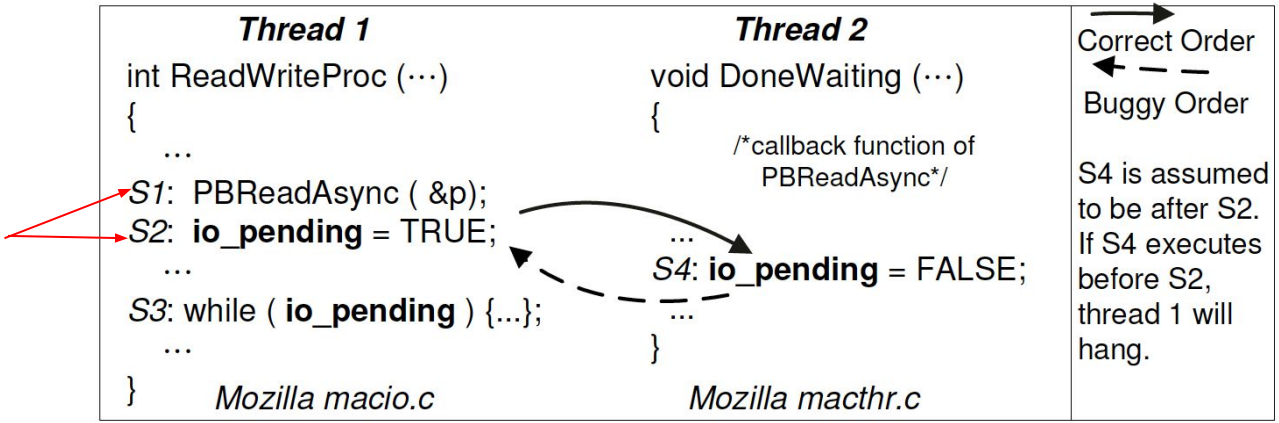
(1) Condition check (denoted as COND):

Ex) if(strlen(mContent)>= mOffset+mLength)



Bug fix study

(2) Code switch (denoted as Switch)



(3) Algorithm/Data-structure design change

ex) remove some variable from class that does not need to be shared.

Discussion: bug avoidance

Transactional memory (TM)

Application	Total	Can Help	TM might help(concerns:)			Little Help
			Long	Rollback	Nature	
MySQL	23	7	0	14	0	2
Apache	17	7	0	3	1	6
Mozilla	57	25	8	9	5	10
OpenOffice	8	2	0	4	0	2
Overall	105	41	8	30	6	20

Table 10. Can TM help avoid concurrency bugs?

Discussion: bug avoidance

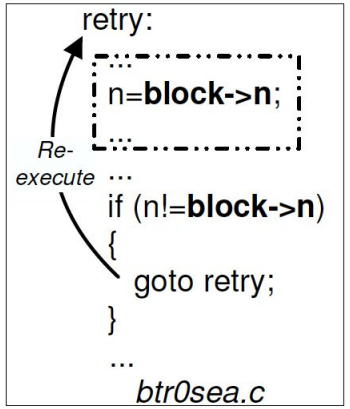


Figure 8. A MySQL bug fi

Transactional memory (TM)

- Atomicity violation bugs and deadlock bugs with relatively small and simple critical code regions can benefit the most from TM, which can help programmers clearly specify this type of atomicity
- intention.

- Figure 8 shows an example, where programmers use a consistency check with re-execution to fix the bug. Here, a transaction(with abort, rollback and replay) is exactly what programmers want.

Discussion: bug avoidance

Concern with Transactional Memory

- I/O operations:

As operations like I/O are hard to roll back, it is hard to use TM to protect the atomicity of code regions which include such operations.

- Too large memory footprint:

Mozilla bugs include the whole garbage collection process. These regions could have too large memory footprint to be effectively handled by hardware-TM

Discussion: bug avoidance

Problem with Transactional Memory

- The basic TM designs cannot help enforce the intention that "A has to be executed before B". Therefore, they cannot help avoid many related order-violation bugs

Conclusions and future work

- Design new bug detection tools to address multiple-variable bugs and order violation bugs.
- can pairwisely test concurrent program threads and focus on partial orders of small groups of memory accesses to make the best use of testing effort.
- can have better language features to support “order” semantics to further ease concurrent programming.

A Case for an Interleaving Constrained Shared-Memory Multi-Processor

Motivation

Writing parallel programs is hard because….

- INTERLEAVING
  - Verifying simple contracts is NP-complete
  - Hard to guarantee correctness
  - Hard to debug

Proposed Solution…

- Predecessor Set (PSet)
  - Constrain program to follow tested interleavings (that are good)
  - Better runtime consistency and easier to debug

Motivation - PSet

Tools that are capable of detecting:

- Data Races
  - Happens-before based vs lockset based detectors
  - Benign data races
- Atomicity Violations
  - Most tools rely on programmer to specify atomic regions
- Ordering violations

Current tools not good at detecting all three, but...

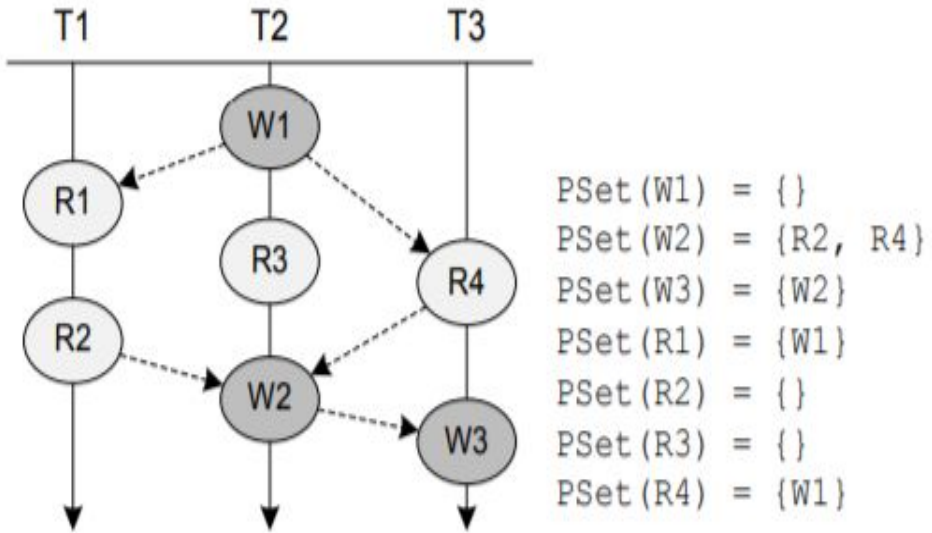
PSet is capable of detecting ALL THREE

How PSet Works

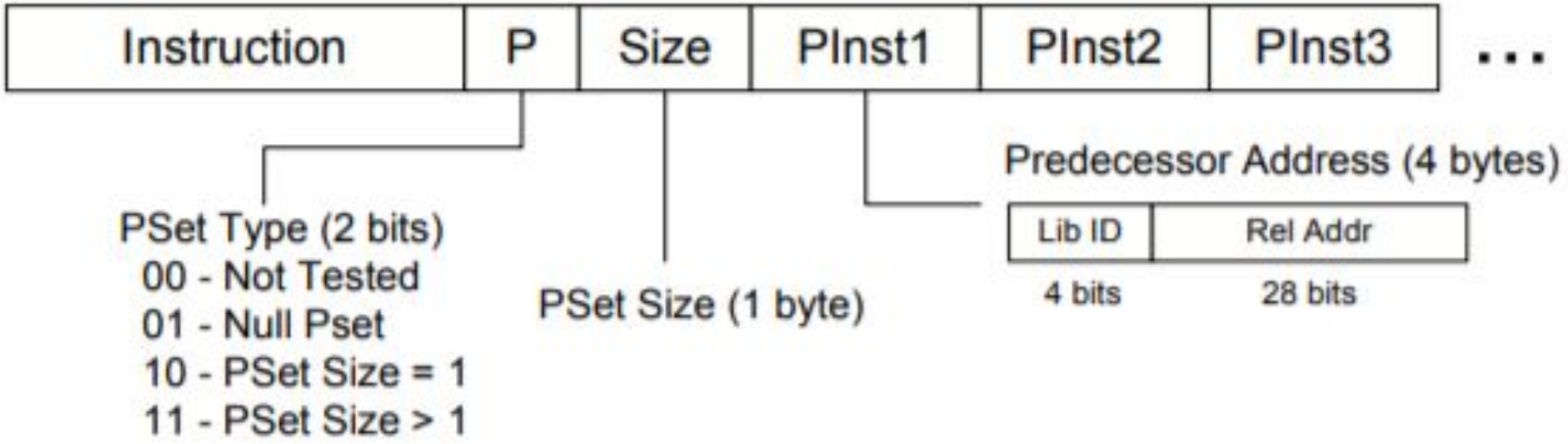
- For each RW section in a thread, a PSet contains the set of all dependencies from other threads that can occur before it
- On each RW section, checks to see if the last RW to memory location is in current section's PSet. If not…

1. STALL: The thread will stall until one of the section's predecessors completes.

2. CHECKPOINT & ROLLBACK: The program returns to a checkpoint and re-executes.



Implementation



Notes on PSet

PSets have a worse case space complexity of $O(N^2)$

- But about 95% of instructions have no PSet

Implementing reset requires a lot of additional architecture

- Add pset instructions to ISA
- Space to track last reader/writer as well as PSet constraints

The constraints need to be acquired through learning before runtime

Notes on PSet

Violation handling isn't full-proof:

- Stalling can enter a deadlock scenario
  - Solution: Time-out scheme (thread resumes after timeout)
- There is no good tested interleave path at checkpoint
  - Solution: After some number of tries, go back to further checkpoint

Design specified in paper "does not account for the interleavings between two or more memory operations accessing different memory locations."

Results

Bug #	Program	Type	Stall	Rollback	True Constraint Violations		False Constraint Violations		Rollback Window Size
					Static	Dynamic	Static	Dynamic	
1	Pbzip2	Real	Yes	Yes	1	1	3	3	0
2	Aget	Real	No	Yes	1	1	2	2	11
3	Pfscan	Injected	No	Yes	1	1	0	0	51
4	Apache	Real	No	Yes	2	20	1	1	358
5	MySQL	Real	Yes	Yes	1	7	3	6	0
6	MySQL	Extract	No	Yes	1	1	0	0	4760
7	Mozilla	Extract	No	Yes	1	1	3	3	1664
8	Mozilla	Extract	No	Yes	2	2	1	1	1224
9	Mozilla	Extract	No	Yes	1	1	0	0	1210
10	Mozilla	Extract	Yes	Yes	1	1	0	0	0
11	Mozilla	Extract	Yes	Yes	1	1	0	0	0
12	Mozilla	Extract	Yes	Yes	1	1	0	0	0
13	Mozilla	Extract	No	Yes	1	1	0	0	821
14	Mozilla	Extract	Yes	Yes	1	1	0	0	0
15	Mozilla	Extract	No	Yes	2	2	1	1	1674

Table 2: Avoiding bugs using PSet constraints. True constraint violations are related to the bug.

Results

Programs	Stall	Rollback	Cannot Resolve	Total PSet Constraint Violations	Inst. Count
pbzip2	1	5	0	6	1.3E+9
aget	0	0	0	0	1.1E+7
pfscan	1	2	0	3	7.4E+7
apache	1	4	0	5	2.8E+8
mysql	0	2	2	4	9.7E+8
fft	0	0	0	0	2.3E+8
fmm	1	0	0	1	1.6E+9
lu	0	1	0	1	1.6E+8
radix	0	0	0	0	6.4E+7
blackscholes	0	0	0	0	8.1E+8
canneal	1	0	0	1	7.0E+9

Table 3: PSet constraint violations in bug-free executions.

Results

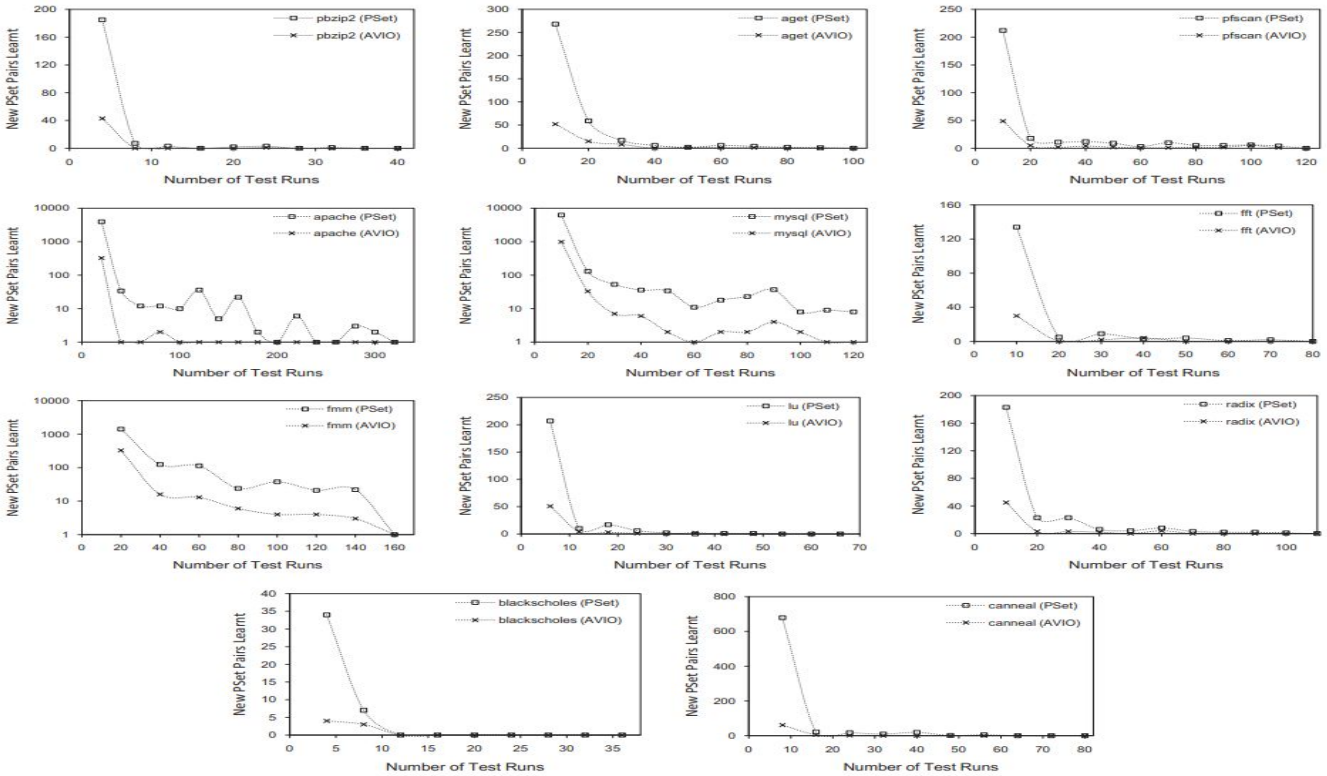


Figure 8: Number of test runs required for learning PSets and AVIO invariants.

Conclusion

This is only a first step!

- Capable of detecting more concurrency bugs than most other tools
  - Accomplishes the goal of allowing programmers to more reliably catch and fix concurrency bugs
- With sufficient testing, PSets can prevent concurrency bugs