Microsoft Research

Learning from Mistakes –
Real World Concurrency Bug Characteristics

Yuanyuan(YY) Zhou
Associate Professor
University of Illinois, Urbana-Champaign

ASPLOS 2008

Paper contributions

- Concurrency bug detection
  - What types of bugs are un-solved?
  - How bugs are diagnosed in practice?
- Concurrent program testing
  - What are the manifestation conditions for concurrency bugs?
- Concurrent program language design
  - How many mistakes can be avoided
  - What other support do we need?
- All can benefit from a closer look at real world concurrency bugs

Study methodology

- 105 real-world concurrency bugs from 4 large open
- source programs
- Study from 4 dimensions
  - Bug patterns
  - Manifestation condition
  - Diagnosing strategy
  - Fixing methods

Bug report

MySQL

Login / Register

Developer Zone | Bugs Home | Report a bug | Statistics | Advanced search | Saved searches | Tags

Bug #38816	**kill + flush tables with read lock + stored procedures causes crashes!**		
Submitted:	15 Aug 2008 7:42	Modified:	7 Aug 2009 0:35
Reporter:	Shane Bester (Platinum Quality Contributor)	Email Updates:	Subscribe
Status:	Closed	Impact on me:	None Affects Me
Category:	MySQL Server: Stored Routines	Severity:	S1 (Critical)
Version:	5.0.66a, 5.1.22, 5.1.28, 6.0.7	OS:	Any
Assigned to:	Gleb Shchepa	CPU Architecture:	Any
Tags:	flush tables with read lock, KILL		
Triage:	Triaged: D1 (Critical) / R2 (Low) / E3 (Medium)		

View | Add Comment | Files | Developer | Edit Submission | View Progress Log | Contributions

[15 Aug 2008 7:42] Shane Bester

Description:

running stored procedures at the same time as flush tables with read lock and killing connections leads to invalid reads of memory and crashes.

Sample crash:

```
mysqld-nt.exe!find_field_in_tables      Line 3889
mysqld-nt.exe!Item_field::fix_fields   Line 3873
mysqld-nt.exe!setup_fields
mysqld-nt.exe!JOIN::prepare
mysqld-nt.exe!mysql_select
mysqld-nt.exe!handle_select
mysqld-nt.exe!mysql_execute_command
mysqld-nt.exe!sp_instr_stmt::exec_core
mysqld-nt.exe!sp_lex_keeper::reset_lex_and_exec_core
mysqld-nt.exe!sp_instr_stmt::execute
mysqld-nt.exe!sp_head::execute
mysqld-nt.exe!sp_head::execute_procedure
mysqld-nt.exe!mysql_execute_command
mysqld-nt.exe!mysql_parse
mysqld-nt.exe!dispatch_command
mysqld-nt.exe!do_command
mysqld-nt.exe!handle_one_connection
```

See attachment for a variety of valgrind errors found during this test.

How to repeat:

compile and run the attached .c testcase.

Applications

	MySQL	Apache	Mozilla	OpenOffice
Software Type	Server	Server	Client	GUI
Language	C++/C	Mainly C	C++	C++
LOC (M line)	2	0.3	4	6
Bug DB history	6 years	7 years	10 years	8 years

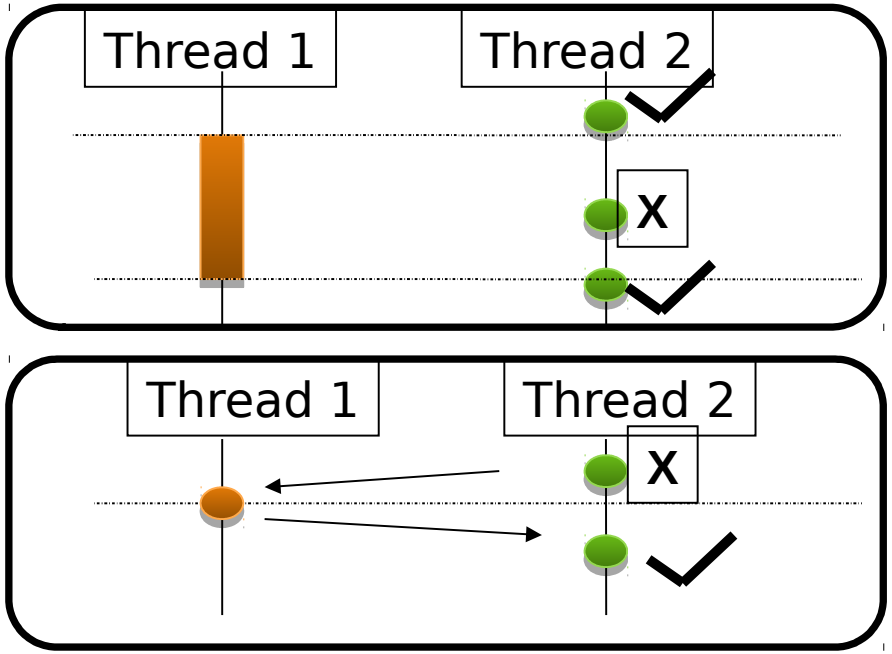
Bugs analyzed

	MySQL	Apache	Mozilla	OpenOffice	Total
Non-deadlock	14	13	41	6	74
Deadlock	9	4	16	2	31

- Limitations
  - No scientific computing applications
  - No JAVA programs
  - Never-enough bug samples

Bug patterns

- Classified based on root causes
- Categories
  - Atomicity violation
    - The desired atomicity of certain
      - code region is violated
  - Order violation
    - The desired order between
      - two (sets of) accesses is flipped
  - Others



Bug patterns: atomicity bug

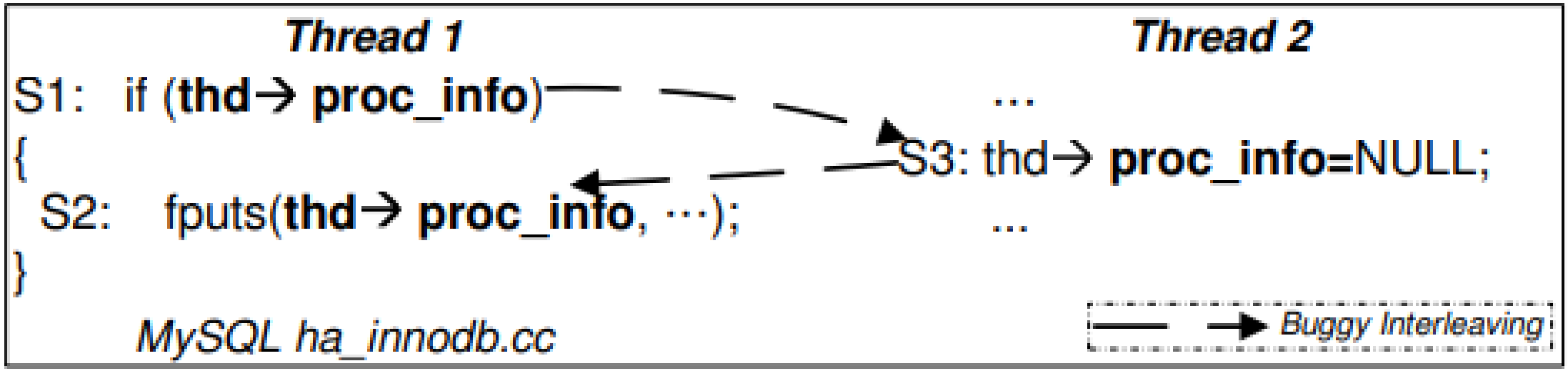


Figure 1. An atomicity violation bug from MySQL.

Bug patterns: order violation bug

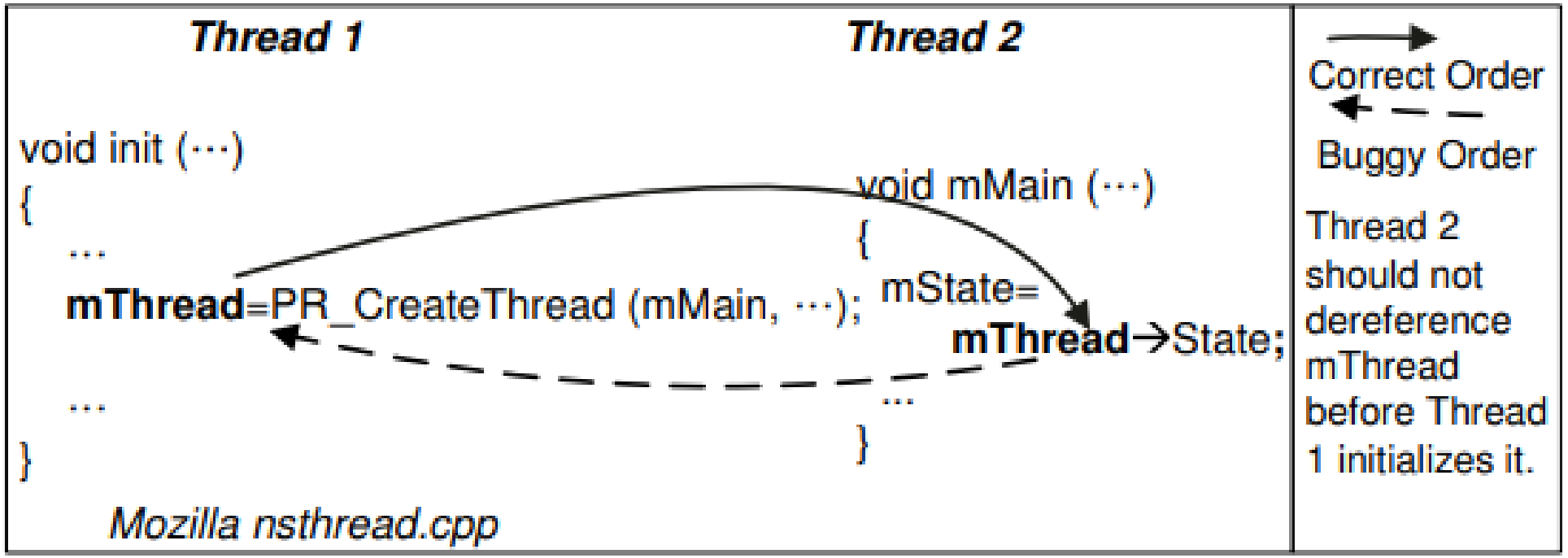
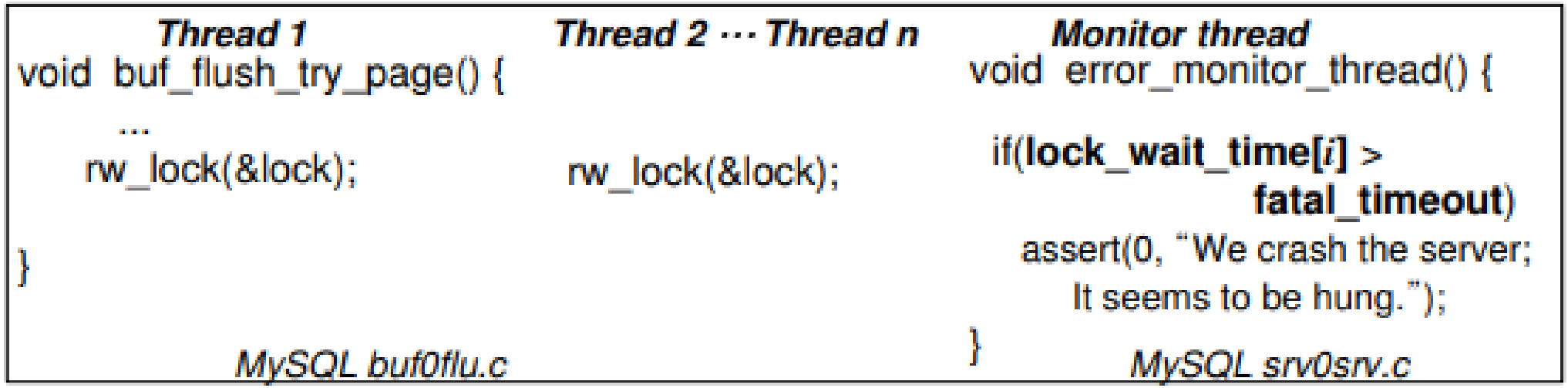


Figure 2. An order violation bug from Mozilla. The program fails to enforce the programmer's order intention: thread 2 should not read `mThread` until thread 1 initializes `mThread`. Note that, this bug could be fixed by making `PR_CreateThread` atomic with the write to `mThread`. However, our bug pattern categorization is based on root cause, regardless of possible fix strategies.

Bug patterns: underestimating #threads



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

Figure 3. A MySQL bug that is neither an atomicity-violation bug nor an order-violation bug. The monitor thread is designed to detect deadlock. It restarts the server when a thread i has waited for a lock for more than `fatal_timeout` amount of time. In this bug, programmers under-estimate the workload (`n` could be very large), and therefore the lock waiting time would frequently exceed `fatal_timeout` and crash the server. (We simplified the code for illustration)

Bug patterns: order violation bug

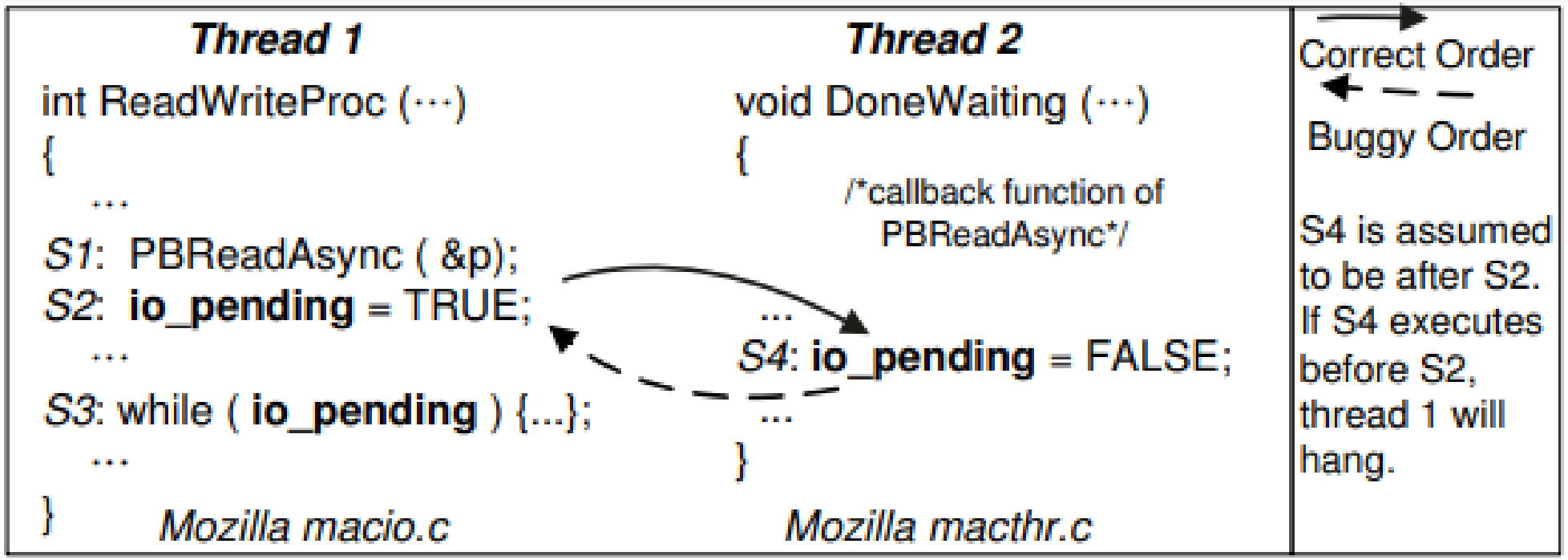


Figure 4. A write-write order violation bug from Mozilla. The program fails to enforce the programmer's order intention: thread 2 is expected to write `io_pending` to be FALSE some time after thread 1 initializes it to be TRUE. Note that, this bug could be fixed by making S1 and S2 atomic. However, our bug pattern categorization is based on root cause, regardless of possible fix strategies.

Bug patterns: order violation bug

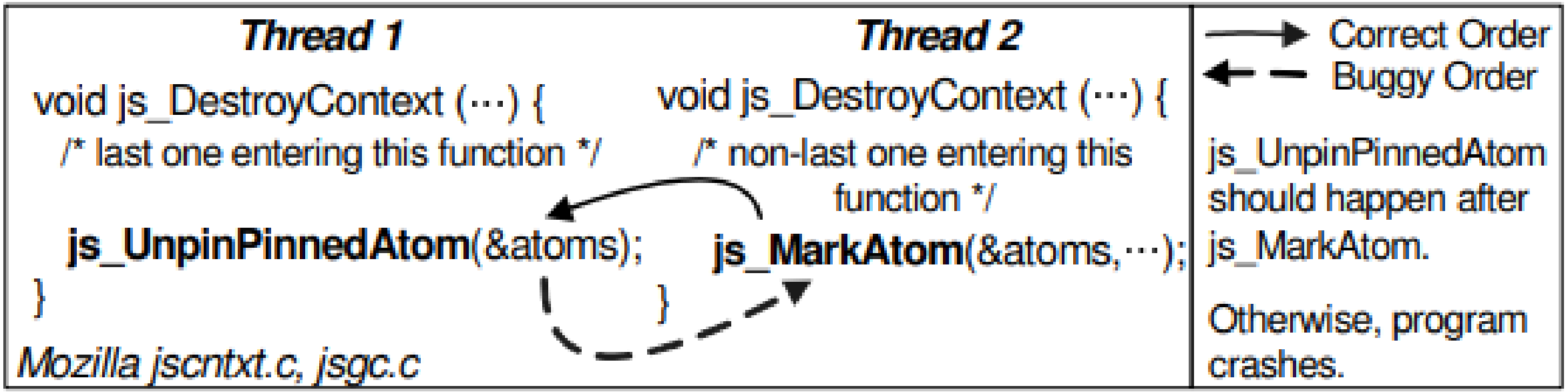
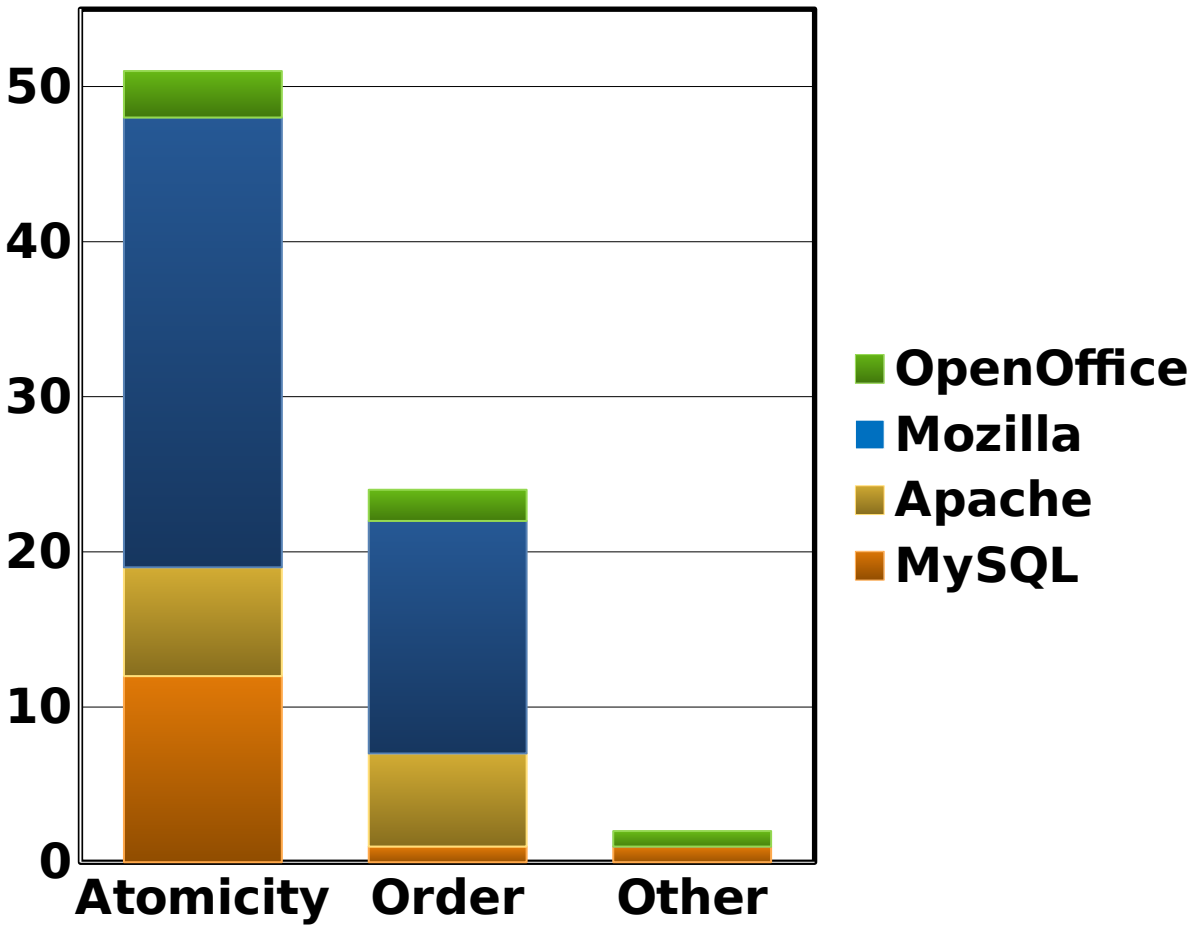


Figure 5. A Mozilla bug that violates the intended order between two groups of operations.

Bug pattern presence in analyzed softwares



Implications

- We should focus on atomicity violation and order violation
- Bug detection tools for order violation bugs are desired

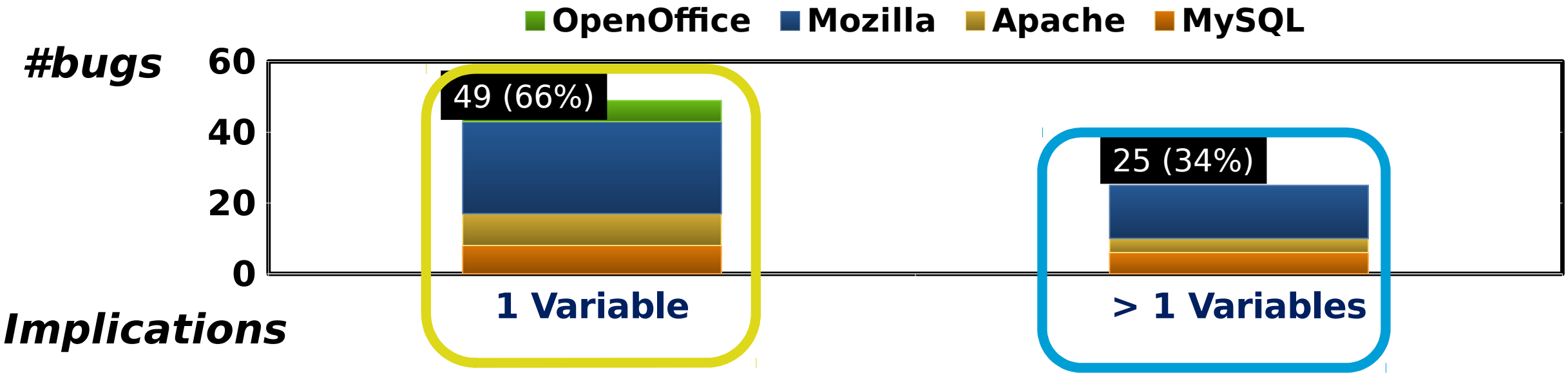
**There are 3-bug overlap between Atomicity and Order*

Bug manifestation study

- Bug manifestation condition
  - A specific execution order among a smallest set of memory accesses
  - The bug is guaranteed to manifest, as long as the condition is satisfied

- How many threads are involved?
- How many variables are involved?
- How many accesses are involved?

Bug manifestation: how many variables?



Implications

- Single variables are more common
  - The widely-used simplification is reasonable
- Multi-variable concurrency bugs are non-negligible
  - Techniques to detect multi-variable concurrency bugs are needed

Bug patterns: multi-variable

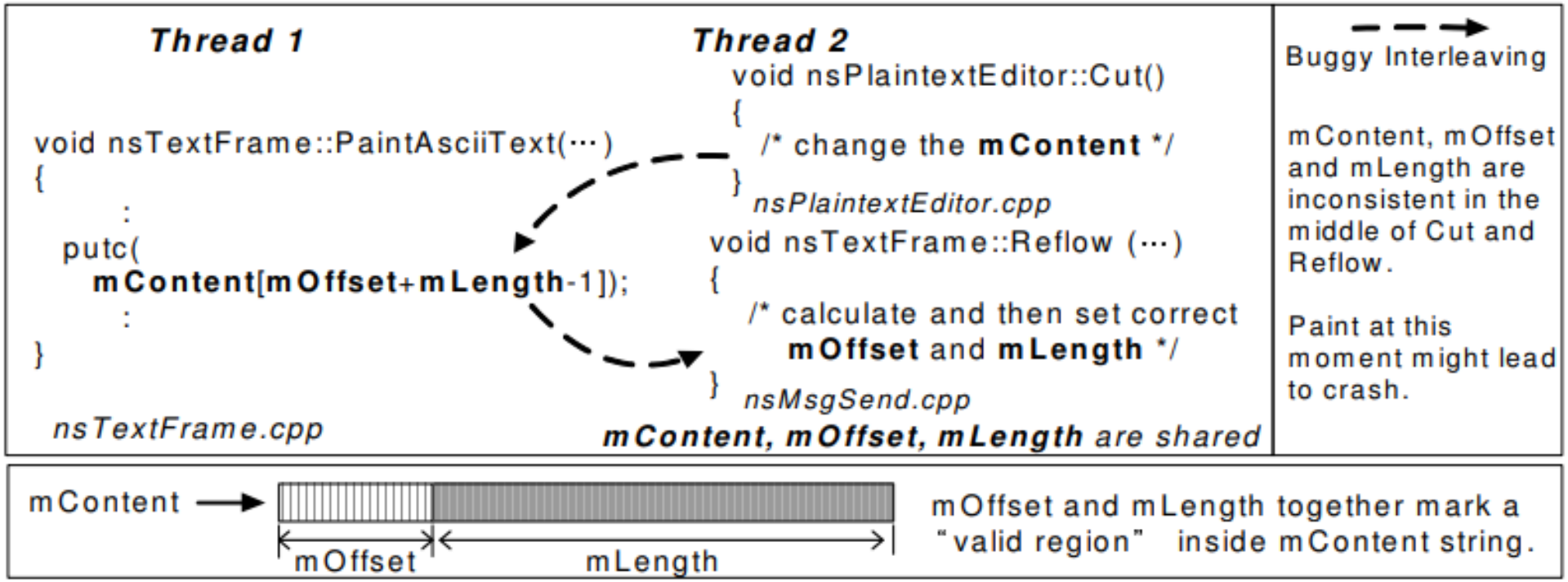
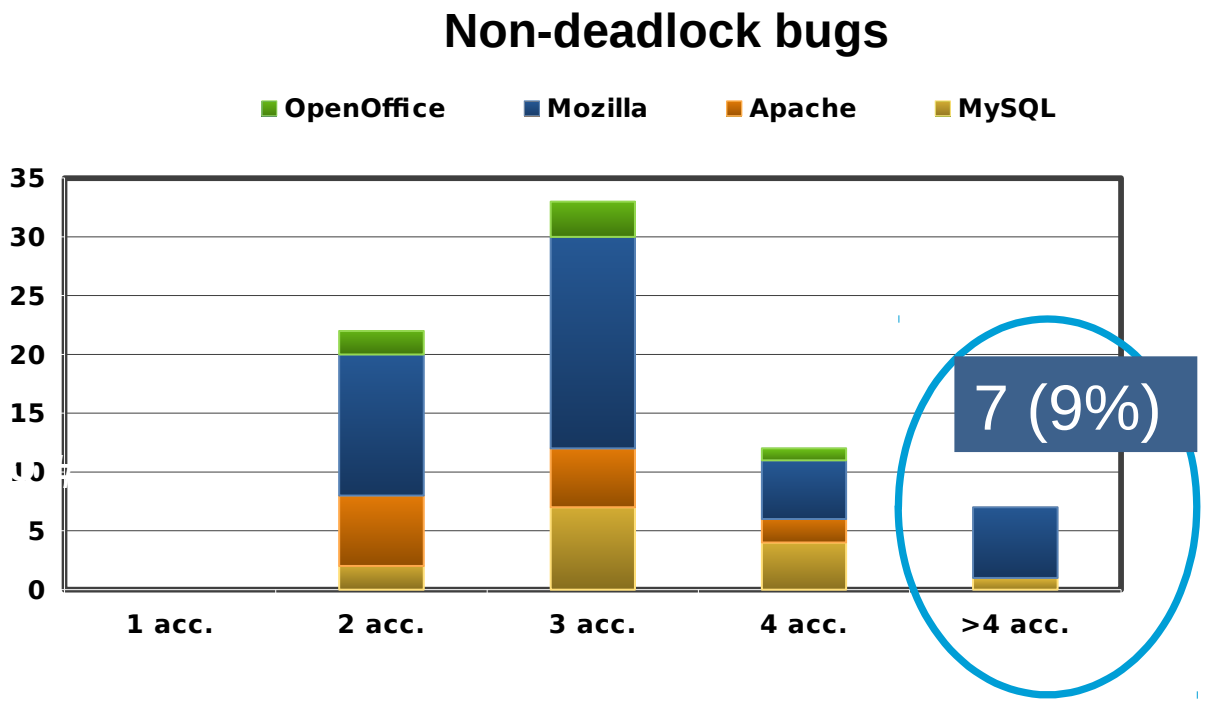
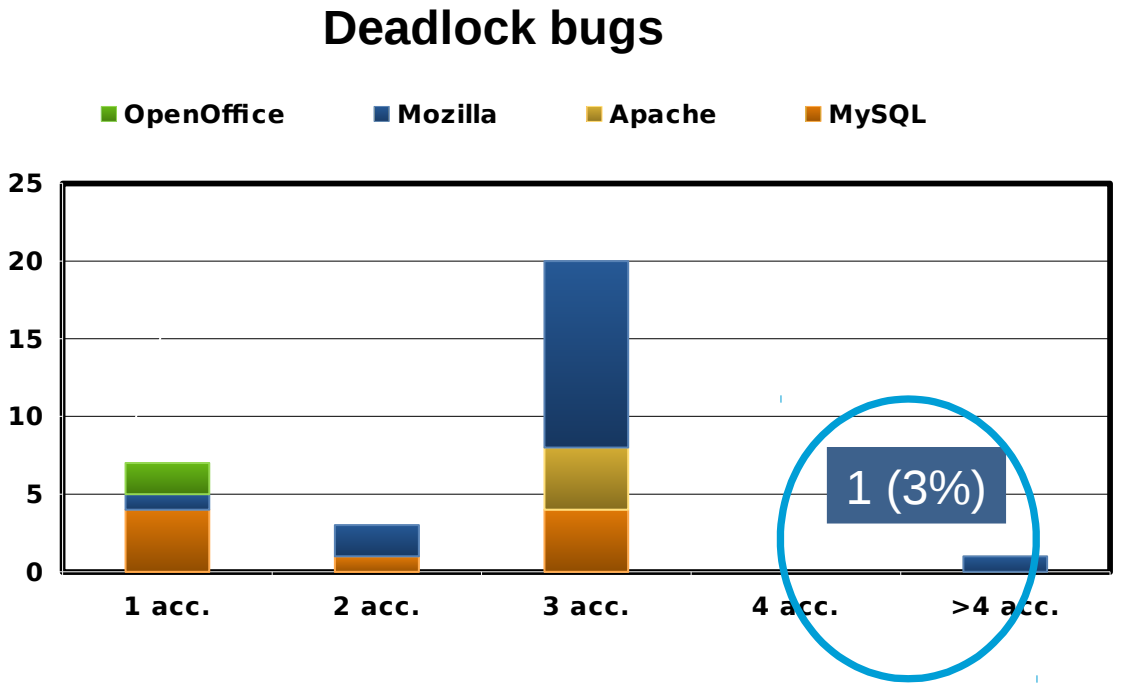


Figure 6. A multi-variable concurrency bug from Mozilla. Accesses to three correlated variables, `mContent`, `mOffset` and `mLength`, should be synchronized.

Bug manifestation: how many accesses?



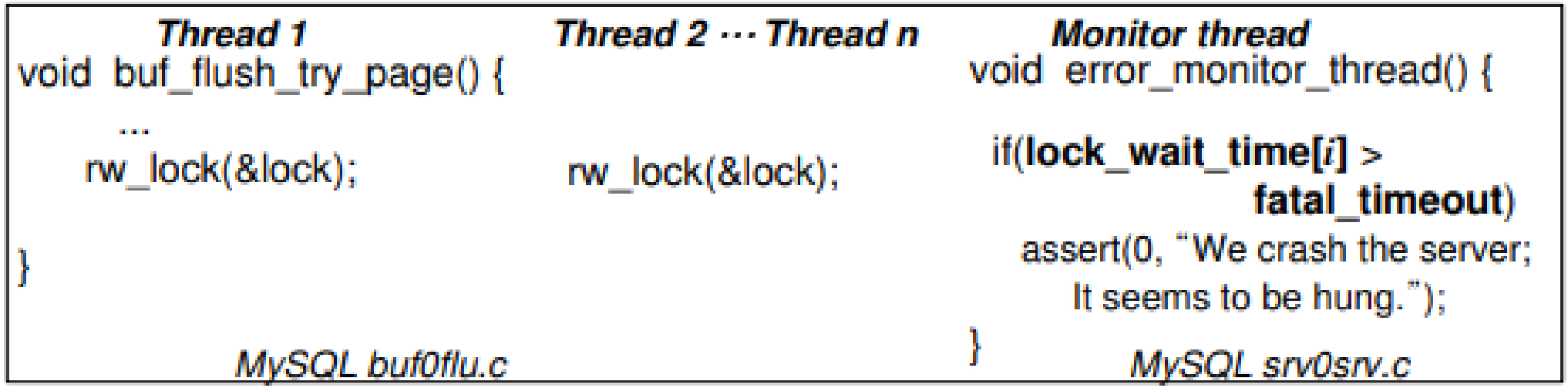


- Concurrent program testing can focus on small groups of accesses
  - The testing target shrinks from exponential to polynomial

Bug manifestation: how many threads?

- 101 out of 105 (96%) bugs involve at most two threads
  - Most bugs can be reliably disclosed if we check all possible interleaving between each pair of threads
- Few bugs cannot
  - Example: Intensive resource competition among many threads causes unexpected delay

Bug patterns: underestimating #threads



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

Figure 3. A MySQL bug that is neither an atomicity-violation bug nor an order-violation bug. The monitor thread is designed to detect deadlock. It restarts the server when a thread i has waited for a lock for more than `fatal_timeout` amount of time. In this bug, programmers under-estimate the workload (`n` could be very large), and therefore the lock waiting time would frequently exceed `fatal_timeout` and crash the server. (We simplified the code for illustration)

Bug fix strategies: non-deadlock bugs

- Adding/changing locks 20 (27%)
- Condition check 19 (26%)20 (27%)
- Data-structure change 19 (26%)
- Code switch 10 (13%)
- Other 6 (8%)

Implications

No silver bullet for fixing concurrency bugs.
Lock usage information is not enough to fix bugs.

Bug fixes: lock vs. condition variable

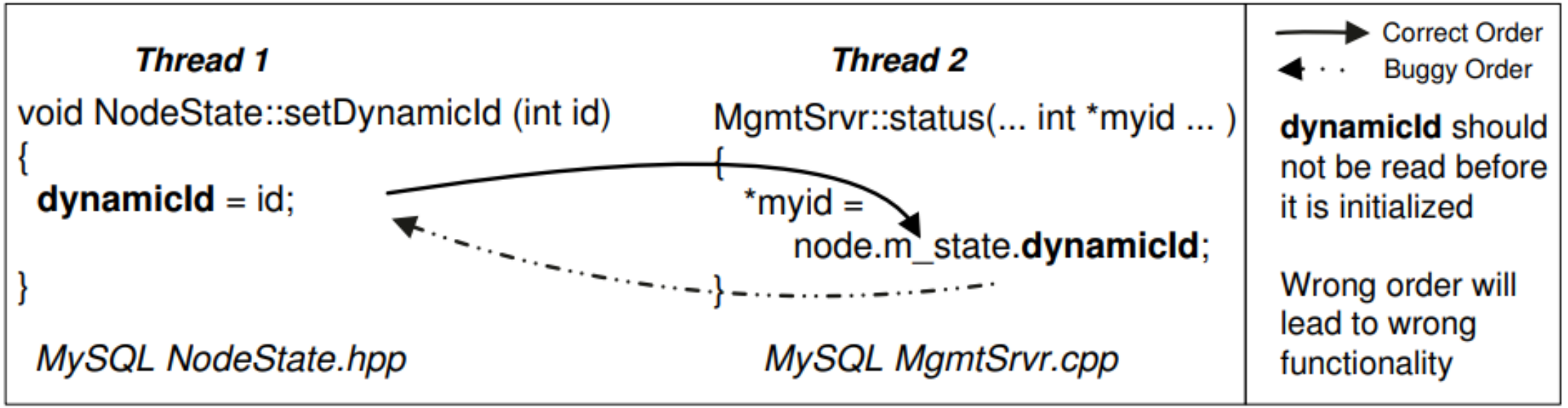
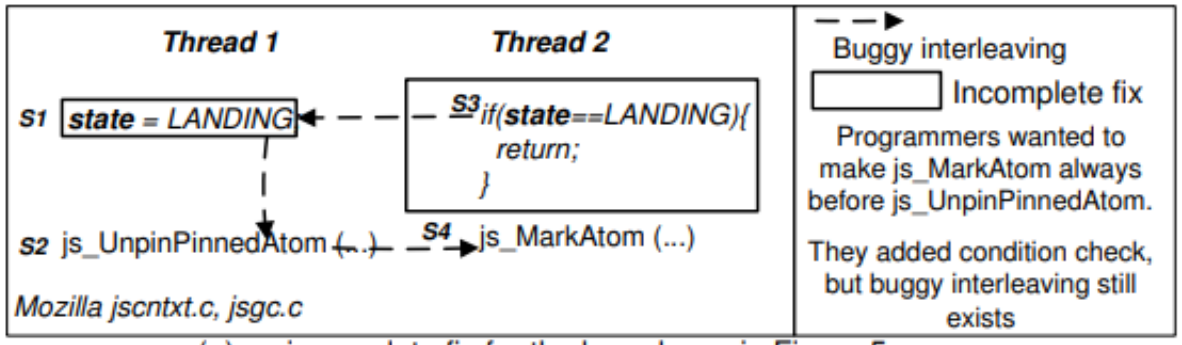
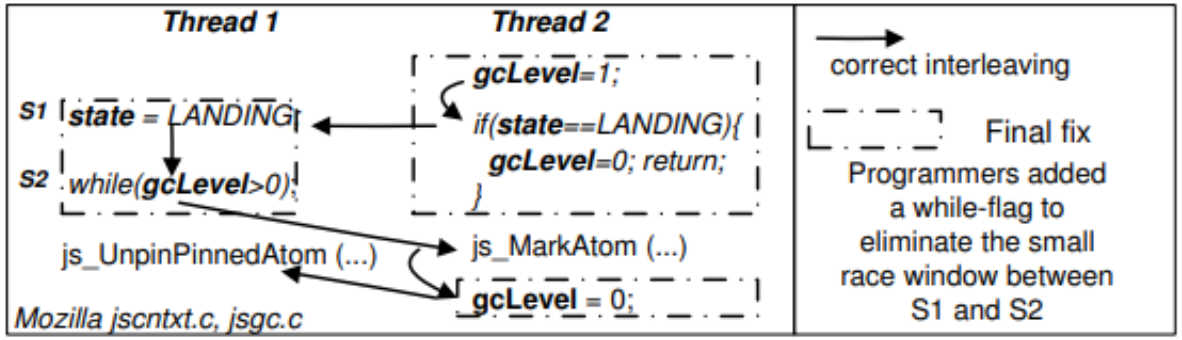


Figure 7. A MySQL bug that cannot be fixed by adding/changing locks.

Bug fixes: two condition variables



(a) an incomplete fix for the bug shown in Figure 5.
This fix left a small window between S1 and S2 unprotected.



(b) a final correct fix.
Now the order between js_MarkAtom and js_UnpinPinnedAtom is enforced.

Figure 9. The process of fixing the bug shown in Figure 5. Programmers want to make sure `js_MarkAtom` will not be called after `js_UnpinPinnedAtom`. They first added a flag variable `state` to fix the bug. However, that left a small window between S1 and S2 unprotected. They finally added a second flag variable `gcLevel` to completely fix the bug.

Bug fix strategies: non-deadlock bugs

- Adding/changing locks 20 (27%)
- Condition check 19 (26%)20 (27%)
- Data-structure change 19 (26%)
- Code switch 10 (13%)
- Other 6 (8%)

Implications

No silver bullet for fixing concurrency bugs.
Lock usage information is not enough to fix bugs.

Bug fixes: code switch

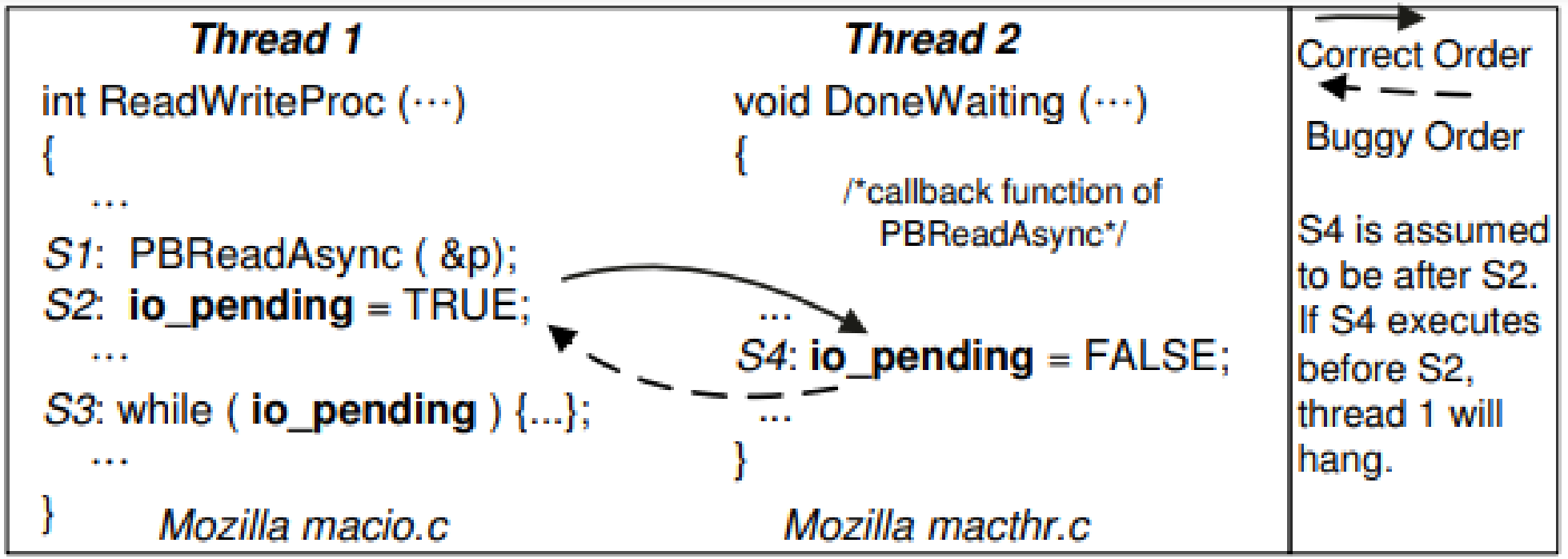


Figure 4. A write-write order violation bug from Mozilla. The program fails to enforce the programmer's order intention: thread 2 is expected to write `io_pending` to be FALSE some time after thread 1 initializes it to be TRUE. Note that, this bug could be fixed by making S1 and S2 atomic. However, our bug pattern categorization is based on root cause, regardless of possible fix strategies.

Bug fix strategies: deadlock bugs

- Give up resource acquisition 19 (61%)
- Change resource acquisition order 7 (23%)
- Split the resource to smaller ones 1 (3%)
- Others 4 (13%

Might introduce non-deadlock bugs

We need to pay attention to the correctness of ``fixed'' deadlock bugs

Summary

- Impact of concurrency bugs
  - ~ 70% leads to program crash or hang
- Reproducing bugs are critical to diagnosis
  - Many examples
- Programmers lack diagnosis tools
  - No automated diagnosis tools mentioned
  - Most are diagnosed via code review
  - Reproduce bugs are extremely hard and directly determines the diagnosing time
- 60% 1st-time patches still contain concurrency bugs (old or new)
- Usefulness and concerns of transactional memory

Seriousness of concurrency bugs

Therac-25

From Wikipedia, the free encyclopedia

The **Therac-25** was a computer-controlled radiation therapy machine produced by Atomic Energy of Canada Limited (AECL) in 1982 after the Therac-6 and Therac-20 units (the earlier units had been produced in partnership with CGR of France).

It was involved in at least six accidents between 1985 and 1987, in which patients were given massive overdoses of radiation.[1]:425 Because of concurrent programming errors, it sometimes gave its patients radiation doses that were hundreds of times greater than normal, resulting in death or serious injury.[2] These accidents highlighted the dangers of software control of safety-critical systems, and they have become a standard case study in health informatics and software engineering. Additionally the overconfidence of the engineers[1]:428 and lack of proper due diligence to resolve reported software bugs are highlighted as an extreme case where the engineers' overconfidence in their initial work and failure to believe the end users' claims caused drastic repercussions.

Computer failure [edit]

A software bug known as a race condition existed in General Electric Energy's Unix-based XA/21 energy management system. Once triggered, the bug stalled FirstEnergy's control room alarm system for over an hour. System operators were unaware of the malfunction. The failure deprived them of both audio and visual alerts for important changes in system state.[9][10]

Unprocessed events queued up after the alarm system failure and the primary server failed within 30 minutes. Then all applications (including the stalled alarm system) were automatically transferred to the backup server, which itself failed at 14:54. The server failures slowed the screen refresh rate of the operators' computer consoles from 1–3 seconds to 59 seconds per screen. The lack of alarms led operators to dismiss a call from American Electric Power about the tripping and reclosure of a 345 kV shared line in northeast Ohio. But by 15:42, after the control room itself lost power, control room operators informed technical support (who were already troubleshooting the issue) of the alarm system problem.[11]