Understanding and Detecting Real-World Performance Bugs

Gouliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu

Presented by Cindy Rubio-González

Feb 10th, 2015

Motivation

- Performance bugs pervasively degrade software performance
- Performance bugs lead to reduced throughput, increased latency, and wasted resources
- Performance bugs exist widely in released software
- Performance bugs are costly to diagnose

Contributions

- Characteristics Study
  - Conducts comprehensive study of 109 real-world performance bugs
  - Finds how performance bugs are introduced, exposed, and fixed
- Bug Detection
  - Identifies useful efficiency rules
  - Implements 25 rule checkers
  - Finds 332 potential performance bugs in real-world applications

Characteristics Study

Benchmarks

Application Description	Language(s)	# Bugs
Apache Suite HTTPD: Web Server TomCat: Web App Server Ant: Build Management Utility	 C Java Java	25
Chronium Suite Google Chrome Browser	 C/C++	10
GCC Suite gcc and g++ Compilers	 C/C++	10
Mozilla Suite Firefox: Web Browser Thunderbird: Email Client	 C++, JavaScript C++ JavaScript	36
MySQL Suite Server: Database Server Connector: DB Client Libraries	 C/C++ C/C++, Java, .Net	28
Total		109

Bug Selection

- GCC, Mozilla, and MySQL
  - Developers use special tags for performance bugs
    - *compile-time-hog, perf, and S5*
- Apache and Chrome
  - Used performance related keywords
    - Slow, performance, latency, throughput, etc.
- Randomly sampled 109 fixed bugs
  - 27 reported before 2004
  - 38 reported between 2004 and 2007
  - 44 reported after 2008

Example 1: Transparent Draw

Mozilla Bug #66461

- A Mozilla bug drawing transparent images
- Useless computation when image is transparent
- Transparent images were not expected to be widely used for layout purposes

```
nsImage::Draw(…) {    …
   + if(mIsTransparent) return;
   …
   // render the input image
}
```

nsImageGTK.cpp

Patch conditionally skips Draw

Example 2: Intensive GC

Mozilla Bug #515287

- A Mozilla bug performing intensive GCs
- Firefox 10x slower than Safari at idle GMail pages
- Bug observed until Web 2.0

```
XMLHttpRequest::OnStop() {
   // at the end of each XHR
   …
   - mScriptContext->GC();
}
```

nsXMLHttpRequest.cpp

Patch removes call to GC

Example 3: Bookmark All

Mozilla Bug #490742

- A Mozilla bug with un-batched DB operations
- Firefox hangs when number of tabs >= 20
- One of the first few batchable database tasks in Firefox

```
for (i = 0; i < tabs.length; i++) {
   …
   - tabs[i].doTransact();
}
+ doAggregateTransact(tabs);
```

nsPlacesTransactionsService.js

Patch adds a new API to aggregate DB transactions

Example 4: Slow Fast-Lock

MySQL Bug #38941

- A MySQL bug with over synchronization
- `fastmutex_lock` calls `random`, which contains a lock
- Users reported a 40x slowdown when using the fast lock

```
int fastmutex_lock (fmutex_t *mp) {
   …
   - maxdelay += (double)random();
   + maxdelay += (double)park_rng();
   …
}
```

thr_mutex.c

Patch replaces `random` with non-synchronized number generator

How are Performance Bugs *introduced*?

- Workload Mismatch
  - Input paradigm could shift later on
  - Workload much diverse and complex

} 41

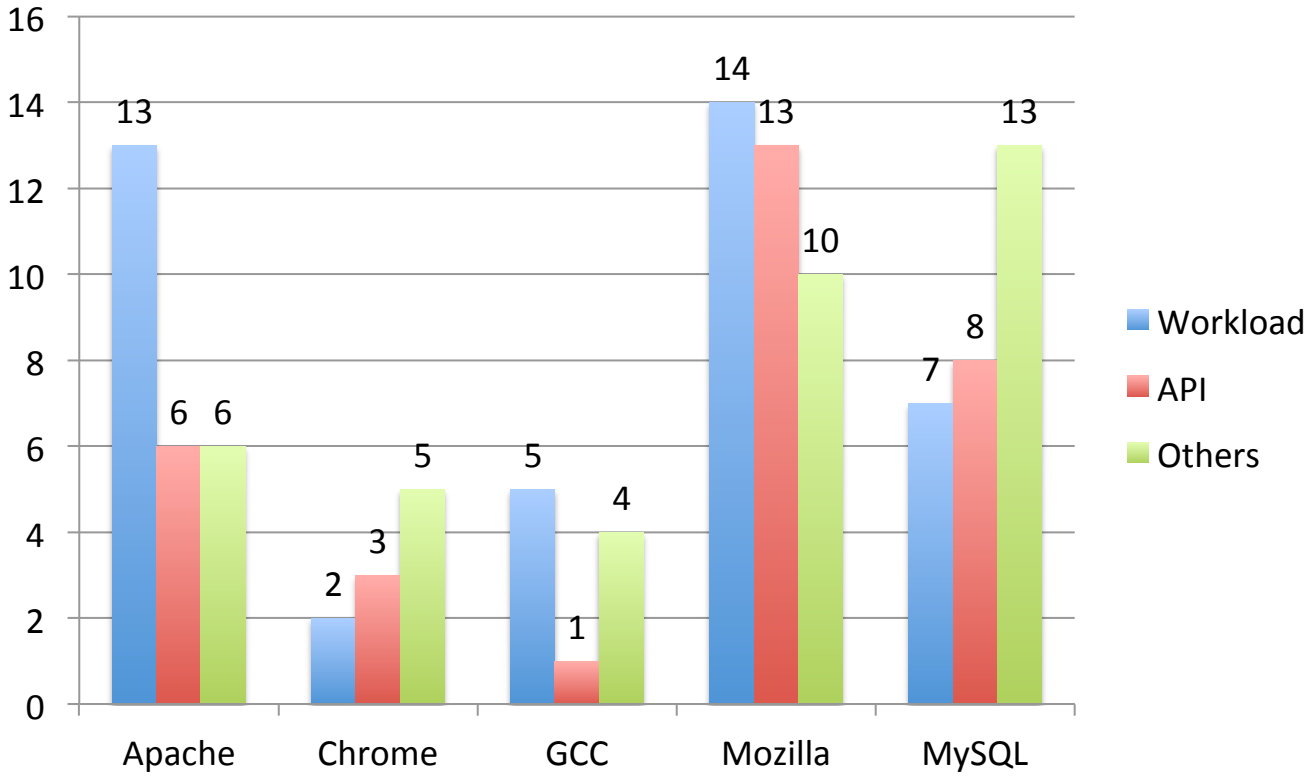
- API Misunderstanding
  - Sensitivity with respect to parameter values
  - Code encapsulation hides details

} 31

- Others
  - Example: Side-effect of functional bugs
    - Failing to reset a busy flag

} 38

How are Performance Bugs *introduced*?



Workload: 41
API: 31
Others: 38

How are Performance Bugs *exposed*?

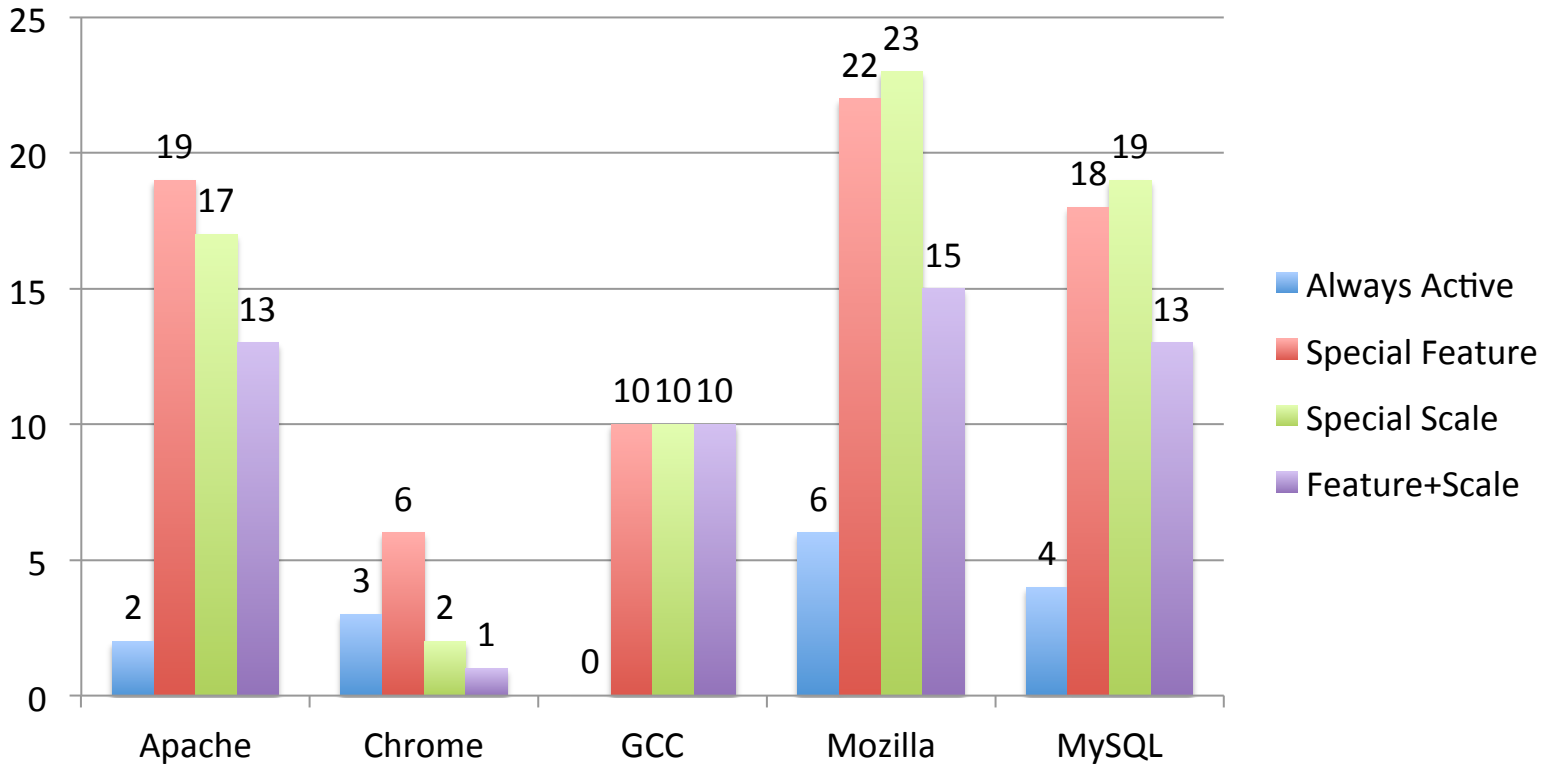
- Always Active Bugs
  - Happen during start-up, shut down, and in heavily executed code
  - In general, affect every single run

} 15

- Input Feature and Scale Conditions
  - Require specialized inputs
  - Require large-scale inputs

} 198

How are Performance Bugs *exposed*?



Always Active: 15
Special Feature: 75
Special Scale: 71
Feature+Scale: 52

How are Performance Bugs *fixed*?

- Change Call Sequence
  - Examples: Bookmark All and Intensive GC

47

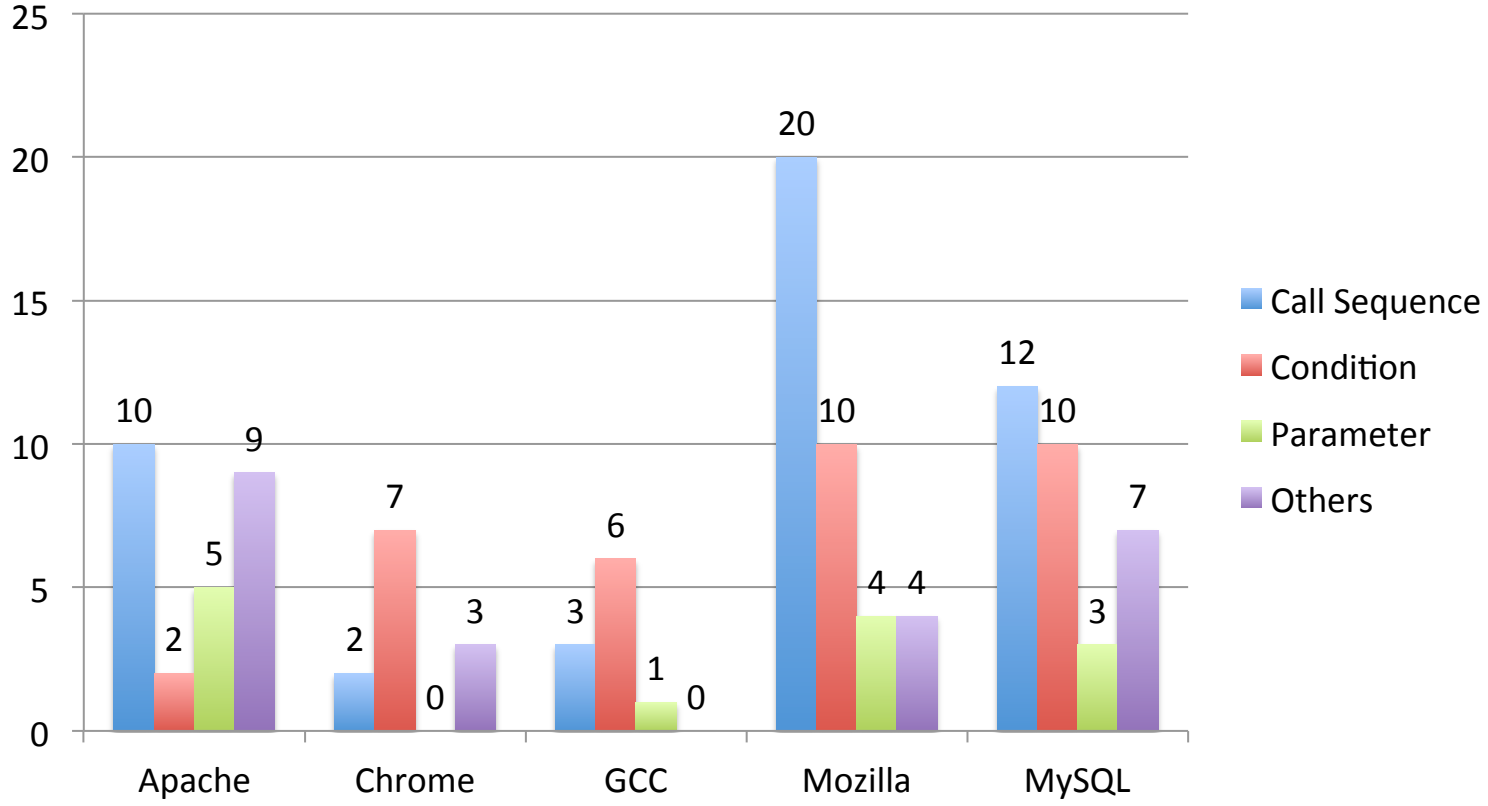
- Change Condition
  - Example: Transparent Draw

35

- Adjust Function Parameters

13

How are Performance Bugs *fixed*?



Call Sequence: 47
Condition: 35
Parameter: 13
Others: 23

Other Characteristics

- Life Span for 36 Mozilla bugs
  - 935 days on average to be discovered
  - 140 days to be fixed
- Bug Location
  - 75% inside input-dependent loop or input-event handler
- Server vs. Client Performance Bugs
  - 41 bugs from servers applications
  - 68 bugs from client applications
  - Synchronization issues different

Lessons Learned

- Performance bugs:
  - hide for long periods of time
  - can emerge from non-buggy code as software evolves
  - cannot be avoided most of the time
  - can escape testing even when coverage is good
- There is a need for:
  - annotation systems
  - workload changes and performance impact testing
  - diagnostic techniques beyond profiling

Bug Detection

Efficiency Rules in Patches

- Code transformation + condition
  - Improve performance while preserving functionality
- Inspected 109 patches
  - 50 contain efficiency rules
  - Most of them related to the Change Call Sequence Fix
- Conditions on
  - Function call sequences
  - Parameter/return variables
  - Calling contexts

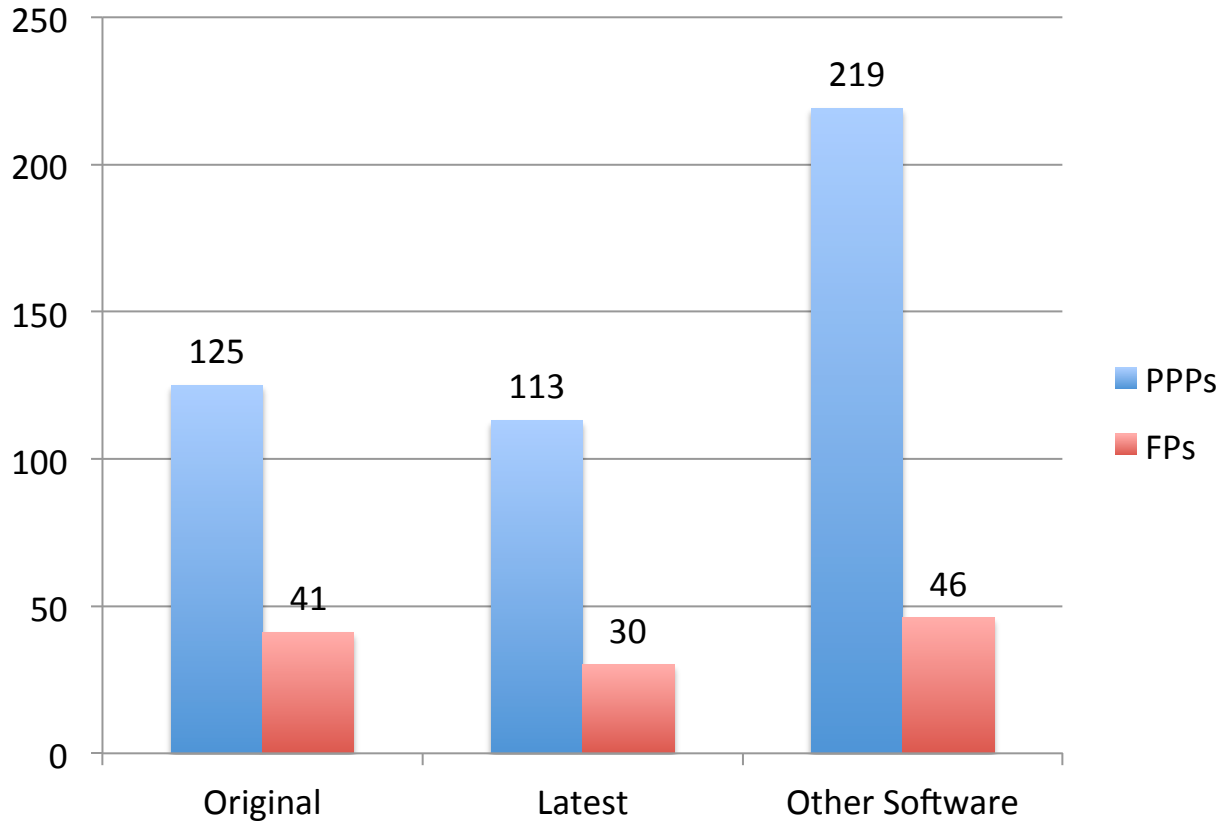
Rule Checkers

- Static vs. Dynamic checks
- Benchmarks
  - Apache, MySQL, and Mozilla
  - 40 patches contain rules, 25 static checkable
- Checkers implementation
  - 14 LLVM checkers for C/C++
  - 11 Python checkers for Java, JavaScript, and C#
- Intraprocedural dataflow analysis in LLVM checkers

Methodology for Rule Checking

- Software Versions
  - Original version (buggy version)
  - Latest version
  - Other software (when cross-application checking)
- Categorization of Results
  - Potential performance problems (PPPs)
  - Bad and good practices
  - False positives

PPPs and False Positives



Performance Bug Example

MySQL Bug 15811

```
- char *end=str+strlen(str);
- if (ismbchar(cs, str, end))
+ if (ismbchar(cs, str, str + cs->mbmaxlen))
```

strings/ctype-mb.c

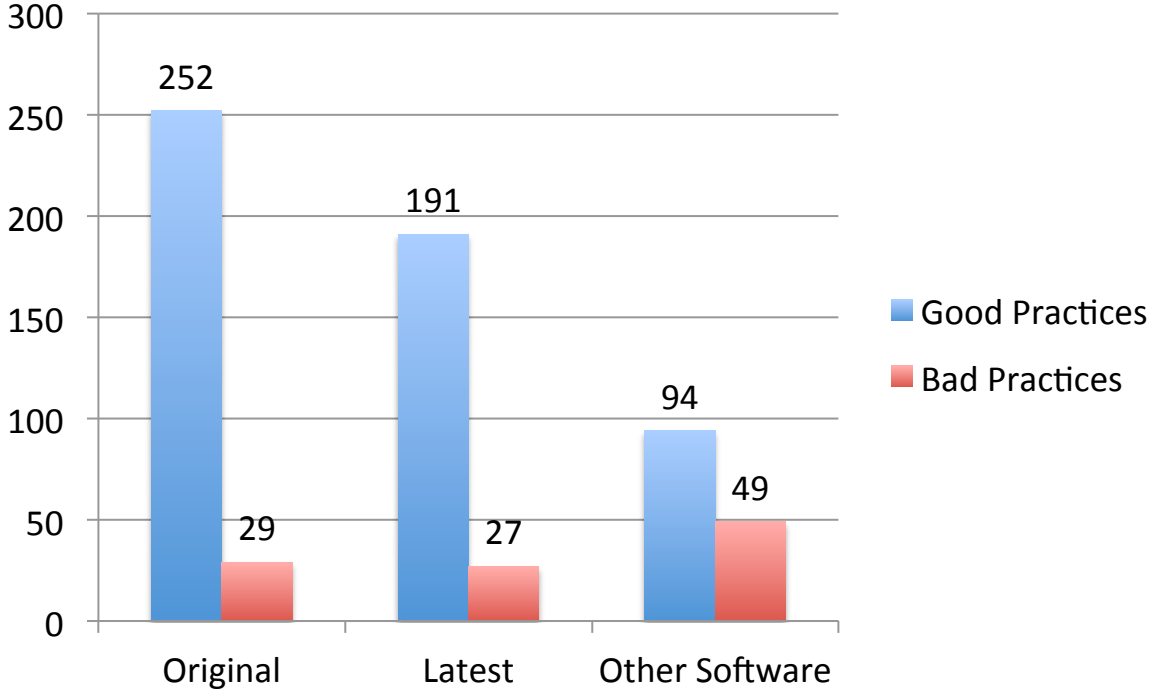
```
// 'end' is only used in the ismbchar checking
- for (end=s; *end ; end++) ;
- if (ismbchar(mysqlcs, s, end) )
+ if (ismbchar(mysqlcs, s, s+mysqlcs->mbmaxlen)
```

libmysql/libmysql.c

False Positives

- 25% false positive rate
- Three main sources:
  1. Python checkers lack type information
  2. Non-function rules difficult to express and check
  3. Some rules require workloads, and measuring runtime

Bad and Good Practices



Performance and Effectiveness

- Analysis Performance
  - Python checkers take 90 sec per 10 MLOC
  - Each LLVM checker takes 4 to 1270 sec per application
- Analysis Effectiveness
  - Saving effort
  - Improving Performance
  - Maintaining Code Readability
  - Other Usage: Performance Specifications

Can PPPs be detected by other tools?

- Copy-Paste Detectors?
  - Mistakes go beyond copy and paste errors
- Compiler Optimizations?
  - Many PPPs involve library functions and algorithmic inefficiencies
  - Might require expensive interprocedural and points to analyses
- General Rule-Based Detectors?
  - Unlike functional correctness rules, efficiency rules are not rare

Summary

- Studied 109 real-world performance bugs
- Provided guidance for future research on
  - Performance-bug avoidance
  - Performance testing
  - Performance-bug detection
- Explored rule-based performance-bug detection
  - Identified efficiency rules from real-world code
  - Found hundreds of performance bugs