SOFT620020.01
Advanced Software Engineering

Bihuan Chen, Pre-Tenure Assoc. Prof.

bhchen@fudan.edu.cn

https://chenbihuan.github.io

Course Outline

Date	Topic
Sep. 09	Introduction
Sep. 16	Testing Overview
Sep. 23	Guided Random Testing
Sep. 30	Search-Based Testing
Oct. 12	**Performance Analysis**
Oct. 14	Presentation 1
Oct. 21	Security Testing
Oct. 28	Compiler Testing

Date	Topic
Nov. 04	Mobile Testing
Nov. 11	Delta Debugging
Nov. 18	Presentation 2
Nov. 25	Bug Localization
Dec. 02	Automatic Repair
Dec. 09	Symbolic Execution
Dec. 16	Big Code Analysis
Dec. 23	Presentation 3

Discussion – Performance Analysis?

- Do you care about the performance of your program?
- How do you tune the performance of your program?
- How will you define a performance problem/issue/bug?

Understanding and Detecting Real-World Performance Bugs

Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, Shan Lu

PLDI 2012, Citation: 220

Software Efficiency is Critical

- No one wants slow and inefficient software
  - Frustrate end users
  - Cause economic loss
- Software efficiency is increasingly important
  - Hardware is not getting faster (per-core)
  - Software is getting more complex
  - Energy saving is getting more urgent

Performance Bugs – Case Study 1

Buggy Code Snippet

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

- When the input image is a transparent image, all the computation in Draw is useless
- Mozilla developers did not expect that transparent images are commonly used by web developers to help layout

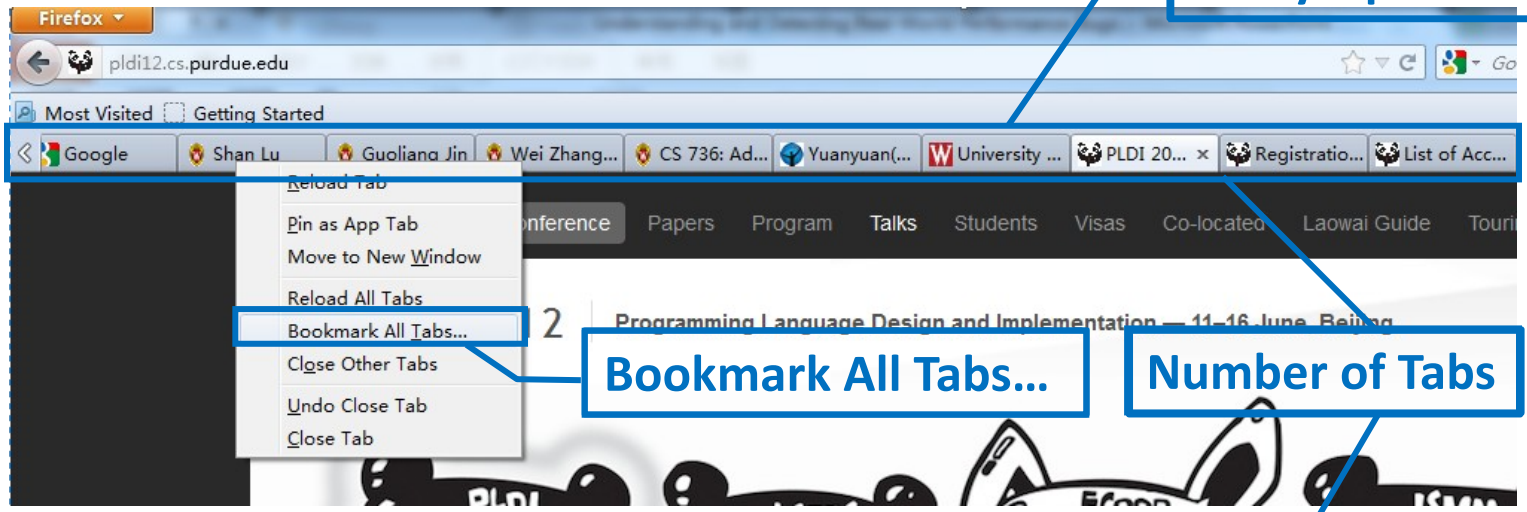
Performance Bugs – Case Study 2

Buggy Code Snippet

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

- This was not a bug until Web 2.0, where XMLHttpRequests (XHRs) are very frequent
- It caused Firefox to consume 10X more CPU at idle GMail pages than Safari

Performance Bugs – Case Study 3



Buggy Code Snippet

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

Bookmark One URL

Performance Bugs – Case Study 4

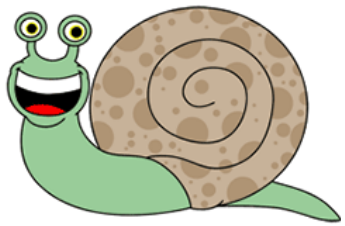
Buggy Code Snippet

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

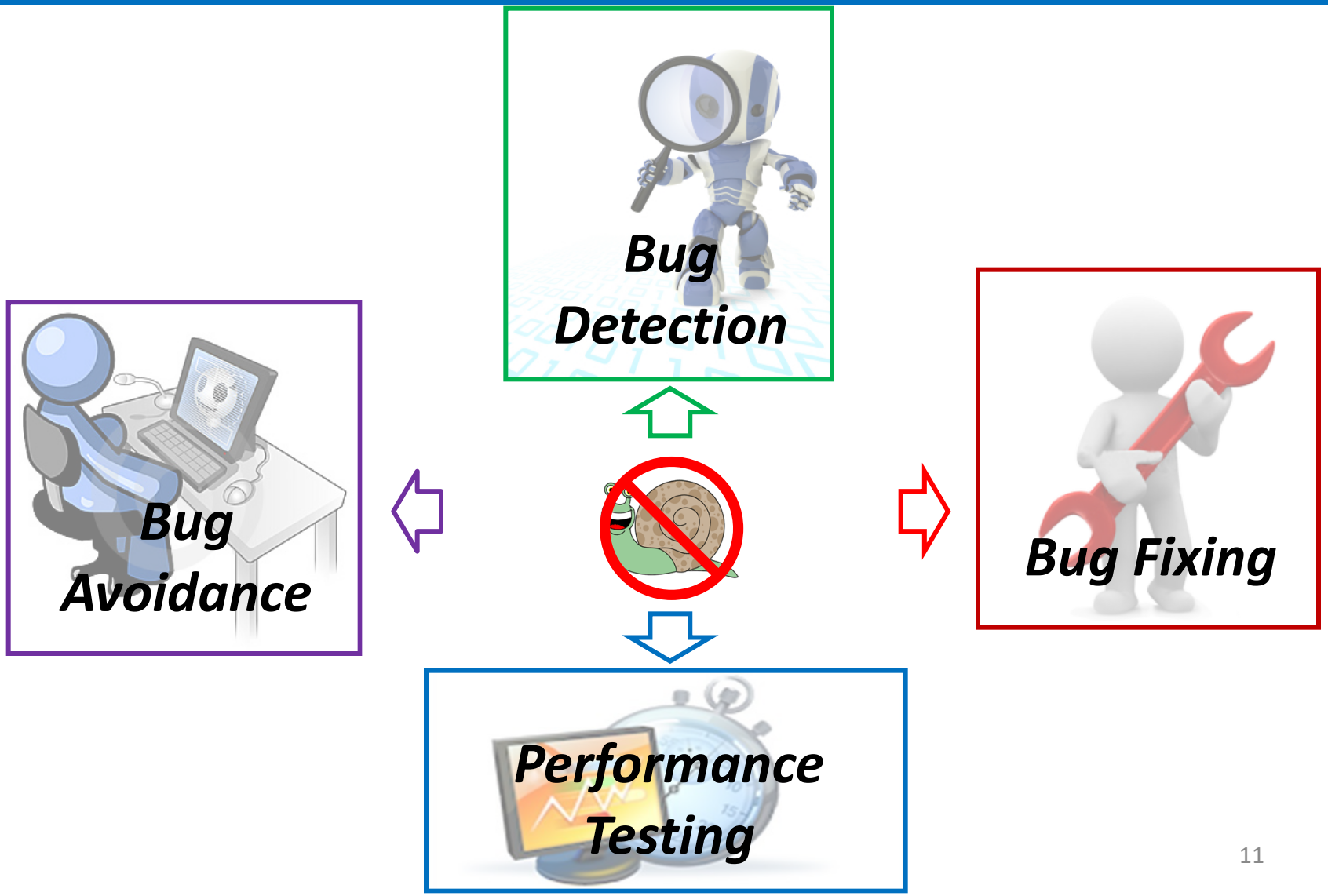
- random() contains a lock, which serializes every thread that invokes it
- Using it in “fastmutex” caused 40X slowdown

Need to Fight Performance Bugs

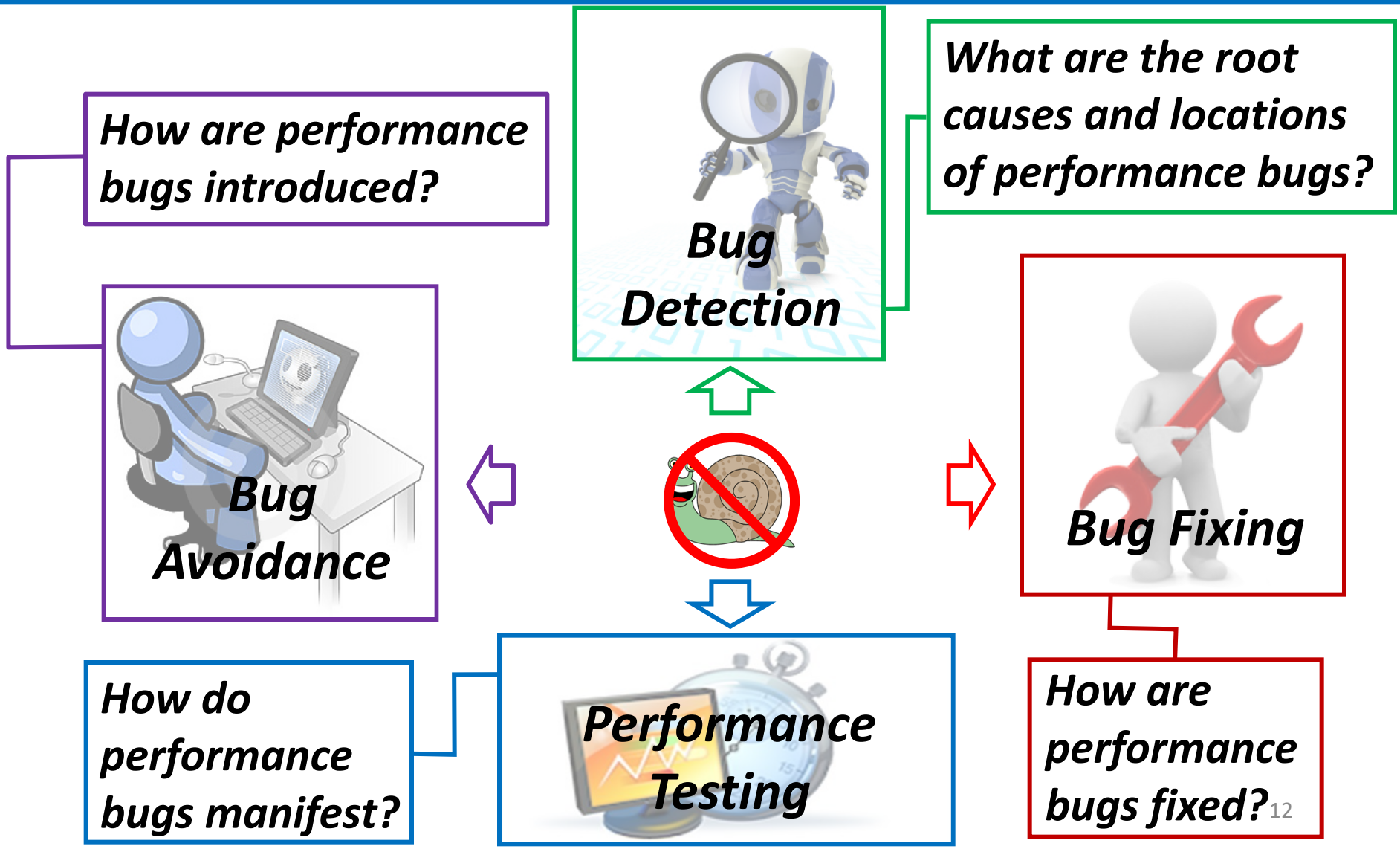
- Performance bugs are severe
  - Escaping compiler optimization
  - Large speed up after fixing small code regions
- Performance bugs are common
  - 5 to 50 Mozilla perf. bugs are fixed each month over the past 10 years
- Performance bugs are costly to diagnose
  - Non fail-stop symptoms
  - Several months of effort by experts to find a couple of performance bugs



How to Fight Performance Bugs



How to Fight Performance Bugs



Contributions

- 1st comprehensive study of real-world perf. bugs
  - Study 109 bugs from 5 real-world applications
  - Guide future performance bug research (avoidance, detection, testing, and fixing)

- Rule-based bug detection
  - Build 25 checkers
  - Find 332 previously unknown potential perf. bugs
  - Guide future performance bug detection

Methodology

- Applications and Bug Source

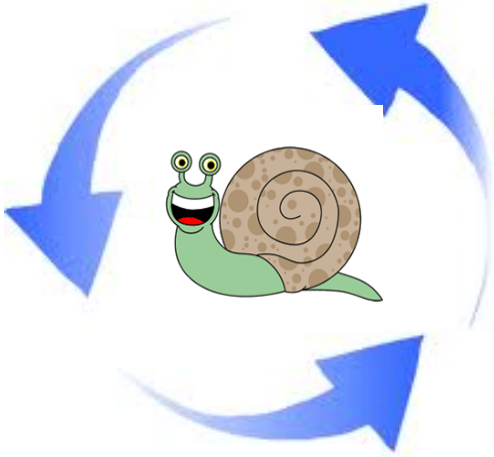
Application	Software Type	Language	MLOC	Bug DB History	Tags	# Fixed Bugs
Apache	Utility + Server + Library	C/Java	0.45	13 y	N/A	25
Chrome	GUI Application	C/C++	14.0	4 y	N/A	10
GCC	Compiler	C/C++	5.7	13 y	Compile-time-hog	10
Mozilla	GUI Application	C++/JS	4.7	14 y	perf	36
MySQL	Server Software	C/C++/C#	1.3	10 y	S5	28

Total: 109

Outline of Characteristics Study

What are the root causes and locations of perf. bugs?

How are perf. bugs introduced?



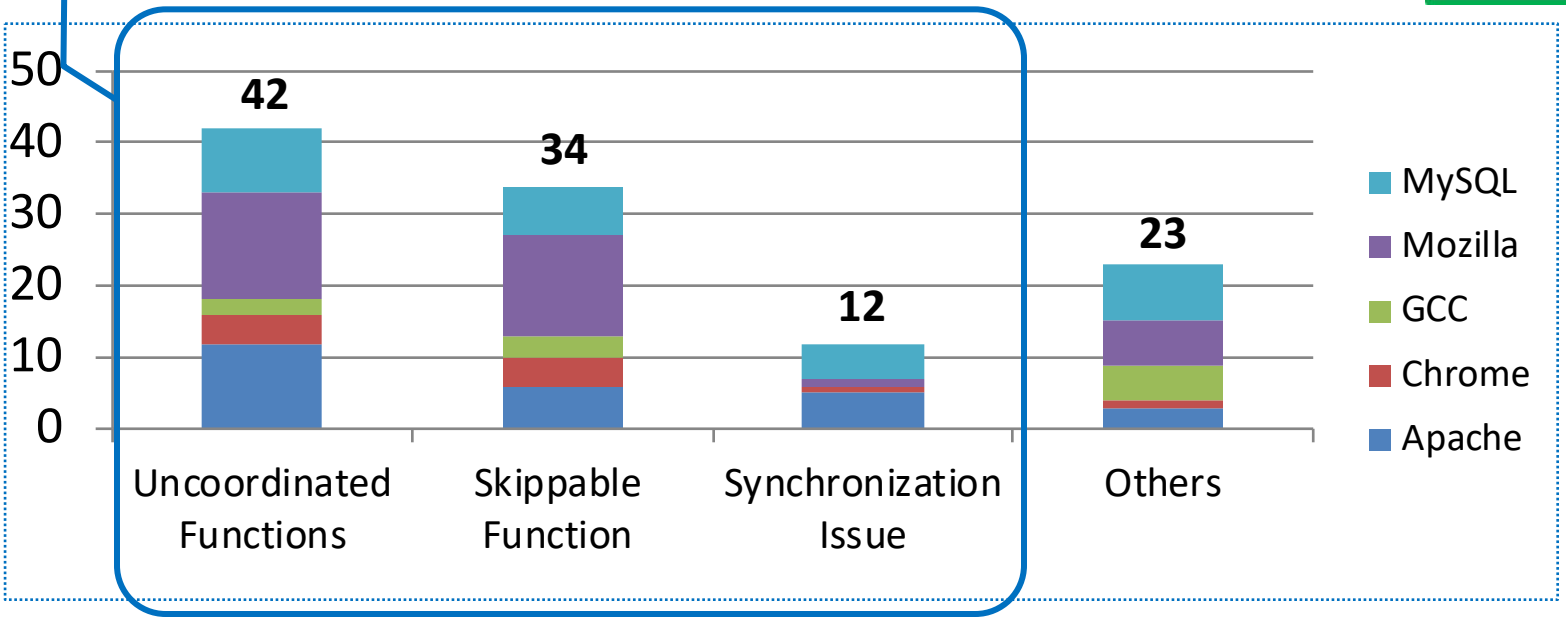
How are perf. bugs fixed?

How can perf. bugs manifest?

Root Causes of Perf. Bugs

Performance Bug Detection

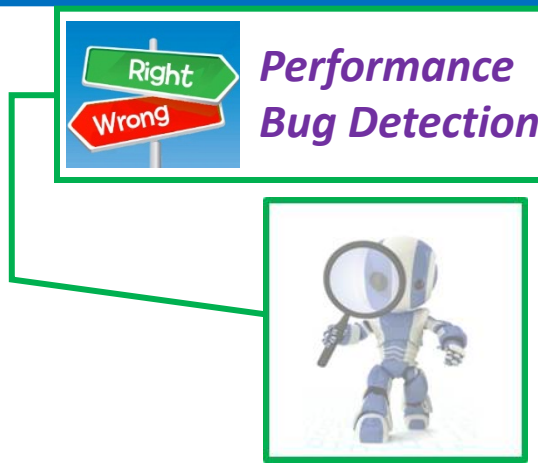
Dominating

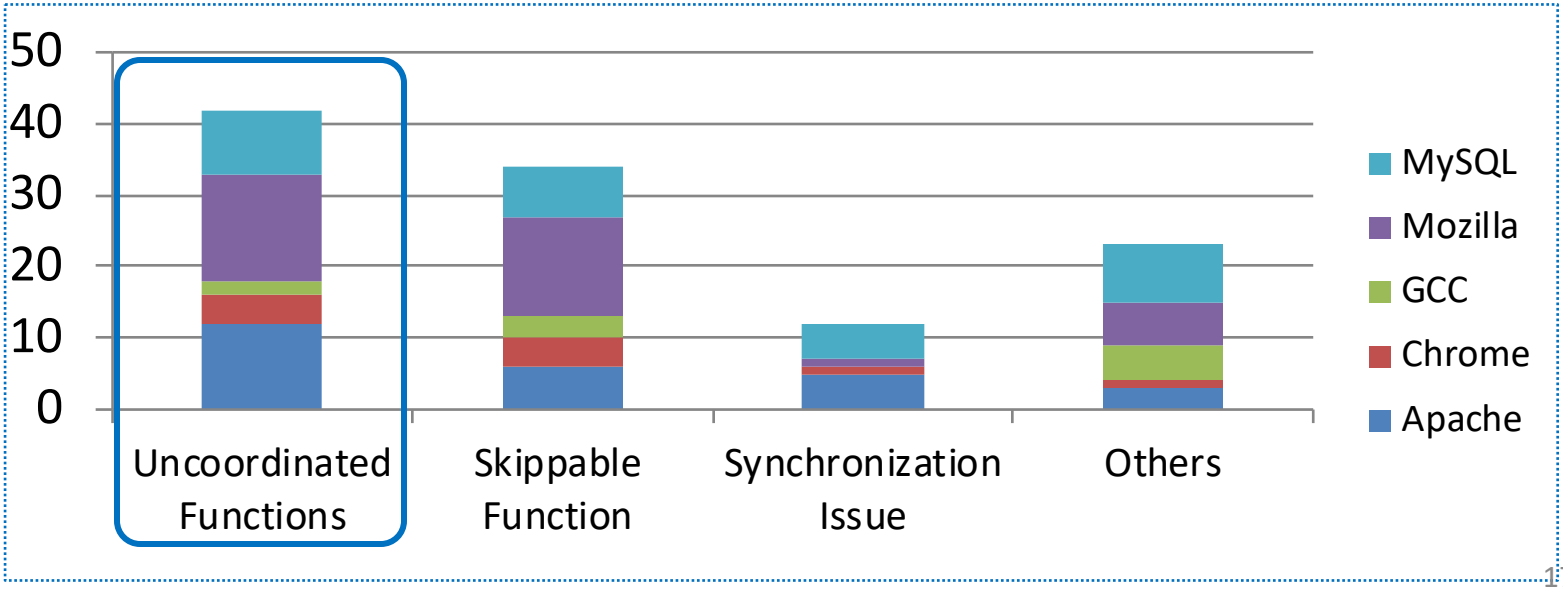


Root Causes of Perf. Bugs (cont.)

Mozilla Bug 490742 & Patch

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





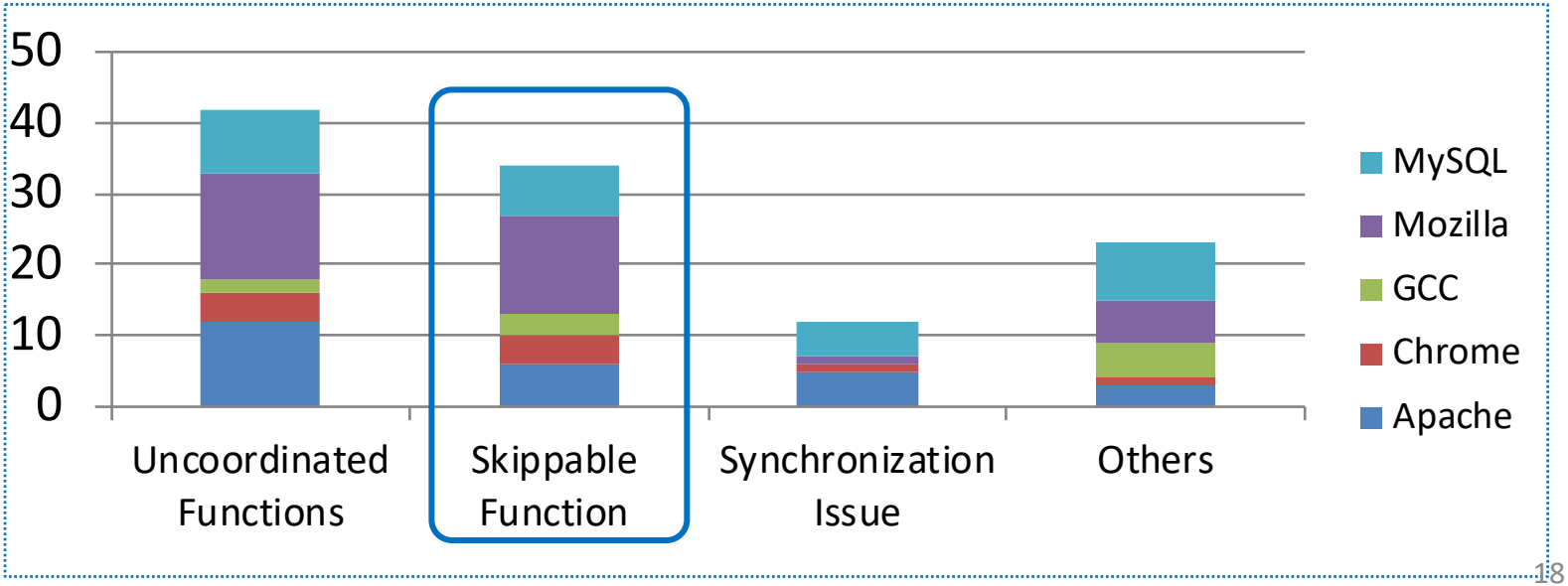
Function calls take a detour to generate results

Root Causes of Perf. Bugs (cont.)

Mozilla Bug 66461

```
nsImage::Draw(…) {
+ if(mIsTransparent) return;
  …
}
```

Performance Bug Detection

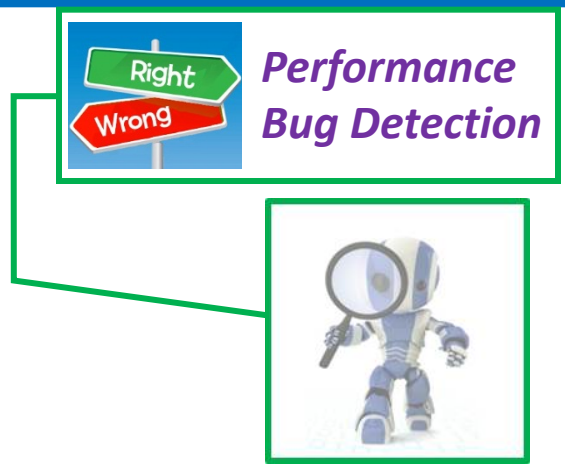


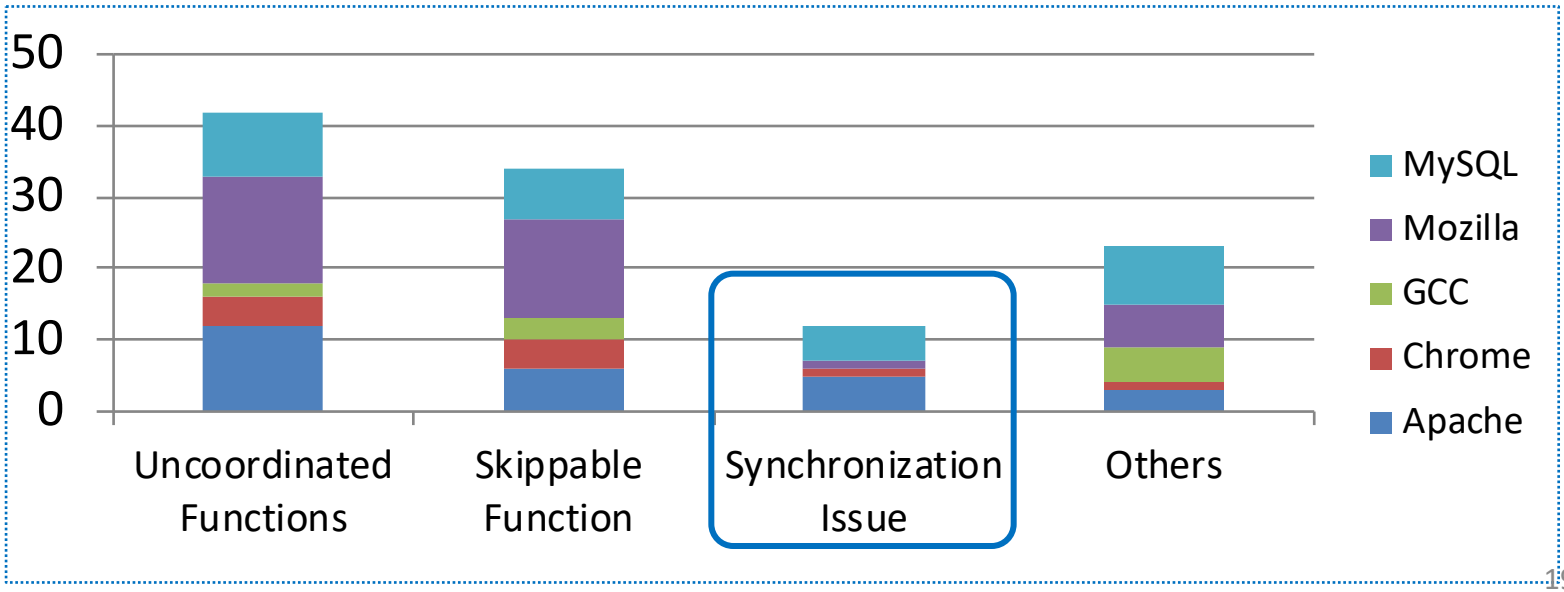
A function call with un-used results

Root Causes of Perf. Bugs (cont.)

MySQL Bug 38941 & Patch

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

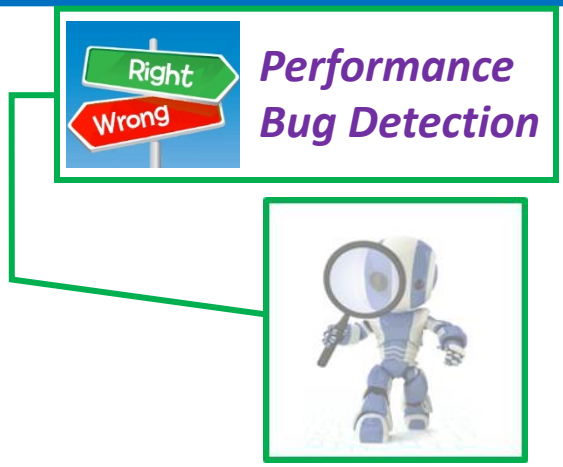


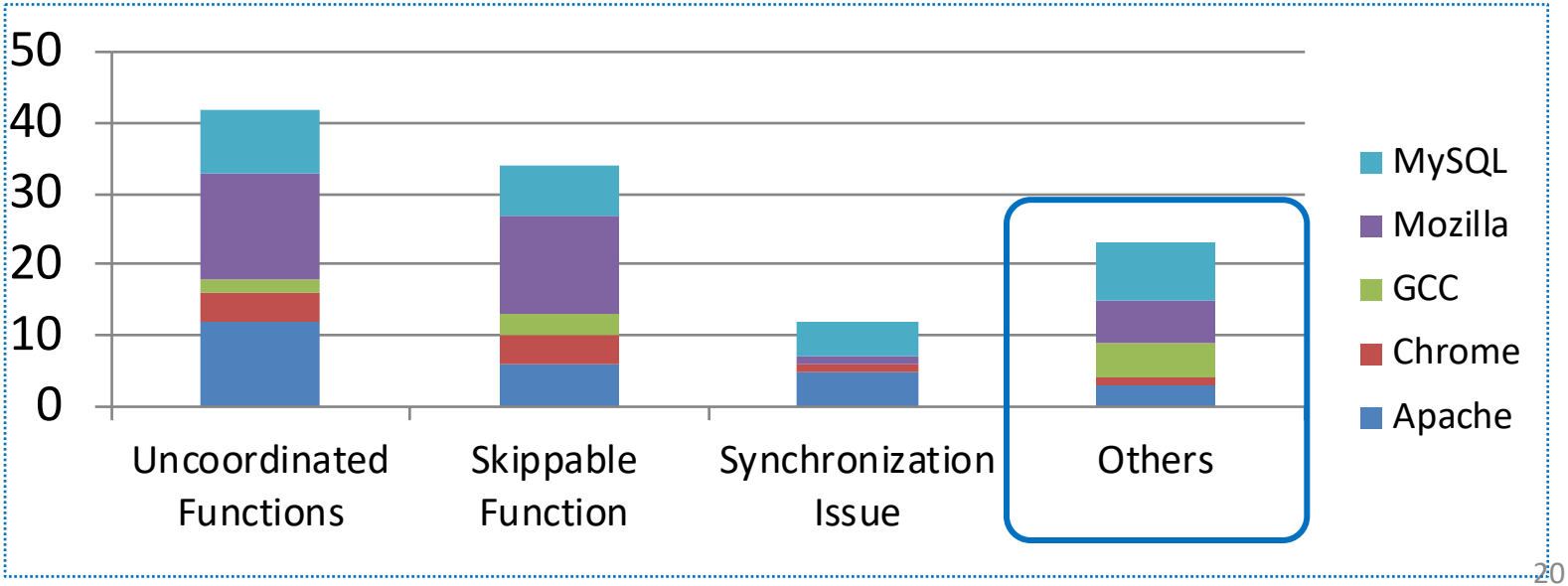


Inefficient synchronization among threads

Root Causes of Perf. Bugs (cont.)

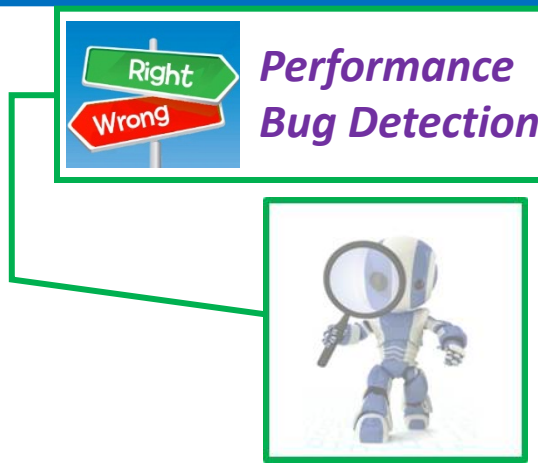
Some use wrong data structures; some are related to hardware architecture issues; and Some are caused by high-level design/algorithm issues.

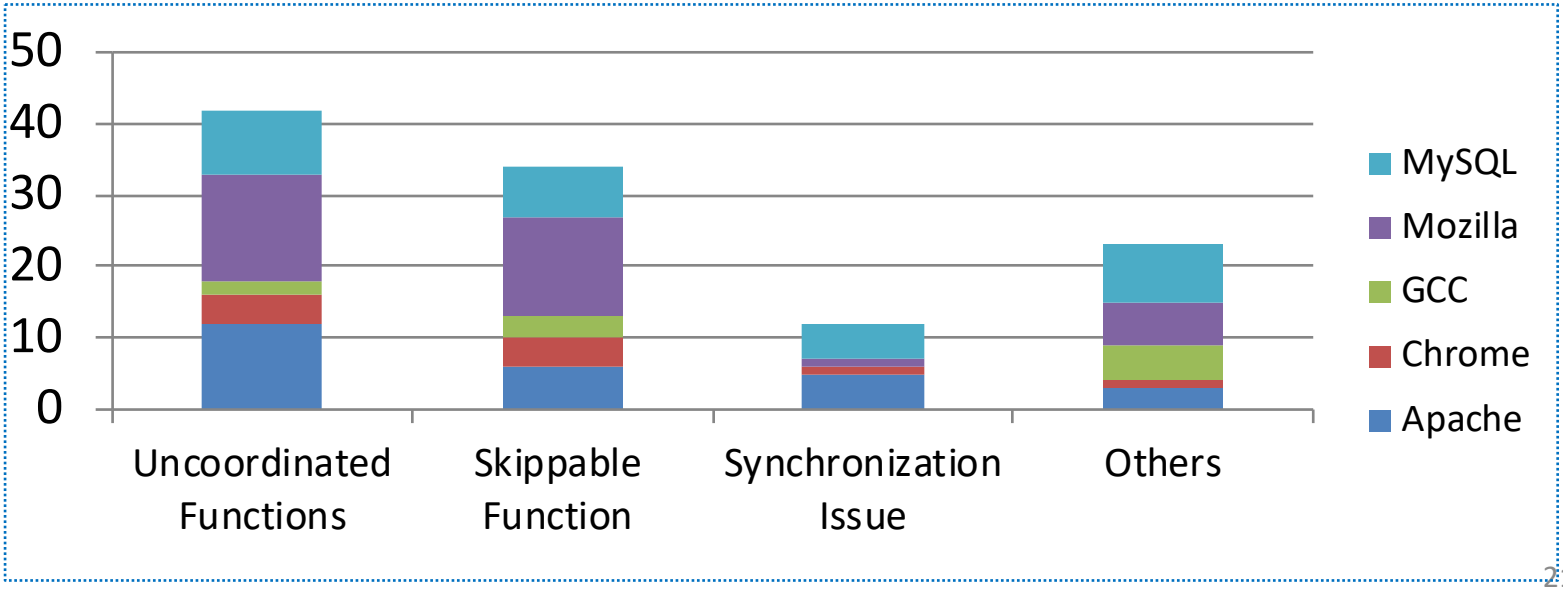




Root Causes of Perf. Bugs (cont.)

Implication: Future bug detection research should focus on these common root causes.

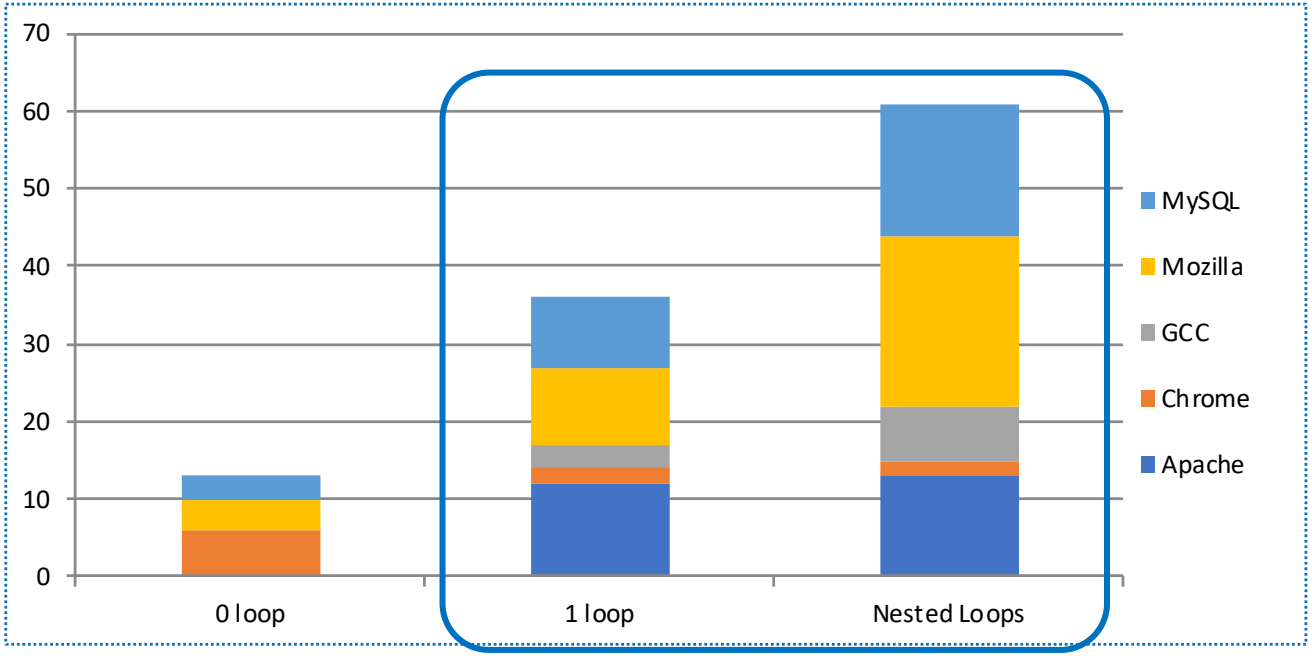




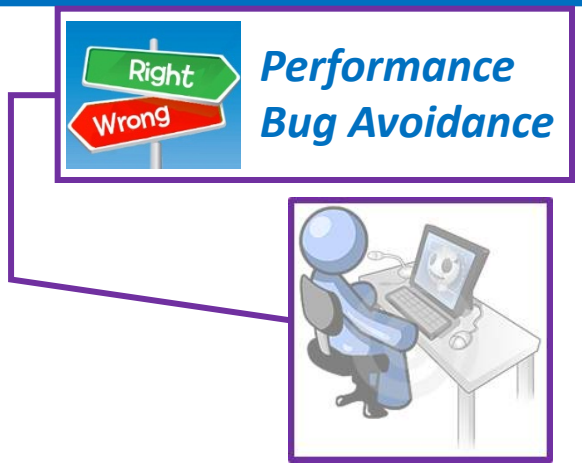
Locations of Perf. Bugs

Implication: Detecting inefficiency in loops is critical.

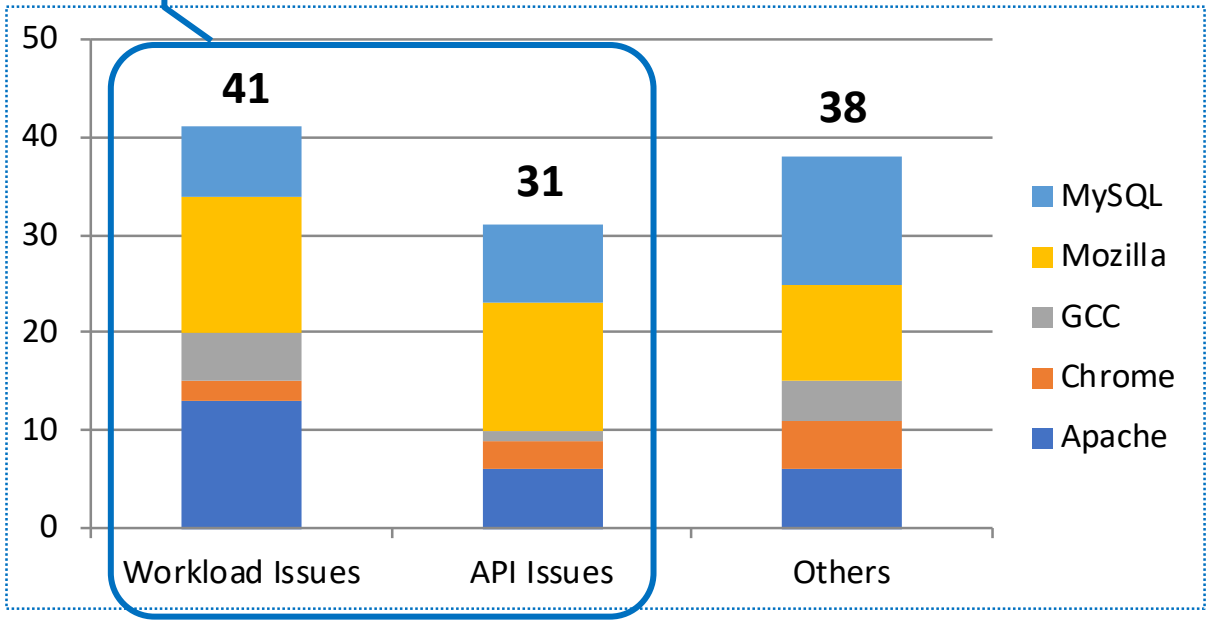
Performance Bug Detection



How Perf. Bugs are Introduced



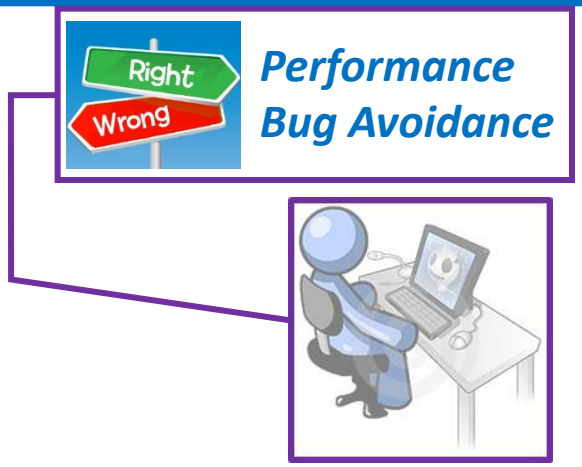
Dominating

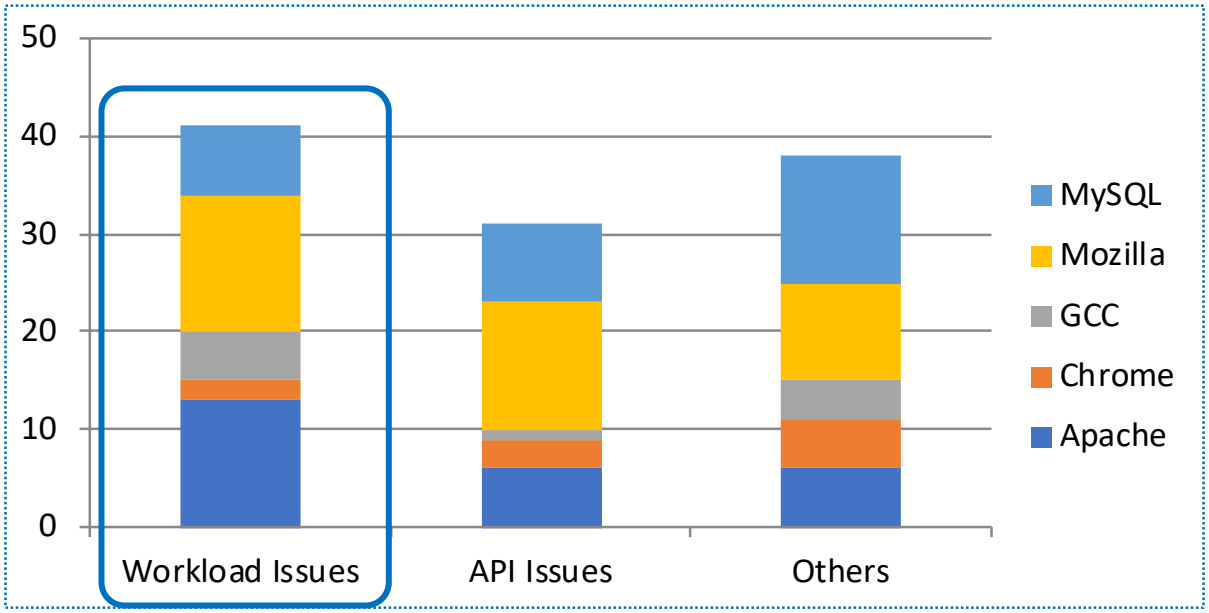


How Perf. Bugs are Introduced (cont.)

Mozilla Bug 66461

```
nsImage::Draw(...) {
+   Not Born Buggy!
    ...
}
```



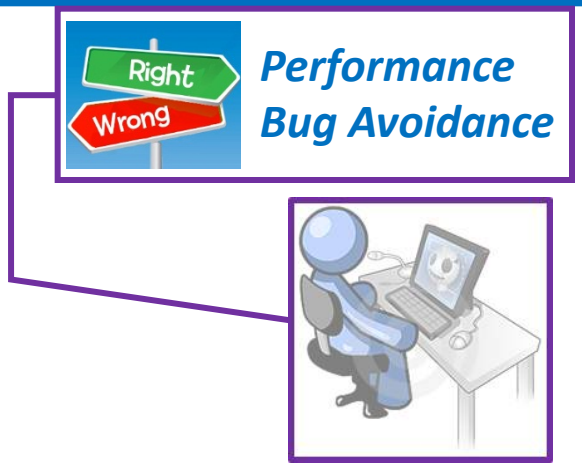


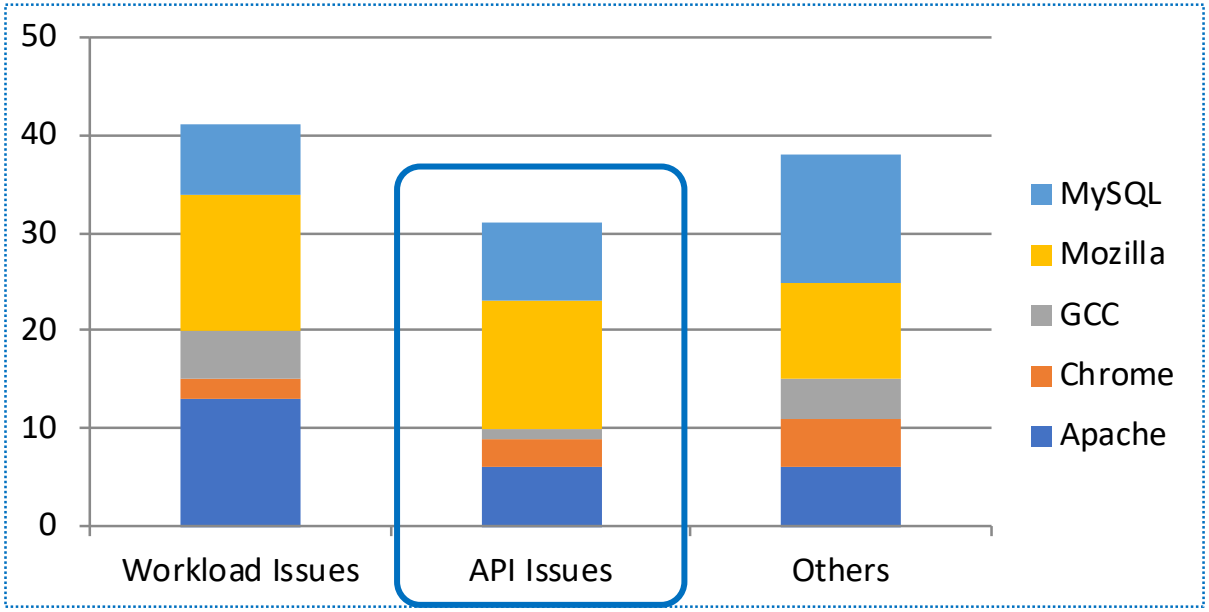
Developers' workload assumption is wrong or outdated

How Perf. Bugs are Introduced (cont.)

MySQL Bug 38941 & Patch

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

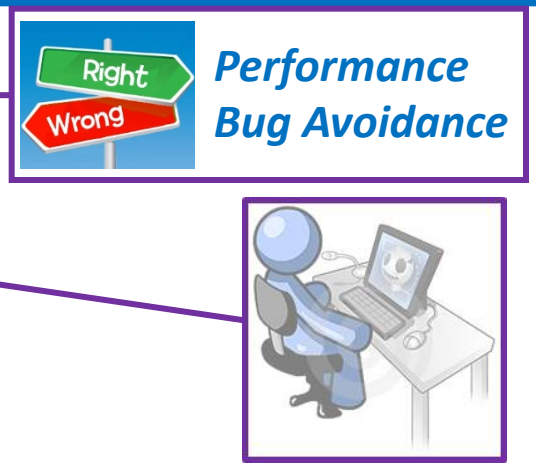


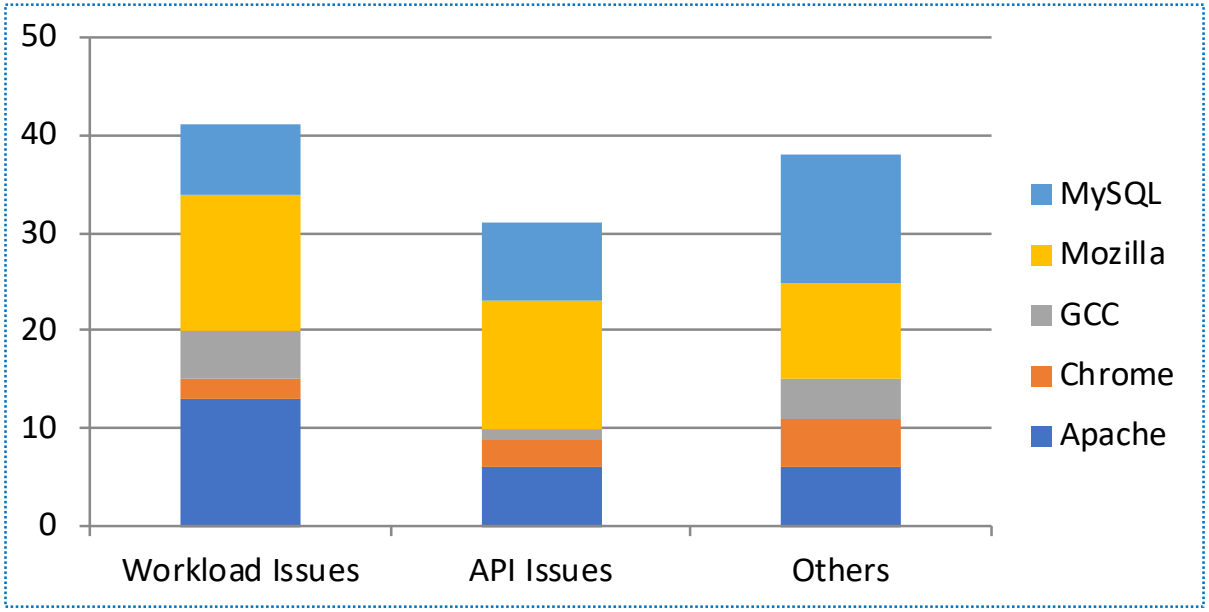


Misunderstand performance features of functions/APIs

How Perf. Bugs are Introduced (cont.)

Implication: Performance aware annotation systems and change-impact analysis tools are needed.

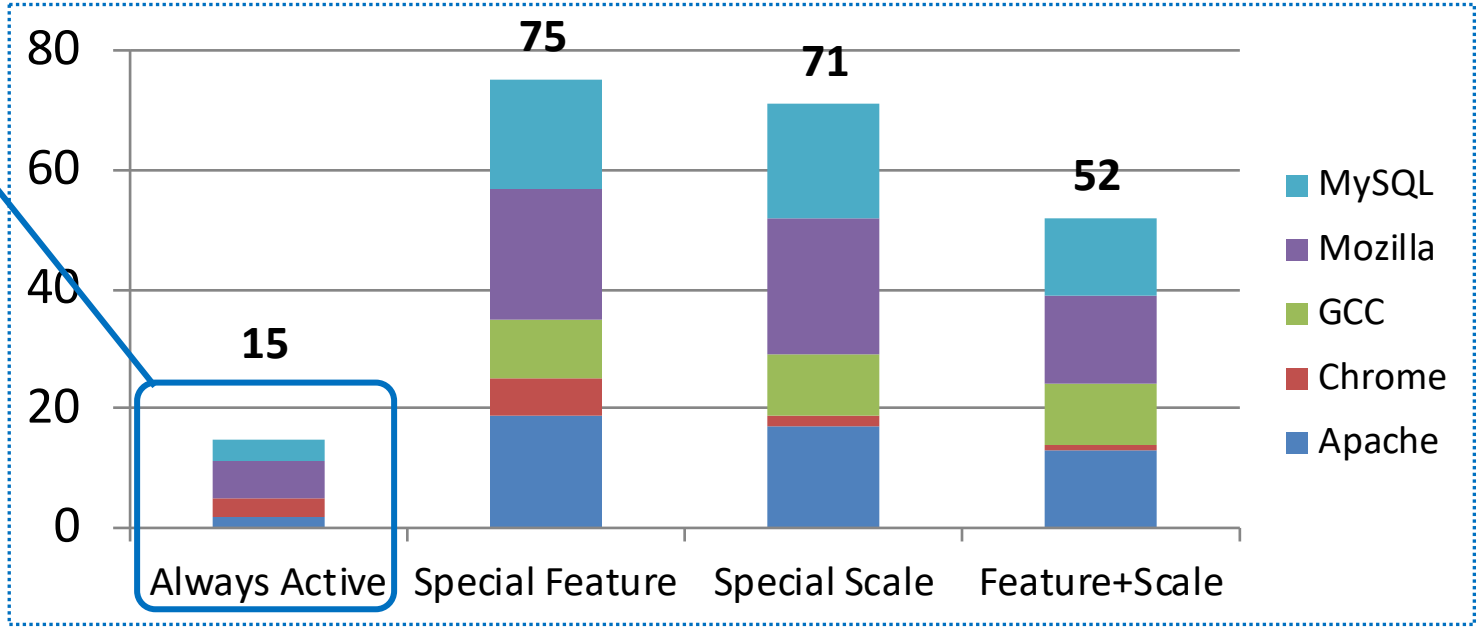




How Perf. Bugs Manifest

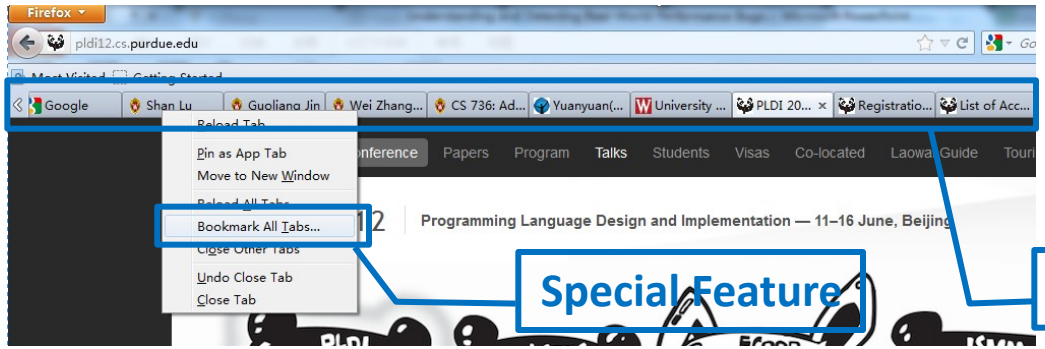
Performance Testing

Unique, Severe



Almost every input on every platform can trigger this bug

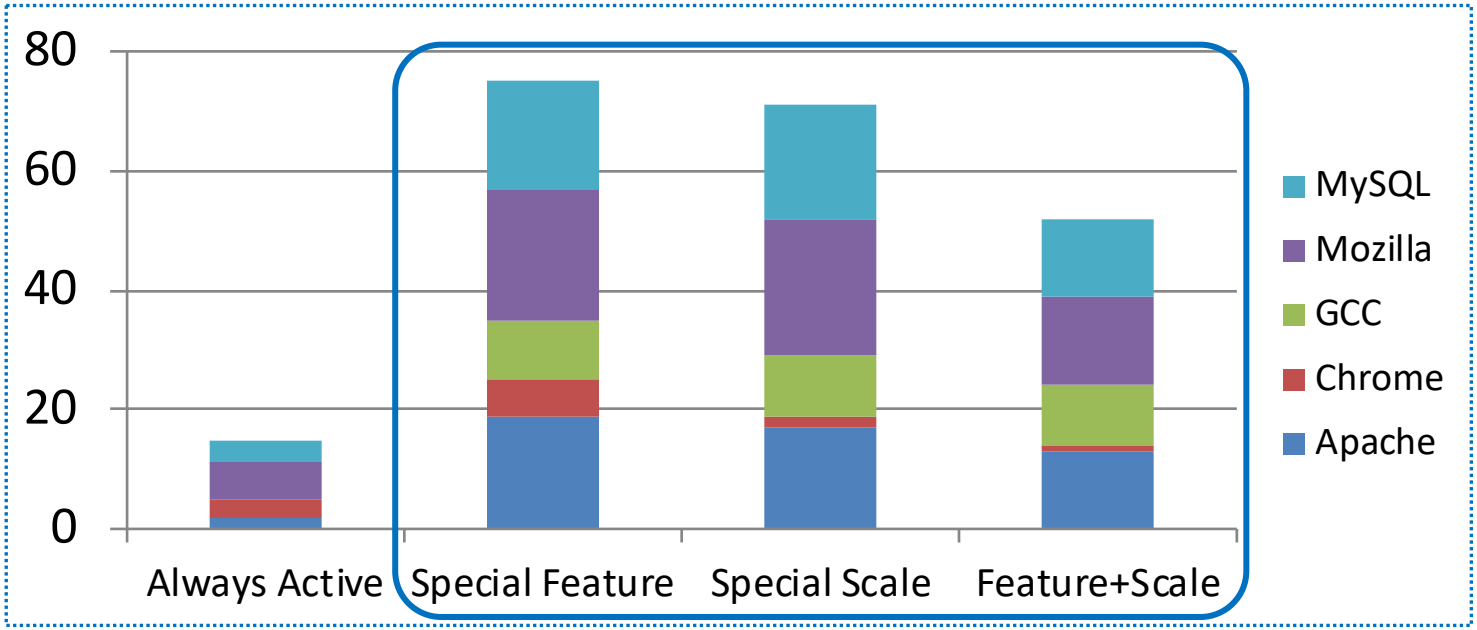
How Perf. Bugs Manifest (cont.)



Special Feature

Large Scale

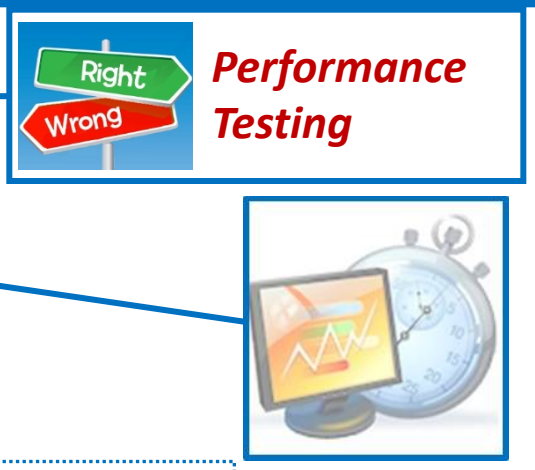
Performance Testing

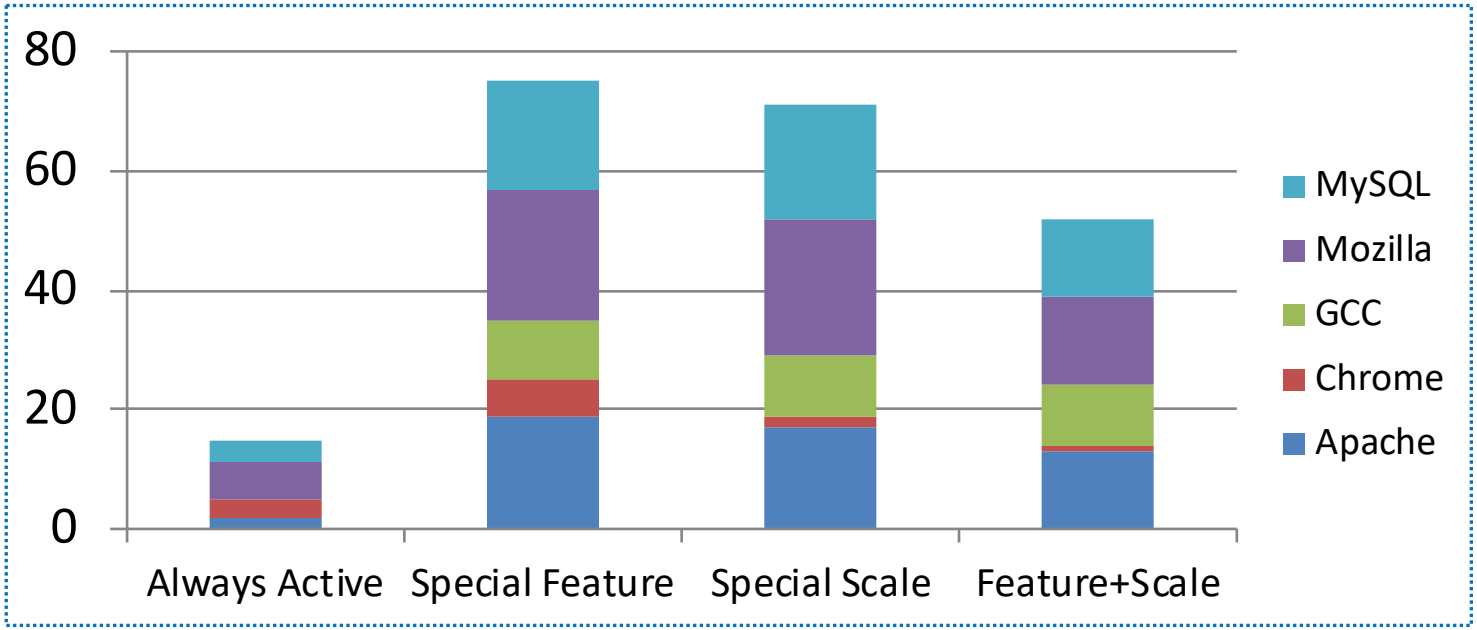


Special-value (large-scale) inputs to execute specific code regions (many times)

How Perf. Bugs Manifest (cont.)

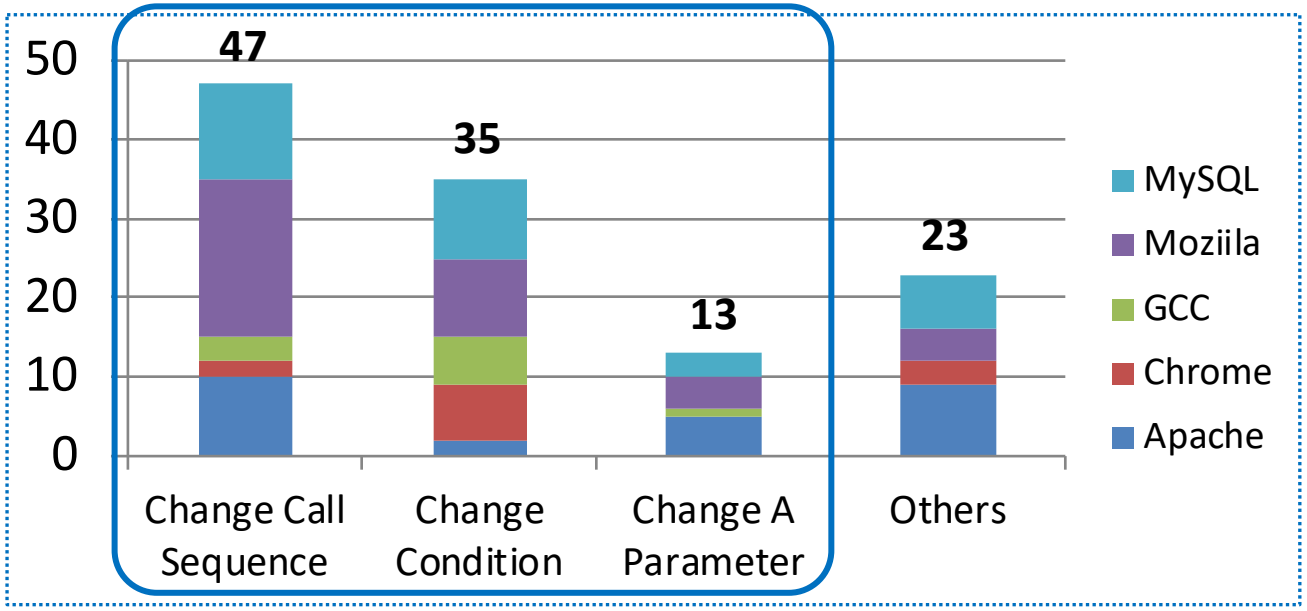
Implication: New input generation tools are needed.





How Perf. Bugs are Fixed

- Patch sizes are small
  - 42 patches are no larger than 5 LOC
  - Median patch size = 8 lines of codes
  - Fixing perf. bugs does not hurt readability



Other Characteristics

- Performance bugs hide for a long time
  - 36 Mozilla perf. bugs took **935** days on average to get discovered, and another **140** days on average to be fixed
  - 36 Mozilla functional bugs took **252** days on average to get discovered, and another **117** days on average to be fixed
- Server vs. Client; Java vs. C/C++
  - More server bugs caused by synchronization issues
  - Others are consistent
- Correlations among characteristic categories
  - **skippable function** root cause ↔ **change condition** fix

Efficiency Rules

- An efficiency rule includes two parts
  - A transformation that improves code efficiency
  - An applying condition of the transformation

Rules applicable to only one application

Mozilla Bug 490742 & Patch

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

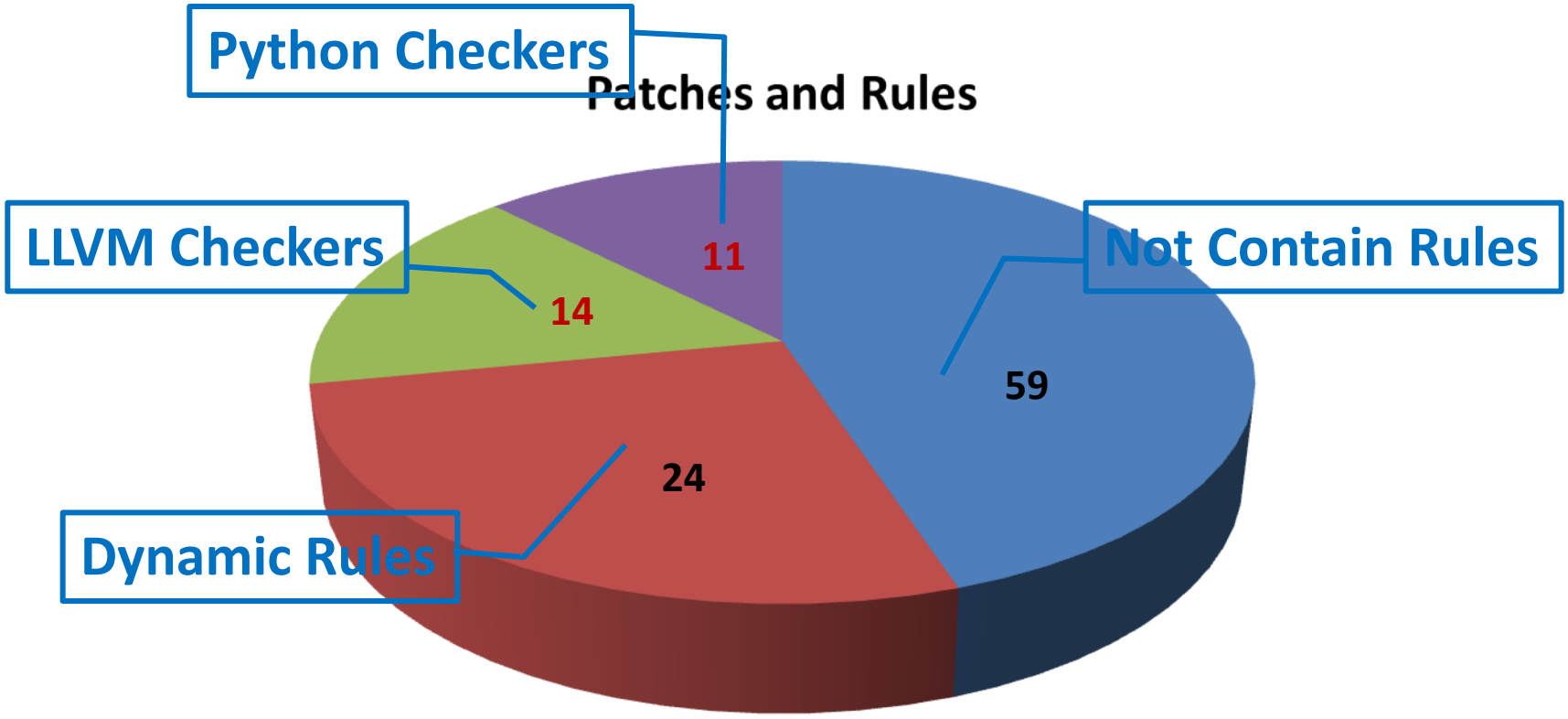
Rules applicable cross applications

MySQL Bug 38941 & Patch

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

Rule Extraction and Checker Implementation

- Identifying Efficiency Rules

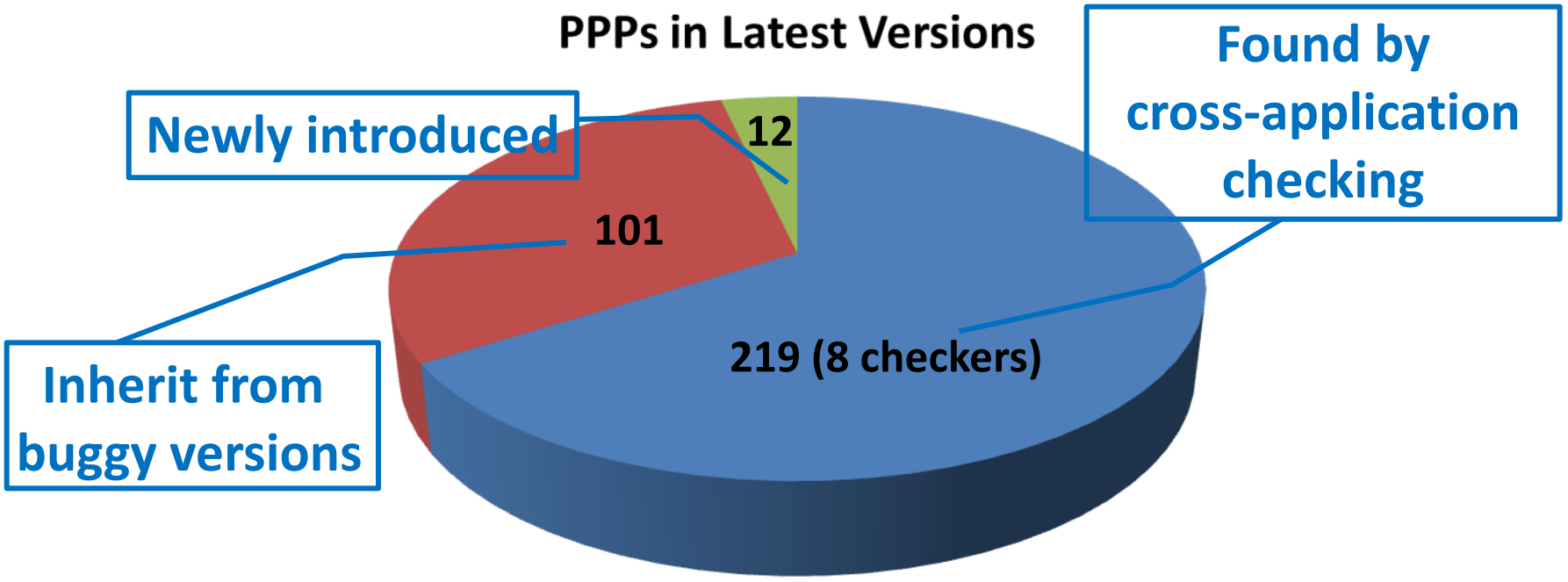


Rule Examples

Call Sequence Conditions
function C::f() is invoked
function f1 is always followed by f2
function f1 is called once in each iteration of a loop
Parameter/Return Conditions
*n*th parameter of f1 equals K (constant)
*n*th parameter of f1 is the same variable as the return of f2
a param. of f1 and a param. of f2 point to the same object
the return of f1 is not used later
the parameter of f1 is not modified within certain scope
the input is a long string
Calling Context Conditions
function f1 is only called by one thread
function f1 can be called simultaneously by multiple threads
function f1 is called many times during the execution

Rule-Violation Detection Results

- 17 checkers find **125** PPPs in original buggy versions
- 13 checkers find **332** PPPs in latest versions



Efficiency rules and rule-based performance-bug detection is promising!

* PPP: Potential Performance Problem

Findings and Implications

Findings	Implications
Skippable Functions and Uncoordinated Functions are most common root causes.	Inefficiency detection should focus on these common patterns.
Most bugs involve nested loops.	
Most bugs are introduced by Workload Mismatch and API Misunderstanding.	Inefficiency avoidance needs performance annotations and change-impact analysis.
29 out of 109 bugs were not born buggy.	
Many bug-exposing inputs need special features and large scales.	Test-input generation should consider coverage + intensity.
Most patches are a few lines of code.	Efficiency != bad readability
Similar mistakes are made in many places by many developers in many software.	Efficiency rules are important!

Conclusions

- First characteristics study on performance bugs
- Efficiency rules widely exist and are very useful

CARAMEL: Detecting and Fixing Performance Problems That Have Non-Intrusive Fixes

Adrian Nistor, Po-Chun Chang, Cosmin Radoi, Shan Lu

ICSE 2015, ACM SIGSOFT Distinguished Paper Award

How Likely to Fix a Detected Perf. Bug?

- Potential drawbacks of fixing perf. bugs
  - Introduce functional bugs
  - Break good SE practices
  - Take time and effort
  - Slow down other code

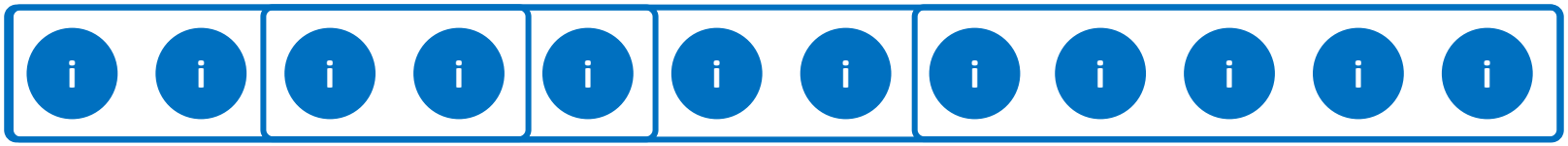
- Potential benefits of fixing perf. bugs
  - Speed up code execution
  - Often difficult to estimate the speedup
    - How frequent or important are the bug-triggering input?
    - How is the speedup for other bug-triggering inputs?

Contributions

- Detect and fix perf. bugs with non-intrusive fixes
  - When a condition becomes true during loop executions, all the remaining computation in the loop is wasted
- Evaluation on 11 Java applications
  - 61 new perf. bugs with 51 of them been fixed
- Evaluation on 4 C/C++ applications
  - 89 new perf. bugs with 65 of them been fixed

Characteristics about Loop Perf. Bugs

- Where the computation is wasted?
  - Every iteration
  - Every iteration at the end of the loop
  - Every iteration at the start of the loop



- How the computation is wasted?
  - RI is not executed under break condition
  - RI generates useless results under break condition

	Every	Late	Early
No Result	Type 1	Type 2	Type Y
Useless Result	Type X	Type 3	Type 4

* RI: Result Instruction that write to variables live and memory reachable after the loop

Example of Type 1 RI

	Every	Late	Early
No Result	Type 1	Type 2	Type Y
Useless Result	Type X	Type 3	Type 4

```
1 Class[] argTypes = ...
2 for (Iterator i = methods.iterator(); i.hasNext();) {

4    MethodNode mn = (MethodNode) i.next();
5    boolean isZeroArg = (argTypes == null || argTypes.length == 0);
6    boolean match =mn.getName().equals(methodName) && isZeroArg;
7    if (match) return true; // RI
8 }
```

Example of Type 2 RI

	Every	Late	Early
No Result	Type 1	Type 2	Type Y
Useless Result	Type X	Type 3	Type 4

```
1  boolean alreadyPresent = false;
2  while (itActualEmbeddedProperties.hasNext()) { // Loop 1

4     ... // non-RIs
5        while (itNewValues.hasNext()) { // Loop 2
6           ... // non-RIs
7           while (itOldValues.hasNext() && !alreadyPresent) { // Loop 3
8              oldVal = (TextType) itOldValues.next();
9              if(oldVal.getStringValue().equals(newVal.getStringValue())){
10                alreadyPresent = true; // RI 1
11          }}
12          if (!alreadyPresent) {
13             embeddedProp.getContainer().addProperty(newVal);// RI 2
14 }}}}}
```

Example of Type 3 RI

	Every	Late	Early
No Result	Type 1	Type 2	Type Y
Useless Result	Type X	Type 3	Type 4

```
1  boolean elExp = ...
2  while (nodes.hasNext()) {

4    ELNode node = nodes.next();
5    if (node instanceof ELNode.Root) {
6      if (((ELNode.Root) node).getType() == '$') {
7        elExp = true; // RI 1
8      } else if (checkDeferred && ((ELNode.Root) node).getType()=='#'
9                  && !pageInfo.isDeferredSyntaxAllowedAsLiteral() ) {
10       elExp = true; // RI 2
11 }}}
```

Example of Type 4 RI

	Every	Late	Early
No Result	Type 1	Type 2	Type Y
Useless Result	Type X	Type 3	Type 4

```
1 int length = ...
2 for (int idx = 0; idx < headerSize; idx++) {

4   Header hd = mngr.getHeader(idx);
5   if (HTTPConstants.HEADER.equalsIgnoreCase(hd.getName())) {
6     length = Integer.parseInt(hd.getValue()); // RI

8 }}
```

Type X and Type Y

	Every	Late	Early
No Result	Type 1	Type 2	Type Y
Useless Result	Type X	Type 3	Type 4

```
1  boolean elExp = true;
2  while (nodes.hasNext()) {

4     ELNode node = nodes.next();
5     if (node instanceof ELNode.Root) {
6        if (((ELNode.Root) node).getType() == '$') {
7           elExp = true; // RI 1
8        } else if (checkDeferred && ((ELNode.Root) node).getType()=='#'
9                    && !pageInfo.isDeferredSyntaxAllowedAsLiteral() ) {
10          elExp = true; // RI 2
11 }}}
```

Discussion – RI Types

```
1  boolean annotNotFound = ...
2  boolean sigFieldNotFound = ...
3  for ( COSObject cosObject : cosObjects ) {

5     ... // some non−RIs
6     if (annotNotFound && COSName.ANNOT.equals(type)) {
7        ... // RI 1 and some non-RIs
8        signatureField.getWidget().setRectangle(rect); // RI 2
9        annotNotFound = false; // RI 3
10    }
11    if (sigFieldNotFound && COSName.SIG.equals(ft)&&apDict!=null){
12       ... // RI 4, RI 5, RI 6, RI 7 and some non-RIs
13       acroFormDict.setItem(COSName.DR, dr); // RI 8
14       sigFieldNotFound=false; // RI 9
15 }}
```

High-Level Algorithm

```
1  void detectPerformanceBug(Loop l, Method m, AliasInfo alias) {
2     Set⟨Instruction⟩ allRIs = getRIs(l, m, alias);
3     Set⟨Condition⟩ allCond = new Set⟨Condition⟩();
4     for (Instruction r : allRIs) {
5        Condition one = typeOne(r, l, m, alias);
6        Condition two = typeTwo(r, l, m, alias);
7        Condition three = typeThree(r, l, m, alias);
8        Condition four = typeFour(r, l, m, alias, allRIs.size());
9        if(one.false()&&two.false()&&three.false()&&four.false()) return;
10       allCond.putIfNotFalse(one, two, three, four);
11    }
12    if(satisfiedTogether(allCond) && notAlreadyAvoided(allCond, l)) {
13       String fix = generateFix(allCond, l, m);
14       reportBugAndFix(fix, allRIs, allCond);
15 }}
```

Evaluation – Target Applications

L	#	App	Description	LoC	Classes(J) Files(C)
Java	1	Ant	build tool	140,674	1,298
	2	Groovy	dynamic language	161,487	9,582
	3	JMeter	load testing tool	114,645	1,189
	4	Log4J	logging framework	51,936	1,420
	5	Lucene	text search engine	441,649	5,814
	6	PDFBox	PDF framework	108,796	1,081
	7	Sling	web app. framework	202,171	2,268
	8	Solr	search server	176,937	2,304
	9	Struts	web app. framework	175,026	2,752
	10	Tika	content extraction	50,503	717
	11	Tomcat	web server	295,223	2,473
C/C++	12	Chromium	web browser	13,371,208	10,951
	13	GCC	compiler	1,445,425	781
	14	Mozilla	web browser	5,893,397	5,725
	15	MySQL	database server	1,774,926	1,684

Evaluation – New Bugs

Application	Type 1 RIs	Type 2+3 RIs	Type 3 RIs	Type 4 RIs	SUM
Ant	0	0	1	0	**1**
Groovy	2	0	7	0	**9**
JMeter	0	0	3	1	**4**
Log4J	0	0	5	1	**6**
Lucene	6	0	7	1	**14**
PDFBox	1	5	3	1	**10**
Sling	0	0	6	0	**6**
Solr	0	0	2	0	**2**
Struts	2	0	2	0	**4**
Tika	0	0	1	0	**1**
Tomcat	0	0	3	1	**4**
Chromium	0	0	13	9	**22**
GCC	1	0	21	0	**22**
Mozilla	0	0	20	7	**27**
MySQL	3	0	13	2	**18**
SUM:	**15**	**5**	**107**	**23**	**150**

61 new bugs
51 have been fixed

89 new bugs
65 have been fixed

Automatically generated fixes for 149 out of 150 bugs!

Evaluation – Overhead

Application	Sequential	Parallel	Speedup (X)
Ant	183	72	2.54
Groovy	345	128	2.70
JMeter	118	52	2.27
Log4J	108	45	2.40
Lucene	1068	417	2.56
PDFBox	106	38	2.79
Sling	355	190	1.87
Solr	1062	627	1.69
Struts	226	77	2.94
Tika	113	42	2.69
Tomcat	258	89	2.90
Chromium	85	n/a	n/a
GCC	3	n/a	n/a
Mozilla	52	n/a	n/a
MySQL	10	n/a	n/a

Reading Materials

- Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and Detecting Real-world Performance Bugs. In PLDI. 77–88.
- Marija Selakovic and Michael Pradel. 2016. Performance Issues and Optimizations in JavaScript: An Empirical Study. In ICSE. 61–72.
- Linhai Song and Shan Lu. 2014. Statistical Debugging for Real-world Performance Problems. In OOPSLA. 561–578.
- Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2014. Characterizing and Detecting Performance Bugs for Smartphone Applications. In ICSE. 1013–1024.
- Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. 2015. CARAMEL: Detecting and Fixing Performance Problems That Have Non-Intrusive Fixes. In ICSE. 902–912.
- Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. 2013. Toddler: Detecting Performance Problems via Similar Memory-access Patterns. In ICSE. 562–571.
- Monika Dhok and Murali Krishna Ramanathan. 2016. Directed Test Generation to Detect Loop Inefficiencies. In FSE. 895–907.
- Linhai Song and Shan Lu. 2017. Performance Diagnosis for Inefficient Loops. In ICSE. 370–380.

Q&A?

Bihuan Chen, Pre-Tenure Assoc. Prof.

bhchen@fudan.edu.cn

https://chenbihuan.github.io