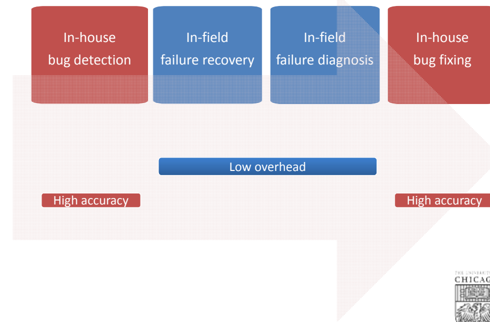
Fighting Software Inefficiency Through Automated Bug Detection

Shan Lu
University of Chicago

Different aspects of fighting bugs



How did this start?

- I worked on detecting bugs for many years
  - Memory bug detection
    - Monitor memory accesses & operations
    - Identify abnormal memory accesses

```
P = malloc (10);
P[100] = 'a';
```

  - Concurrency bug detection
    - Monitor memory accesses & synchronization
    - Identify abnormal memory accesses

```
if (P)          P=NULL;
  *P='a';
```

How did this start?

- One of our bug detectors is strangely slow
  - Why not profiling?
    - Lots of noises in profiling
    - Measuring cost not inefficiency
- My collaborator asks me:

  Why cannot you detect performance bugs?

Do performance bugs exist?

- **Performance bugs**:
  Bugs that cause severe & unnecessary **performance degradation** for some inputs

- Real-world incidents caused by performance bugs:
  - Example 1: Trend Micro (3 million USD, 650+ companies)
    - http://www.pcworld.com/article/120612/article.html
  - Example 2: Wikipedia servers stopped responding
    - http://dom.as/2009/06/26/embarrassment/
  - Example 3: Colorado Benefits System (200 million USD)
    - http://cais.aisnet.org/articles/16-34/journal.pdf
  - Example 4: UK Census site (1.9 million USD)
    - http://news.bbc.co.uk/2/hi/uk_news/2136572.stm

How should I start this research?

Story TiME

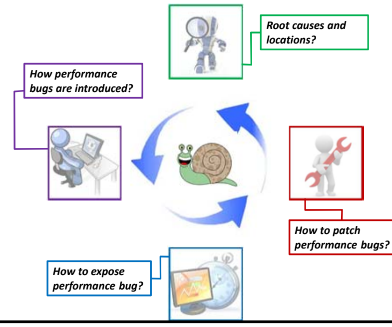
Slide 2

SL41 ideally, this should be a cycle, but …
Shan Lu, 2014-1-7

An Empirical Study

Are there performance bugs? How many?
What types of performance bugs are there?

What types of bugs are there?



Methodology

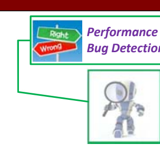
- Application and Bug Source

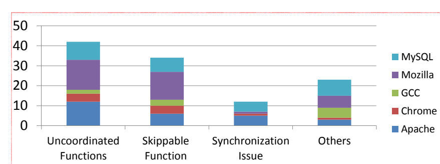
Application	Software Type	Language	MLOC	Bug DB History	Tags	# Bugs
Apache	Command-line Utility + Server + Library	C/Java	0.45	13 y	N/A	25
Chrome	GUI Application	C/C++	14.0	4 y	N/A	10
GCC	Compiler	C/C++	5.7	13 y	Compile-time-hog	10
Mozilla	GUI Application	C++/JS	4.7	14 y	perf	36
MySQL	Server Software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

Total: 109

- Threats to Validity

Root Causes of Performance Bugs

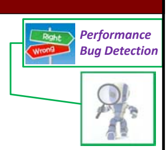


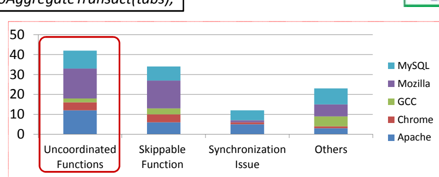


Root Causes of Performance Bugs

Mozilla Bug 490742 & Patch

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

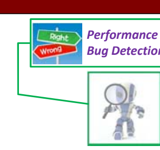


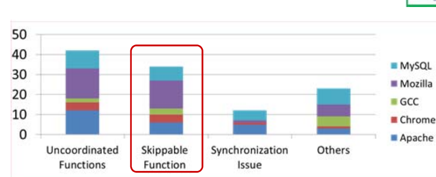


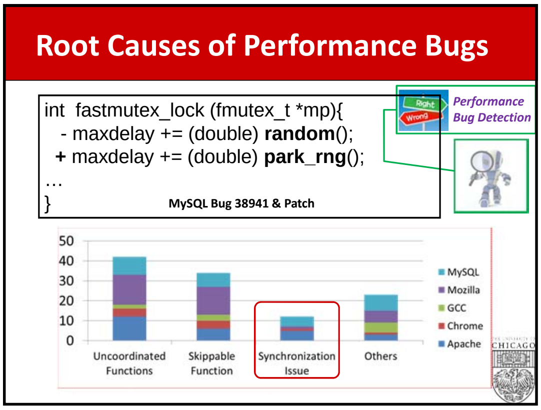
Root Causes of Performance Bugs

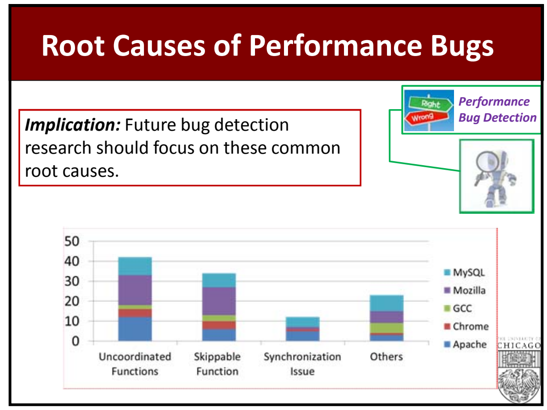
```
nsImage::Draw(…) {
  + if(mIsTransparent) return;
  …
}
```

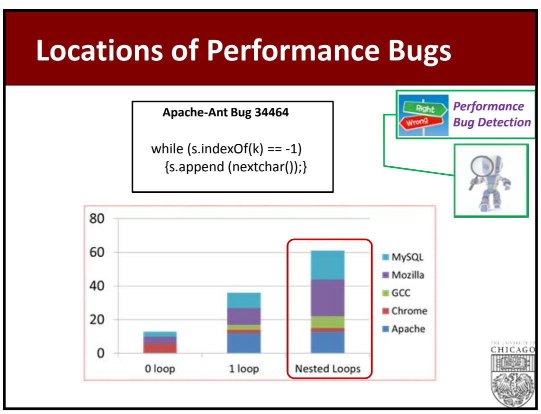
Mozilla Bug 66461

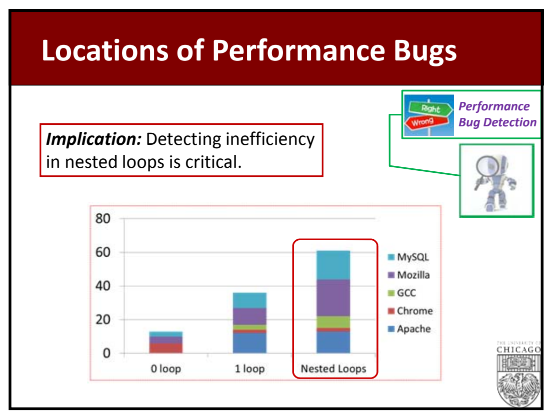


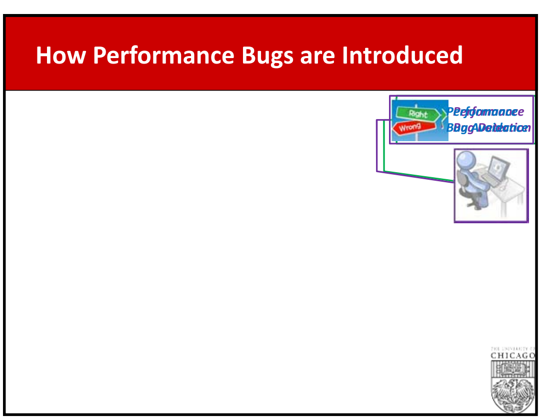


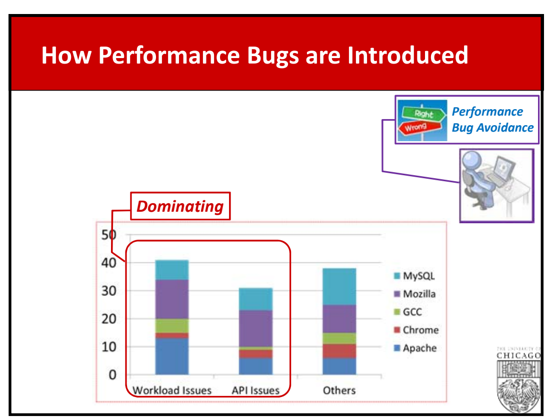

Root Causes of Performance Bugs
int fastmutex_lock (fmutex_t *mp){
- maxdelay += (double) random();
+ maxdelay += (double) park_rng();
…
}
MySQL Bug 38941 & Patch
Performance Bug Detection
50
40
30
20
10
0
Uncoordinated Functions
Skippable Function
Synchronization Issue
Others
MySQL
Mozilla
GCC
Chrome
Apache

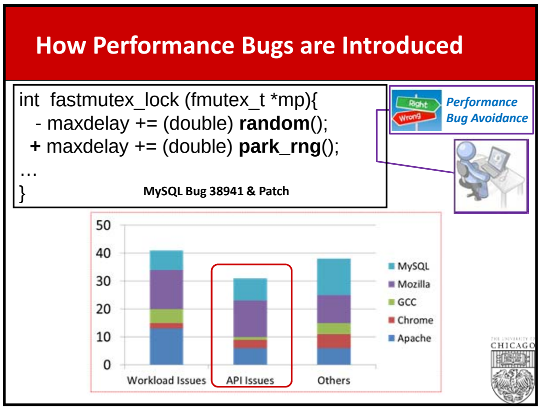

Root Causes of Performance Bugs
Implication: Future bug detection research should focus on these common root causes.
Performance Bug Detection
50
40
30
20
10
0
Uncoordinated Functions
Skippable Function
Synchronization Issue
Others
MySQL
Mozilla
GCC
Chrome
Apache

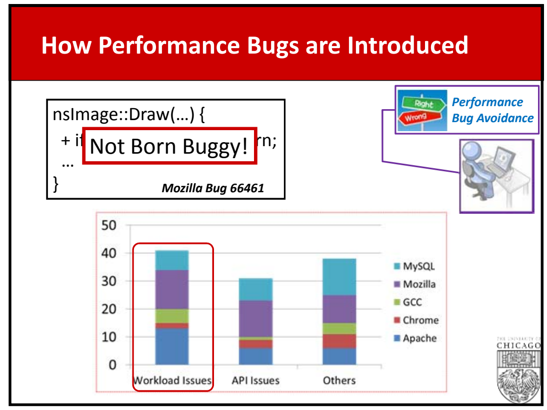

Locations of Performance Bugs
Apache-Ant Bug 34464
while (s.indexOf(k) == -1)
{s.append (nextchar());}
Performance Bug Detection
80
60
40
20
0
0 loop
1 loop
Nested Loops
MySQL
Mozilla
GCC
Chrome
Apache

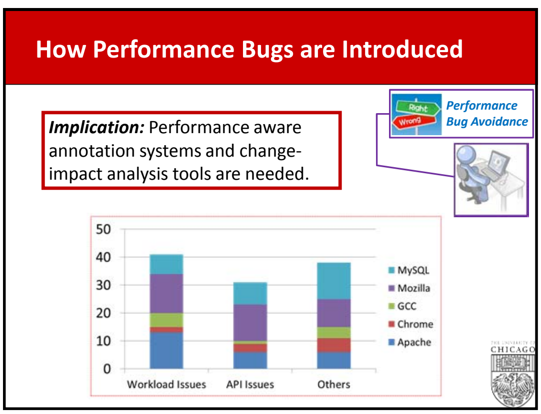

Locations of Performance Bugs
Implication: Detecting inefficiency in nested loops is critical.
Performance Bug Detection
80
60
40
20
0
0 loop
1 loop
Nested Loops
MySQL
Mozilla
GCC
Chrome
Apache

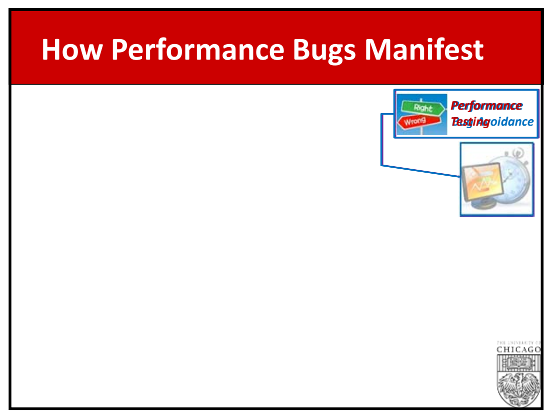

How Performance Bugs are Introduced

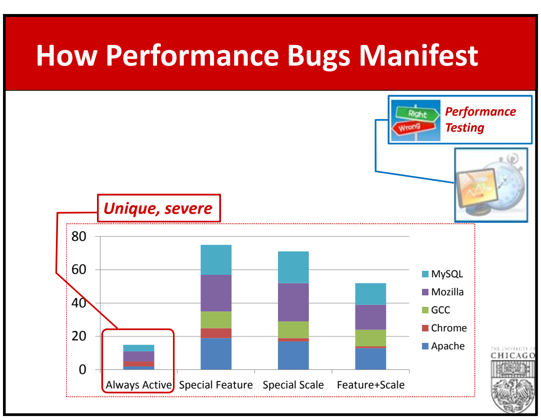

How Performance Bugs are Introduced
Performance Bug Avoidance
Dominating
50
40
30
20
10
0
Workload Issues
API Issues
Others
MySQL
Mozilla
GCC
Chrome
Apache

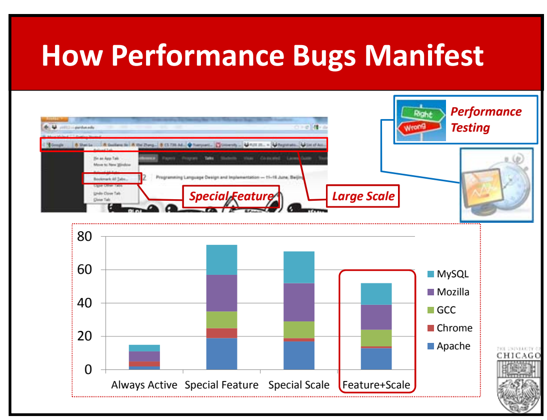

How Performance Bugs are Introduced
int fastmutex_lock (fmutex_t *mp){
- maxdelay += (double) random();
+ maxdelay += (double) park_rng();
…
}
MySQL Bug 38941 & Patch
Performance
Bug Avoidance
Workload Issues
API Issues
Others
MySQL
Mozilla
GCC
Chrome
Apache


How Performance Bugs are Introduced
nsImage::Draw(…) {
Not Born Buggy!
}
Mozilla Bug 66461
Performance
Bug Avoidance
Workload Issues
API Issues
Others
MySQL
Mozilla
GCC
Chrome
Apache


How Performance Bugs are Introduced
Implication: Performance aware annotation systems and change-impact analysis tools are needed.
Performance
Bug Avoidance
Workload Issues
API Issues
Others
MySQL
Mozilla
GCC
Chrome
Apache

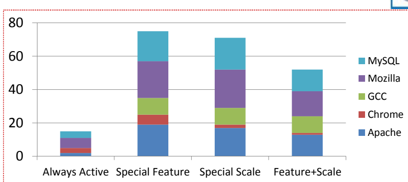

How Performance Bugs Manifest


How Performance Bugs Manifest
Performance
Testing
Unique, severe
Always Active
Special Feature
Special Scale
Feature+Scale
MySQL
Mozilla
GCC
Chrome
Apache


How Performance Bugs Manifest
Performance
Testing
Special Feature
Large Scale
Always Active
Special Feature
Special Scale
Feature+Scale
MySQL
Mozilla
GCC
Chrome
Apache

How Performance Bugs Manifest

Implication: New input generation tools are needed.

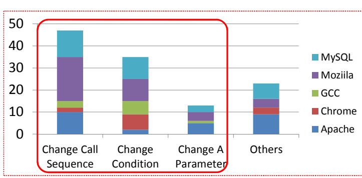


How Performance Bugs are Fixed

How Performance 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



What is next?

Can we detect performance bugs?
What "pattern" did we find?

Static inefficiency patterns exist

Apache-Ant Bug 34464

```
while (s.indexOf(k) == -1) {
    {s.append (nextchar());}
```

What pattern can you get from here?

Static inefficiency patterns exist

Mozilla Bug 490742

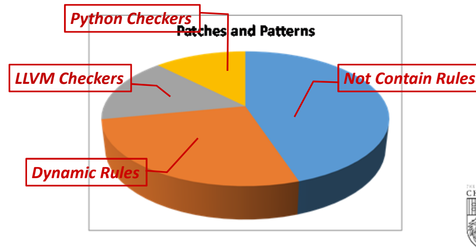
```
for (i = 0; i < tabs.length; i++) {
    ...
    tabs[i].doTransact();
}
```

What pattern can you get from here?

A Rule-Based Inefficiency Detector

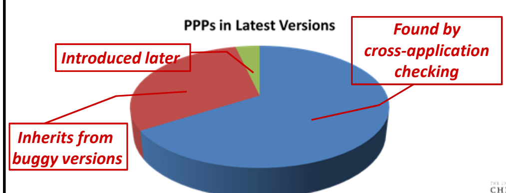
How to get these patterns?

- Manually extract from patches



Rule-Violation Detection Results

- 17 checkers find 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*

What is next?

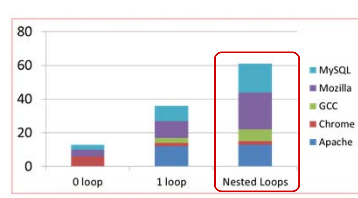
Do we have to manually specify rules?
Can we build generic detectors?

Toddler

A dynamic and generic detector targeting inefficient nested loops

Toddler: Detecting Performance Problems via Similar Memory-Access Patterns [ICSE '13]

What are generic inefficiency patterns?



Previous example

Apache-Ant Bug 34464

```
while (s.indexOf(k) == -1)
    {s.append (nextchar());}
```

↓

`Password: abcdefghi`

Another example in Java

- Previously **unknown** bug in Google Core Libraries

```
set.removeAll(arrayList);
public boolean removeAll(Collection<?> c) {
    if (someCondition) {
        for (Iterator<?> i = iterator(); i.hasNext(); ) { // Outer Loop
            if (c.contains(i.next())) {
                i.remove();
            }
        }
    }
}
public boolean contains(Object o) {
    for (int i = 0; i < size; i++) { // Inner Loop
        if (o.equals(elementData[i])) {
            return true;
        }
    }
}
```

What is the pattern?

What is the pattern?

- What type of nested loops are likely inefficient?
  - Many inner loops are *similar* with each other
    - Some instructions keeps reading similar sequences of values

```
abcdefg
abcdefgh
abcdefghi
```

Steps

- **Input:** Test code + system under test
- **Output:** Loops that are likely performance bugs
- **Steps:**
  1. Instrument the system under test
  2. Run the test with the instrumented code
  3. Analyze trace produced by instrumentation
  4. Detect work that is similar across loop iterations

Instrumentation

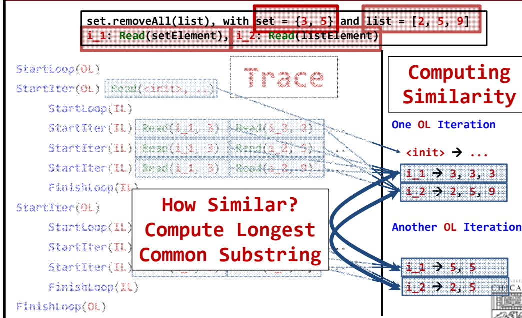
- Loop start
- Loop stop
- Iteration start
- Memory reads from fields
  - Value read
  - Instruction Pointer
  - Stack at the time of the read

Recall Example

- Previously **unknown** bug in Google Core Libraries

```
set.removeAll(arrayList);
public boolean removeAll(Collection<?> c) {
    ...
    for (Iterator<?> i = iterator(); i.hasNext(); ) { // Outer Loop
        if (c.contains(i.next())) {
            i.remove();
        }
    }
}
public boolean contains(Object o) {
    for (int i = 0; i < size; i++) { // Inner Loop
        if (o.equals(elementData[i])) {
            return true;
        }
    }
}
```

Collecting Trace and Computing Similarity



Algorithm

```
Input: trace of dynamic loops
Output: loops with similar iterations, if any

foreach dynamic loop dynLoop
  if dynLoop has more than minIteration iterations
    foreach instruction ins
      if ins appears in more than minSeqRatio(%) of all iterations
        vals = the values accessed by ins
        foreach pair of consecutive iterations consecIt in vals
          are the two iterations in consecIt similar?
        if more than minSimilarRatio(%) of consecIt are similar
          report BUG;

are the two iterations in a consecIt similar?
  lcs = Longest Common Substring between the iterations in consecIt
  if size of lcs is larger than minLCS and
     lcs larger than minLCSRatio(%) of the smallest of the two iterations
    return true
  return false
```

Ignoring Known Benign Patterns

- Values that don't change between iteration
  - for (…) { … if (**this.someField** < 5) … }
  - This is a very frequent pattern and does not indicate a bug
- Computation inside class initializers
  - Developers unlikely to optimize code executed infrequently
- Explicitly specified some fields and methods to ignore
  - Some supposed to have repetitive patterns:
    - Example: for (…) {… **this.cursor++** …}
  - Some typically considered benign by developers
    - Example: appending strings in a loop
  - Done only once for each library
  - Default: only 3 fields and 4 toString/append methods in JDK
    - 7 items for JDK (for almost 200,000 tests) appears reasonable

Evaluation Subjects and New Bugs

Application	Description	LOC	Known Bugs	New Bugs	Fixed	Confirmed
Ant	Build tool	109,765	1	8	1	0
Apache Collections	Collections library	51,516	1	20	10	4
Groovy	Dynamic language	136,994	1	2	2	0
Google Core Libraries	Collections library	156,004	2	10	1	2
JFreeChart	Chart framework	64,184	1	1	0	0
Jmeter	Load testing tool	86,549	1	0	0	0
Lucene	Text search engine	320,899	2	0	0	0
PDFBox	PDF framework	78,578	1	0	0	0
Solr	Search server	373,138	1	0	0	0
JDK standard library				2	0	0
JUnit testing framework				1	1	0
9 Apps + 2 Libs	**50,000 – 320,000**		**11**	**44**	**15**	**6**

- **11 real-world performance bugs**
- **Previously unknown bugs:** 44 found, 15 fixed, 6 confirmed

Toddler vs. HProf

Known Bug	Bug Detected?		False P.	Rank	Slowdown	
	TODD.	PROF	TODD.	PROF	TODD.	PROF
Ant	✓	✗	0	19.3	13.7	4.2
Apache Collections	✓	✓	0	1.0	10.0	2.1
Groovy	✓	✓	0	3.7	15.5	3.7
Google Core Libraries #1	✓	✓	0	1.8	9.0	3.8
Google Core Libraries #2	✓	✗	0	5.3	7.5	3.2
JFreeChart	✓	✗	0	53.7	13.4	8.8
JMeter	✓	✗	0	10.3	8.5	1.9
Lucene #1	✓	✗	0	7.7	6.8	2.5
Lucene #2	✓	✓	0	3.1	25.4	3.1
PDFBox	✓	✗	1	18.8	51.8	12.1
Solr	✓	✗	0	178.3	114.2	7.1
11	**11**	**4**	**1**	**n/a**	**15.9X**	**4.0X**

- Toddler finds **more bugs** with **fewer false positives** than profiler
- Overhead is higher than profiler, but still acceptable for testing

New Bugs and Performance Tests

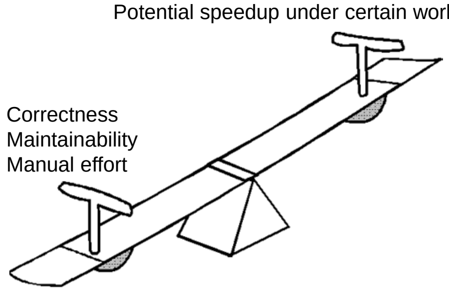
Who	App	Tests	Bugs	Bugs in Test	False Pos.
Auto	Ant	691	5	0	1
	Apache Collections	3,375	18	1	2
	Google Core Libraries	1,703	9	0	0
Expert	Apache Collections	60	10	0	1
	Google Core Libraries	60	2	0	0
Novices	Apache Collections	14	1	6	0
	Apache Collections	20	2	0	0
	Apache Collections	5	1	0	0
	Apache Collections	18	1	0	0
	Apache Collections	5	0	0	0
	Apache Collections	28	2	0	0
	Apache Collections	30	1	0	0
	Apache Collections	5	1	0	0
	Unique bugs:		**35**	**7**	**FPs: 4**

- Performance tests are easy to write even by novices
- Toddler finds **new real bugs with few false positives**

What is next?

Why so many bugs are not fixed by developers?

What are perf. bugs not fixed?



Can we detect bugs with simple fixes?

How can we detect bugs that developers are willing to fix?

Caramel

A static and generic detector targeting inefficient loops with simple patches

CARAMEL: Detecting and Fixing Performance Problems That Have Non-Intrusive Fixes [ICSE'15]

Won SIGSOFT Distinguished Paper Award

What is the pattern?

- What is a typical **simple** fix for an inefficient loop?

What is the pattern?

- What is a typical **simple** fix for an inefficient loop?

```
  for(…)
+     if (cond) break;
```

Results Overview

- 150 new bugs
- 116 bugs fixed
- Only 4 rejected
- 15 applications
- Auto. fixing 149/150 bugs

Example Bug Found By Caramel

- Non-Intrusive fix
- New bug in PDFBox, fixed by developers

```
boolean alreadyPresent = false;
while (isActualEmbeddedProperty.hasNext()) {
  if (alreadyPresent) break; // CondBreak FIX
  if (oldVal.getStr().equals(newVal.getStr()))
    alreadyPresent = true;
  if ( ! alreadyPresent )
    prop.container().addProp(newVal); // side effect
}
```

- Developers fix bugs that have CondBreak fixes:
  - Waste computation in loops
  - Fix is non-intrusive

What loops have CondBreak fixes?

- We thought for a loooong time …

What Bugs Have CondBreak Fixes?

How Is Computation Wasted?	Where Is Computation Wasted? Every Iteration	Late Iterations	Early Iterations
No-Result	Type 1	Type 2	Type Y
Useless-Result	Type X	Type 3	Type 4

What Bugs Have CondBreak Fixes?

How Is Computation Wasted?	Where Is Computation Wasted? Every Iteration	Late Iterations	Early Iterations
No-Result	Type 1 ?	Type 2 ?	Type Y
Useless-Result	Type X	Type 3 ?	Type 4 ?

Type ?

```
boolean alreadyPresent = false;
while (isActualEmbeddedProperty.hasNext()) {
  …
  if (oldVal.getStr().equals(newVal.getStr()))
    alreadyPresent = true;
  if ( ! alreadyPresent )
    prop.container().addProp(newVal); // side effect
}
```

Type ?

```
/* Copy the column definitions */

  memcpy((uchar*) recdef,(uchar*) share.rec,
       (size_t) (sizeof(MI_COLUMNDEF)*(share.base.fields+1)));

  for (rec=recdef,end=recdef+share.base.fields; rec != end ; rec++)
  {
    if (unpack && !(share.options & HA_OPTION_PACK_RECORD) &&
      rec->type != FIELD_BLOB &&
      rec->type != FIELD_VARCHAR &&
      rec->type != FIELD_CHECK)
    {
      rec->type=(int) FIELD_NORMAL;
    }
} // MySQL
```

Type ?

```
i = 0;

for (a = arglist; a; a = a->next)
 if (a->expr == NULL)
   i = 1;
```

Type ?

```
/*Are there any unended events of the same type? */

    for (i = 0; i < DTMFdec_inst->EventBufferSize; i++)
     {

          /* Going through the whole queue even when we have
found a match will ensure that we add to the latest applicable
event  */

          if ((DTMFdec_inst->EventQueue[i] == value) &&
(!DTMFdec_inst->EventQueueEnded[i] || endEvent))

               position = I;
     }
```

Ingredient 1: Result Instruction

```
boolean alreadyPresent = false;
while (isActualEmbeddedProperty.hasNext()) {
  if (alreadyPresent) break; // CondBreak FIX
  if (oldVal.getStr().
    alreadyPresent = t
  if ( ! alreadyPresent
    prop.container().addProp(newVal); // side effect
}
```

Result Instruction

Ingredient 2: Instruction-Condition

```
boolean alreadyPresent = false;
while (isActualEmbeddedProperty.hasNext()) {
  if (alr                                 k FIX
  if (old                                 r()))
    alread
  if ( ! alreadyPresent )
    prop.container().addProp(newVal); // Result Ins.
}
```

Instruction-Condition

66

Ingredient 3: Loop-Condition

- Condition under which **all RIs** do not produce results **for the remaining loop iterations**
- **Conjunction** of the Instruction-Conditions of **all RIs** in the loop

```
boolean alreadyPresent = false;
while (isActualEmbeddedProperty.hasNext()) {
  i                                        FIX
  i                                        ()))
    alreadyPresent = true;
  if ( !alreadyPresent )
    prop.container().addPro
}
```

Also Loop-Condition

Instruction-Condition

67

Type 1 RIs (Groovy)

Every No-result	Late No-Result	
	Late Useless	Early Useless

```
Class[] argTypes = ...
for (Iterator i = methods.iterator(); i.hasNext;) {
  if (!(argTypes==null)&&!(argTypes.length==0)) break;
    Node mn = (Meth          i.next();
    ero = (argTypes == null || argTypes.leng
  bool match = mn.getName().equals(methodName) && zero;
  if (match)
    return true; // Result Instruction
}
```

FALSE FALSE FALSE FALSE FALSE FALSE Not Execute

- **Ins.-Condition**:!(argTypes == null)&&!(argTypes.length==0)
- **Type 1**: If Instruction-Condition is true at beginning of loop
  - The RI is not executed → Category **No-Result**
  - In all iterations → Category **Every**

Type 2 RIs (PDFBox)

Every No-result	Late No-Result	
	Late Useless	Early Useless

```
boolean alreadyPresent = false;
while (isActualEmbeddedProperty.hasNext()) {
  if (alreadyPresent) break; // CondBreak FIX
  if (oldVal.getStr().equals(newVal.getStr()))
    alreadyPresent = true;
  if ( ! alreadyPresent )
    prop.container().addProp(newVal); // Result Ins
}
```

- **Instruction-Condition**: alreadyPresent == true
- **Type 2**: When Instruction-Condition becomes true
  - The RI is not executed → Category **No-Result**
  - In the remaining iterations → Category **Late**

Caramel Algorithm

- Static analysis
- Five steps:
  1. Detect all RIs (e.g., 3 RIs) ← Classical static analysis
  2. Fo
     - ce result)
     -
  3. Check all Instruction-Conditions satisfiable together
     - Checking that Loop-Condition is satisfiable
  4. Check loop not already exit when Loop-Condition
  5. Generate fix: if (Loop-Condition) break;

Details about each step in the paper

Evaluation Subjects and New Bugs

- **15 applications**
  - 11 **Java**, 4 **C/C++**
  - Google Chrome, GCC, Mozilla, Tomcat
- **150 new bugs**
- **116 bugs fixed**
  - 51 in Java
  - 65 in C/C++
- **Only 4** rejected
- **22 bugs in GCC fixed**
- 149/150 fixed automatically

Application	Description	LOC	Bugs
Ant	Build tool	140,674	1
Groovy	Dynamic language	161,487	9
JMeter	Load testing tool	114,645	4
Log4J	Logging framework	51,936	6
Lucene	Text search engine	441,649	14
PDFBox	PDF framework	108,796	10
Sling	Web app. framework	202,171	6
Solr	Search server	176,937	2
Struts	Web app. framework	175,026	4
Tika	Content extraction	50,503	1
Tomcat	Web server	295,223	4
Google Chrome	Web browser	13,371,208	22
GCC	Compiler	1,445,425	22
Mozilla	Web browser	5,893,397	27
MySQL	Database server	1,774,926	18

False Positives

- Three causes:
  1. Complex Analysis
  2. Concurrent
  3. Infrastructure
- Discussed in paper
- **Good ratio** of false positives / bugs

Application	Complex Aly.	Concurrent	Infrastructure
Ant	0	1	0
Groovy	0	0	0
JMeter	0	0	0
Log4J	0	2	0
Lucene	2	3	0
PDFBox	0	0	1
Sling	0	0	1
Solr	0	0	1
Struts	1	0	1
Tika	2	0	0
Tomcat	1	0	3
Google Chrome	0	0	0
GCC	1	0	0
Mozilla	2	0	0
MySQL	1	0	0
Total	23 false positives 150 bugs		

Conclusions

1. Novel perspective: Performance bugs that have non-intrusive fixes
2. Identify new family of performance bugs
3. Detection ← Static Analysis
4. Automated fixing
5. **116 bugs fixed**, 15 popular apps

What is next?

- Have we detected all performance bugs?
  – Absolutely not

Thanks!
Questions?

My collaborators

- Prof. Darko Marinov
- Adrian Nistor
- Linhai Song