Understanding and Detecting Real-World Performance Bugs

Guoliang Jin Linhai Song Xiaoming Shi Joel Scherpelz * Shan Lu
University of Wisconsin–Madison
{aliang, songlh, xiaoming, shanlu}@cs.wisc.edu joel.scherpelz@gmail.com

Abstract

Developers frequently use inefficient code sequences that could be fixed by simple patches. These inefficient code sequences can cause significant performance degradation and resource waste, referred to as performance bugs. Meager increases in single threaded performance in the multi-core era and increasing emphasis on energy efficiency call for more effort in tackling performance bugs.

This paper conducts a comprehensive study of 109 real-world performance bugs that are randomly sampled from five representative software suites (Apache, Chrome, GCC, Mozilla, and MySQL). The findings of this study provide guidance for future work to avoid, expose, detect, and fix performance bugs.

Guided by our characteristics study, efficiency rules are extracted from 25 patches and are used to detect performance bugs. 332 previously unknown performance problems are found in the latest versions of MySQL, Apache, and Mozilla applications, including 219 performance problems found by applying rules across applications.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging; D.4.8 [*Operating Systems*]: Performance

General Terms Languages, Measurement, Performance, Reliability

Keywords performance bugs, characteristics study, rule-based bug detection

1. Introduction

1.1 Motivation

Slow and inefficient software can easily frustrate users and cause financial losses. Although researchers have devoted decades to transparently improving software performance, *performance bugs* continue to pervasively degrade performance and waste computation resources in the field [40]. Meanwhile, current support for combating performance bugs is preliminary due to the poor understanding of real-world performance bugs.

Following the convention of developers and researchers on this topic [5, 26, 40, 50], we refer to performance bugs as software defects where relatively simple *source-code* changes can significantly speed up software, while preserving functionality. These defects can**not** be optimized away by state-of-practice compilers, thus bothering end users. Figure 1 shows an example of a real-world performance bug. Apache HTTPD developers forgot to change a parameter of API apr_stat after an API upgrade. This mistake caused more than ten times slowdown in Apache servers.

Patch for Apache Bug 45464	What is this bug
modules/dav/fs/repos.c	An Apache-API upgrade causes **apr_stat** to retrieve more information from the file system and take longer time.
status = apr_stat (fscontext->info, - APR_DEFAULT); + APR_TYPE);	Now, APR_TYPE retrieves exactly what developers originally needed through APR_DEFAULT.
Impact: causes httpd server **10+ times slower** in file listing	

Figure 1: A performance bug from Apache-HTTPD ('+' and '-' denote the code added and deleted to fix this bug)

Performance bugs exist widely in released software. For example, Mozilla developers have fixed 5–60 performance bugs reported by users *every month* over the past 10 years. The prevalence of performance bugs is inevitable because little work has been done to help developers avoid performance-related mistakes. In addition, performance testing mainly relies on ineffective black-box random testing and manual input design, which allows the majority of performance bugs to escape [40].

Performance bugs lead to reduced throughput, increased latency, and wasted resources in the field. In the past, they have caused several highly publicized failures, causing hundred-million dollar software projects to be abandoned [41, 45].

Worse still, performance problems are costly to diagnose due to their non fail-stop symptoms. Software companies may need several months of effort by experts to find a couple of performance bugs that cause a few hundred-millisecond delay in the 99th percentile latency of their service [49].

The following trends will make the performance-bug problem more critical in the future:

Hardware: For many years, Moore's law ensured that hardware would make software faster over time with no software development effort. In the multi-core era, when each core is unlikely to become faster, performance bugs are particularly harmful.

Software: The increasing complexity of software systems and rapidly changing workloads provide new opportunities for performance waste and new challenges in diagnosis [11].

Energy efficiency: Increasing energy costs provide a powerful economic argument for avoiding performance bugs. When one is willing to sacrifice the service quality to reduce energy consumption [3, 31], ignoring performance bugs is unforgivable. For exam-

* This work was done when the author was a graduate student at University of Wisconsin. Currently the author is working in NVIDIA.

PLDI'12, June 11–16, 2012, Beijing, China.

ple, by fixing bugs that have doubled the execution time, one may potentially halve the carbon footprint of buying and operating computers.

Performance bugs may not have been reported as often as functional bugs, because they do not cause fail-stop failures. However, considering the preliminary support for combating performance bugs, it is time to pay more attention to them when we enter a new resource-constrained computing world.

1.2 Contribution 1: Characteristics Study

Many empirical studies [7, 34, 43, 46, 53] have been conducted for traditional bugs that lead to incorrect software functionality, referred to as *functional bugs*. These studies have successfully guided the design of functional software testing, functional bug detection, and failure diagnosis.

Poor understanding of performance bugs and wrong perceptions, such as "*performance is taken care of by compilers and hardware*", are partly the causes of today's performance-bug problem [12]. The lack of empirical studies on topics like "how performance bugs are introduced", "what input conditions are necessary to expose performance bugs", "what the common root causes of real-world performance bugs are", and "how performance bugs are fixed by developers", have severely limited the design of performance-bug avoidance, testing, detection, and fixing tools.

This paper makes the first, to the best of our knowledge, comprehensive study of real-world performance bugs based on 109 bugs randomly collected from the bug databases of five representative open-source software suites (Apache, Chrome, GCC, Mozilla, and MySQL). Our study has made the following findings.

Guidance for bug avoidance. Two thirds of the studied bugs are introduced by developers' wrong understanding of workload or API performance features. More than one quarter of the bugs arise from previously correct code due to workload or API changes. To avoid performance bugs, developers need performance-oriented annotation systems and change-impact analysis. (Section 4.2).

Guidance for performance testing. Almost half of the studied bugs require inputs with **both** special features and large scales to manifest. New performance-testing schemes that combine the input-generation techniques used by functional testing [4, 17] with a consideration towards large scales will significantly improve the state of the art (Section 4.3).

Guidance for bug detection. Recent works [5, 11, 26, 47, 57, 58] have demonstrated the potential of performance-bug detection. Our study found common root-cause and structural patterns of real-world performance bugs that can help improve the coverage and accuracy of performance-bug detection (Sections 4.1 and 4.5).

Guidance for bug fixing and detection. Almost half of the examined bug patches include reusable efficiency rules that can help detect and fix performance bugs (Section 4.4).

Comparison with functional bugs. Performance bugs tend to hide for much longer time in software than functional bugs (Section 4.5). Unlike functional bugs, performance bugs cannot all be modeled as rare events, because a non-negligible portion of them can be triggered by almost all inputs (Section 4.3).

General motivation (1) Many performance-bug patches are small. The fact that we can achieve significant performance improvement through a few lines of code change motivates researchers to pay more attention to performance bugs (Section 4.4). (2) A non-negligible portion of performance bugs in multi-threaded software are related to synchronization. Developers need tool support to avoid over-synchronization traps (Section 4.3).

1.3 Contribution 2: Bug Detection

Rule-based bug detection is effective for detecting functional bugs [6, 21, 30, 42]. Following our characteristics study, we hypothesize that (1) efficiency-related rules exist; (2) we can extract rules from performance-bug patches; and (3) we can use the extracted rules to discover previously unknown performance bugs.

To test these hypotheses, we collected rules from 25 Apache, Mozilla, and MySQL bug patches and built static checkers to find violations to these rules.

Our checkers automatically found 125 *potential performance problems* (*PPP*s) in the original buggy versions of Apache, Mozilla, and MySQL. Programmers failed to fix them together with the original 25 bugs where the rules came from.

Our checkers also found 332 previously unknown PPPs in the latest versions of Apache, Mozilla, and MySQL. These include 219 PPPs found by checking an application using rules extracted from a *different* application.

Our thorough code reviews and unit testings confirm that each PPP runs significantly slower than its functionality-preserving alternate suggested by the checker. Some of these PPPs are already confirmed by developers and fixed based on our report.

The main contribution of our bug-detection work is that it confirms the existence and value of efficiency rules: efficiency rules in our study are usually violated at more than one place, by more than one developer, and sometimes in more than one program. Our experience motivates future work to automatically generate efficiency rules, through new patch languages [43], automated patch analysis [36], source code analysis, or performance-oriented annotations. Future work can also improve the accuracy of performance-bug detection by combining static checking with dynamic analysis and workload monitoring.

2. Methodology

2.1 Applications

Application Suite Description (language)	# Bugs
Apache Suite HTTPD: Web Server (C) TomCat: Web Application Server (Java) Ant: Build management utility (Java)	25
Chromium Suite Google Chrome browser (C/C++)	10
GCC Suite GCC & G++ Compiler (C/C++)	10
Mozilla Suite Firefox: Web Browser (C++, JavaScript) Thunderbird: Email Client (C++, JavaScript)	36
MySQL Suite Server: Database Server (C/C++) Connector: DB Client Libraries (C/C++/Java/.Net)	28
Total	109

Table 1: Applications and bugs used in the study

We chose five open-source software suites to examine: Apache, Chrome, GCC, Mozilla, and MySQL. These popular, award-winning software suites [22] are all large-scale and mature, with millions of lines of source code and well maintained bug databases.

As shown in Table 1, these five suites provide a good coverage of various types of software, such as interactive GUI applications, server software, command-line utilities, compilers, and libraries. They are primarily written in C/C++ and Java. Although they are all open-source software, Chrome is backed up by Google and MySQL was acquired by Sun/Oracle in 2008. Furthermore, the Chrome browser was first released in 2008, while the other four have had 10–15 years of bug reporting history. From these applications, we can observe both traditions and new software trends such as web applications.

2.2 Bug Collection

GCC, Mozilla, and MySQL developers *explicitly* mark certain reports in their bug databases as performance bugs using special tags,

which are *compile-time-hog*, *perf*, and *S5* respectively. Apache and Chrome developers do not use any special tag to mark performance bugs. Therefore, we searched their bug databases using a set of performance-related keywords ('slow', 'performance', 'latency', 'throughput', etc.).

From these sources, we *randomly* sampled 109 fixed bugs that have sufficient documentation. The details are shown in Table 1.

Among these bugs, 44 were reported after 2008, 38 were reported between 2004 and 2007, and 27 were reported before 2004. 41 bugs came from server applications and 68 bugs came from client applications.

2.3 Caveats

Our findings need to be taken with the methodology in mind. The applications in our study cover representative and important software categories, workload, development background, and programming languages. Of course, there are still uncovered categories, such as scientific computing software and distributed systems.

The bugs in our study are collected from five bug databases without bias. We have followed the decisions made by developers about what are performance bugs, and have not intentionally ignored any aspect of performance problems in bug databases. Of course, some performance problems may never be reported to the bug databases and some reported problems may never be fixed by developers. Unfortunately, there is no conceivable way to study these unreported or unfixed performance problems. We believe the bugs in our study provide a representative sample of the reported and fixed performance bugs in these representative applications.

We have spent more than one year to study all sources of information related to each bug, including forum discussions, patches, source code repositories, and others. Each bug is studied by at least two people and the whole process consists of several rounds of bug (re-)study, bug (re-)categorization, cross checking, etc.

Finally, we do not emphasize any quantitative characteristic results, and most of the characteristics we found are consistent across all examined applications.

3. Case Studies

The goal of our study is to improve software efficiency by inspiring better techniques to avoid, expose, detect, and fix performance bugs. This section uses four motivating examples from our bug set to demonstrate the feasibility and potential of our study. Particularly, we will answer the following questions using these examples:

(1) Are performance bugs too different from traditional bugs to study along the traditional bug-fighting process (i.e., bug avoidance, testing, detection, and fixing)? (2) If they are not too different, are they too similar to be worthy of a study? (3) If developers were more careful, do we still need research and tool support to combat performance bugs?

Transparent Draw (Figure 2) Mozilla developers implemented a procedure nsImage::Draw for figure scaling, compositing, and rendering, which is a waste of time for transparent figures. This problem did not catch developers' attention until two years later when 1 pixel by 1 pixel transparent GIFs became general purpose spacers widely used by Web developers to work around certain idiosyncrasies in HTML 4. The patch of this bug skips nsImage::Draw when the function input is a transparent figure.

Intensive GC (Figure 3) Users reported that Firefox cost 10 times more CPU than Safari on some popular Web pages, such as gmail.com. Lengthy profiling and code investigation revealed that Firefox conducted an expensive garbage collection process GC at the end of *every* XMLHttpRequest, which is too frequent. A developer then recalled that GC was added there five years ago when XHRs were infrequent and each XHR replaced substantial portions of the DOM in JavaScript. However, things have changed

Mozilla Bug 66461 & Patch	What is this bug

nsImage::Draw(...) {
...
+ if(mIsTransparent) return;
...
//render the input image
}
nsImageGTK.cpp
	When the input 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.
	The patch conditionally skips *Draw*.

Figure 2: A Mozilla bug drawing transparent figures

Mozilla Bug 515287 & Patch	What is this bug

XMLHttpRequest::OnStop(){
//at the end of each XHR
...
-- *mScriptContext->GC();*
}
nsXMLHttpRequest.cpp
	This was not a bug until Web 2.0, where doing garbage collection (*GC*) after every XMLHttpRequest (XHR) is too frequent.
	It causes Firefox to consume **10X more CPU** at idle GMail pages than Safari.

Figure 3: A Mozilla bug doing intensive GCs

Mozilla Bug 490742 & Patch	What is this bug
for (i = 0; i < tabs.length; i++) { ... - tabs[i].doTransact(); } + doAggregateTransact(tabs); *nsPlacesTransactionsService.js*	*doTransact* saves one tab into `bookmark' SQLite Database. **Firefox hangs** @ `bookmark all (tabs)'. The patch adds a new API to aggregate DB transactions.

Figure 4: A Mozilla bug with un-batched DB operations

MySQL Bug 38941 & Patch	What is this bug

int fastmutex_lock (fmutex_t *mp){
...
- maxdelay += (double) **random**();
+ maxdelay += (double) **park_rng**();
...
}
thr_mutex.c
	random() is a serialized global-mutex-protected glibc function.
	Using it inside `*fastmutex*' causes **40X slowdown** in users' experiments.

Figure 5: A MySQL bug with over synchronization

in modern Web pages. As a primary feature enabling web 2.0, XHRs are much more common than five years ago. This bug is fixed by removing the call to GC.

Bookmark All (Figure 4) Users reported that Firefox hung when they clicked 'bookmark all (tabs)' with 20 open tabs. Investigation revealed that Firefox used *N* database transactions to bookmark *N* tabs, which is very time consuming comparing with batching all bookmark tasks into a single transaction. Discussion among developers revealed that the database service library of Firefox did not provide interface for aggregating tasks into one transaction, because there was almost no batchable database task in Firefox a few years back. The addition of batchable functionalities such as 'bookmark all (tabs)' exposed this inefficiency problem. After replacing *N* invocations of doTransact with a single doAggregateTransact, the hang disappears. During patch review, developers found two more places with similar problems and fixed them by doAggregateTransact.

Slow Fast-Lock (Figure 5) MySQL synchronization-library developers implemented a fastmutex_lock for fast locking. Unfortunately, users' unit test showed that fastmutex_lock could be 40 times slower than normal locks. It turns out that library function random() actually contains a lock. This lock serializes every

threads that invoke random(). Developers fixed this bug by replacing random() with a non-synchronized random number generator.

These four bugs can help us answer the questions asked earlier.

(1) They have similarity with traditional bugs. For example, they are all related to usage rules of functions/APIs, a topic well studied by previous work on detecting functional bugs [30, 33].

(2) They also have interesting differences compared to traditional bugs. For example, the code snippets in Figure 2–4 turned buggy (or buggier) long after they were written, which is rare for functional bugs. As another example, testing designed for functional bugs cannot effectively expose bugs like *Bookmark All*. Once the program has tried the 'bookmark all' button with one or two open tabs, bookmarking more tabs will not improve the statement or branch coverage and will be skipped by functional testing.

(3) Developers cannot fight these bugs by themselves. They cannot predict future workload or code changes to avoid bugs like *Transparent Draw*, *Intensive GC*, and *Bookmark All*. Even experts who implemented synchronization libraries could not avoid bugs like *Slow Fast-Lock*, given opaque APIs with unexpected performance features. Research and tool support are needed here.

Of course, it is premature to draw any conclusion based on four bugs. Next, we will comprehensively study 109 performance bugs.

4. Characteristics Study

We will study the following aspects of real-world performance bugs, following their life stages and different ways to combat them.

1. What are the root causes of performance bugs? This study will provide a basic understanding of real-world performance bugs and give guidance to bug detection research.

2. How are performance bugs introduced? This study will shed light on how to avoid introducing performance bugs.

3. How can performance bugs manifest? This study can help design effective testing techniques to expose performance bugs after they are introduced into software.

4. How are performance bugs fixed? Answers to this question will help improve the patching process.

The result of this study is shown in Table 2.

4.1 Root Causes of Performance Bugs

There are a large variety of potential root causes for inefficient code, such as poorly designed algorithms, non-optimal data structures, cache-unfriendly data layouts, etc. Our goal here is **not** to discover previously unheard-of root causes, but to check whether there are common root-cause patterns among real-world performance bugs that bug detection can focus on.

Our study shows that the majority of real-world performance bugs in our study are covered by only a couple of root-cause categories. Common patterns do exist and performance is mostly lost at call sites and function boundaries, as follows.

Uncoordinated Functions More than a third of the performance bugs in our study are caused by inefficient function-call combinations composed of efficient individual functions. This occurs to the *Bookmark All* example shown in Figure 4. Using doTransact to bookmark one URL in one database transaction is efficient. However, bookmarking *N* URLs using *N* separate transactions is less efficient than calling doAggregateTransact to batch all URL bookmarkings in one transaction.

Skippable Function More than a quarter of bugs are caused by calling functions that conduct unnecessary work given the calling context, such as calling nsImage::Draw for transparent figures in the *Transparent Draw* bug (Figure 2) and calling unnecessary GCs in the *Intensive GC* bug (Figure 3).

Synchronization Issues Unnecessary synchronization that intensifies thread competition is also a common cause of performance loss, as shown in the *Slow Fast-Lock* bug (Figure 5). These bugs are especially common in server applications, contributing to 4 out of 15 Apache server bugs and 5 out of 26 MySQL server bugs.

Others The remaining 23 bugs are caused by a variety of reasons. Some use wrong data structures. Some are related to hardware architecture issues. Some are caused by high-level design/algorithm issues, with long propagation chains. For example, MySQL39295 occurs when MySQL mistakenly invalidates the query cache for read-only queries. This operation itself does not take time, but it causes cache misses and performance losses later. This type of root cause is especially common in GCC.

4.2 How Performance Bugs Are Introduced

We have studied the discussion among developers in bug databases and checked the source code of different software versions to understand how bugs are introduced. Our study has particularly focused on the challenges faced by developers in writing efficient software, and features of modern software that affect the introduction of performance bugs.

Our study shows that developers are in a great need of tools that can help them avoid the following mistakes.

Workload Mismatch Performance bugs are most frequently introduced when developers' workload understanding does not match with the reality.

Our further investigation shows that the following challenges are responsible for most workload mismatches.

Firstly, the input paradigm could shift *after* code implementation. For example, the HTML standard change and new trends in web-page content led to *Transparent Draw* and *Intensive GC*, shown in Figure 2 and Figure 3.

Secondly, software workload has become much more diverse and complex than before. A single program, such as Mozilla, may face various types of workload issues: the popularity of transparent figures on web pages led to *Transparent Draw* in Figure 2; the high frequency of XMLHttpRequest led to *Intensive GC* in Figure 3; users' habit of not changing the default configuration setting led to Mozilla Bug110555.

The increasingly dynamic and diverse workload of modern software will lead to more performance bugs in the future.

API Misunderstanding The second most common reason is that developers misunderstand the performance feature of certain functions. This occurs for 31 bugs in our study.

Sometimes, the performance of a function is sensitive to the value of a particular parameter, and developers happen to use performance-hurting values.

Sometimes, developers use a function to perform task *i*, and are unaware of an irrelevant task *j* conducted by this function that hurts performance but not functionality. For example, MySQL developers did not know the synchronization inside random and introduced the *Slow Fast-Lock* bug shown in Figure 5.

Code encapsulation in modern software leads to many APIs with poorly documented performance features. We have seen developers explicitly complain about this issue [51]. It will lead to more performance bugs in the future.

Others Apart from workload issues and API issues, there are also other reasons behind performance bugs. Interestingly, some performance bugs are side-effects of functional bugs. For example, in Mozilla196994, developers forgot to reset a busy-flag. This semantic bug causes an event handler to be constantly busy. As a result, a performance loss is the only externally visible symptom of this bug.

When a bug was not buggy An interesting trend is that 29 out of 109 bugs were not born buggy. They became inefficient long after they were written due to workload shift, such as that in *Transparent Draw* and *Intensive GC* (Figures 2 and 3), and code changes in other part of the software, such as that in Figure 1.

	Apache	Chrome	GCC	Mozilla	MySQL	Total
Number of bugs studied	25	10	10	36	28	109
Root Causes of Performance Bugs						
Uncoordinated Functions: function calls take a detour to generate results	12	4	2	15	9	42
Skippable Function: a function call with un-used results	6	4	3	14	7	34
Synchronization Issues: inefficient synchronization among threads	5	1	0	1	5	12
Others: all the bugs not belonging to the above three categories	3	1	5	6	8	23
How Performance Bugs Are Introduced						
Workload Issues: developers' workload assumption is wrong or out-dated	13	2	5	14	7	41
API Issues: misunderstand performance features of functions/APIs	6	3	1	13	8	31
Others: all the bugs not belonging to the above two categories	6	5	4	10	13	38
How Performance Bugs Are Exposed						
Always Active: almost every input on every platform can trigger this bug	2	3	0	6	4	15
Special Feature: need special-value inputs to cover specific code regions	19	6	10	22	18	75
Special Scale: need large-scale inputs to execute a code region many times	17	2	10	23	19	71
Feature+Scale: the intersection of Special Feature and Special Scale	13	1	10	15	13	52
How Performance Bugs Are Fixed						
Change Call Sequence: a sequence of function calls reorganized/replaced	10	2	3	20	12	47
Change Condition: a condition added or modified to skip certain code	2	7	6	10	10	35
Change A Parameter: changing one function/configuration parameter	5	0	1	4	3	13
Others: all the bugs not belonging to the above three categories	9	3	0	4	7	23

Table 2: Categorization for Sections 4.1 – 4.4 (most categories in each section are not exclusive)

In Chrome70153, when GPU accelerator became available, some software rendering code became inefficient. Many of these bugs went through regression testing without being caught.

4.3 How Performance Bugs Are Exposed

We define *exposing a performance bug* as causing a *perceivably* negative performance impact, following the convention used in most bug reports.

Our study demonstrates several unique challenges for performance testing.

Always Active Bugs A non-negligible portion of performance bugs are almost always active. They are located at the start-up phase, shutdown phase, or other places that are exercised by almost all inputs. They could be very harmful in the long term, because they waste performance at every deployment site during every run of a program. Many of these bugs were caught during comparison with other software (e.g., Chrome vs. Mozilla vs. Safari).

Judging whether performance bugs have manifested is a unique challenge in performance testing.

Input Feature & Scale Conditions About two thirds of performance bugs need inputs with special features to manifest. Otherwise, the buggy code units cannot be touched. Unfortunately, this is not what black-box testing is good at. Much manual effort will be needed to design test inputs, a problem well studied by past research in functional testing [4, 5].

About two thirds of performance bugs need large-scale inputs to manifest in a perceivable way. These bugs cannot be effectively exposed if software testing executes each buggy code unit only once, which unfortunately is the goal of most functional testing.

Almost half of the bugs need inputs that have special features **and** large scales to manifest. For example, to trigger the bug shown in Figure 4, the user has to click 'bookmark all' (i.e., special feature), with many open tabs (i.e., large scale).

4.4 How Performance Bugs Are Fixed

We have manually checked the final patches to answer two questions. Are there common strategies for fixing performance bugs? How complicated are performance patches?

The result of our study is opposite to the intuition that performance patches must be complicated and lack common patterns.

Fixing strategies There are three common strategies in fixing performance bugs, as shown in Table 2.

The most common one is to change a function-call sequence, referred to as *Change Call Sequence*. It is used to fix 47 bugs. Many bugs with uncoordinated function calls are fixed by this strategy (e.g., *Bookmark All* in Figure 4). Some bugs with skippable function calls are also fixed by this strategy, where buggy function calls are removed or relocated (e.g. *Intensive GC* in Figure 3).

The second most common strategy is *Change Condition*. It is used in 35 patches, where code units that do not always generate useful results are conditionally skipped. For example, Draw is conditionally skipped to fix *Transparent Draw* (Figure 2).

Finally, 13 bugs are fixed by simply changing a parameter in the program. For example, the Apache bug shown in Figure 1 is fixed by changing a parameter of `apr_stat`; MySQL45475 is fixed by changing the configuration parameter `TABLE_OPEN_CACHE_MIN` from 64 to 400.

Are patches complicated? Most performance bugs in our study can be fixed through simple changes. In fact, 42 out of 109 bug patches contain five or fewer lines of code changes. The median patch size for all examined bugs is 8 lines of code.

The small patch size is a result of the above fixing strategies. *Change A Parameter* mostly requires just one line of code change. 33 out of the 47 *Change Call Sequence* patches involve only existing functions, with no need to implement new functions. Many *Change Condition* patches are also small.

4.5 Other Characteristics

Life Time We chose Mozilla to investigate the life time of performance bugs, due to its convenient CVS query interface. We consider a bug's life to have started when its buggy code was first written. The 36 Mozilla bugs in our study took 935 days on average to get discovered, and another 140 days on average to be fixed. For comparison, we randomly sampled 36 functional bugs from Mozilla. These bugs took 252 days on average to be discovered, which is much shorter than that of performance bugs in Mozilla.

These bugs took another 117 days on average to be fixed, which is a similar amount of time with those performance bugs.

Location For each bug, we studied the location of its minimum unit of inefficiency. We found that over three quarters of bugs are located inside either an input-dependent loop or an input-event handler. For example, the buggy code in Figure 3 is executed at every XHR completion. The bug in Figure 2 wastes performance for every transparent image on a web page. About 40% of buggy code units contain a loop whose number of iterations scales with input. For example, the buggy code unit in Figure 4 contains a loop that iterates as many times as the number of open tabs in the browser. In addition, about half performance bugs involve I/Os or other time-consuming system calls. There are a few bugs whose buggy code units only execute once or twice during each program execution. For example, the Mozilla110555 bug wastes performance while processing exactly two fixed-size default configuration files, userChrome.css and userContent.css, during the startup of a browser.

Correlation Among Categories Following previous empirical studies [29], we use a statistical metric *lift* to study the correlation among characteristic categories. The *lift* of category A and category B, denoted as *lift(AB)*, is calculated as $\frac{P(AB)}{P(A)P(B)}$, where P(AB) is the probability of a bug belonging to both categories A and B. When *lift(AB)* equals 1, category A and category B are independent with each other. When *lift(AB)* is greater than 1, categories A and B are positively correlated: when a bug belongs to A, it likely also belongs to B. The larger the *lift* is, the more positively A and B are correlated. When *lift(AB)* is smaller than 1, A and B are negatively correlated: when a bug belongs to A, it likely does not belong to B. The smaller the *lift* is, the more negatively A and B are correlated.

Among all categories, the *Skippable Function* root cause and the *Change Condition* bug-fix strategy are the most positively correlated with a 2.02 lift. The *Workload Issues* bug-introducing reason is strongly correlated with the *Change-A-Parameter* bug-fix strategy with a 1.84 lift. The *Uncoordinated Functions* root cause and the *API Issues* bug-introducing reason are the third most positively correlated pair with a 1.76 lift. On the other hand, the *Synchronization Issues* root cause and the *Change Condition* fix strategy are the most negatively correlated categories of different characteristic aspects[1]. Their lift is only 0.26.

Server Bugs vs. Client Bugs Our study includes 41 bugs from server applications and 68 bugs from client applications. To understand whether these two types of bugs have different characteristics, we apply chi-square test [56] to each category listed in Table 2. We choose 0.01 as the significance level of our chi-square test. Under this setting, if we conclude that server and client bugs have different probabilities of falling into a particular characteristic category, this conclusion only has 1% probability to be wrong.

We find that, among all the categories listed in Table 2, only the *Synchronization Issues* category is significantly different between server bugs and client bugs — *Synchronization Issues* have caused 22% of server bugs and only 4.4% of client bugs.

5. Lessons from Our Study

Comparison with Functional Bugs There are several interesting comparisons between performance and functional bugs. (1) The distribution of performance-failure rates over software life time follows neither the bathtub model of hardware errors nor the gradually maturing model of functional bugs, because performance bugs have long hiding periods (Section 4.5) and can emerge from non-buggy places when software evolves (Section 4.2). (2) Unlike functional bugs, performance bugs cannot always be modeled as rare events, because some of them are always active (Section 4.3). (3) The percentage of synchronization problems among performance bugs in our study is higher than the percentage of synchronization problems among functional bugs in a previous study for a similar set of applications [29] (Section 4.1).

Bug Detection Our study motivates future research in performance-bug detection: performance bugs cannot be easily avoided (Section 4.2); and, they can escape testing even when the buggy code is exercised (Section 4.3).

Our study provides future bug detectors with common root cause and location patterns (Section 4.1 and Section 4.5).

Rule-based bug detection [13, 21, 30, 33] are promising for detecting performance bugs. It will be discussed in Section 6 in detail. Invariant-based bug detection and delta debugging [14, 60] are also promising. Our study shows that the majority of performance bugs require special inputs to manifest (Section 4.3). In addition, the same piece of code may behave buggy and non-buggy in different software versions (Section 4.2). This provides opportunities for invariant extraction and violation checking.

Annotation Systems Annotation systems are used in many software development environments [37, 54]. Unfortunately, they mainly communicate functionality information.

Our study calls for performance-aware annotation systems [44, 55] that help developers maintain and communicate APIs' performance features and workload assumptions (Section 4.2). Simple support such as warning about the existence of locks in a library function, specifying the complexity of a function, and indicating the desired range of a performance-sensitive parameter can go a long way in avoiding performance bugs. Recent work that automatically calculates function complexity is also promising [18].

Testing Regression testing and change-impact analysis have to consider workload changes and performance impacts, because new performance bugs may emerge from old code (Section 4.2).

Performance testing can be improved if its input design combines smart input-generation techniques used in functional testing [4, 17] with an emphasis on large scale (Section 4.3).

Expressing performance oracles and judging whether performance bugs have occurred are critical challenges in performance testing (Section 4.3). Techniques that can smartly compare performance numbers across inputs and automatically discover the existence of performance problems are desired.

Diagnosis Profiling is frequently used to bootstrap performance diagnosis. Our root-cause study shows that extra analysis is needed to help diagnose performance bugs. It is difficult to use profiling alone to locate the root cause of a performance bug that has a long propagation chain or wastes computation time at function boundaries, which is very common (Section 4.1).

Future Directions One might argue that performance sometimes needs to be sacrificed for better productivity and functional correctness. However, the fact that we can often achieve significant performance improvement through only a few lines of code change motivates future research to pay more attention to performance bugs (Section 4.4). Our study suggests that the workload trend and API features of modern software will lead to more performance bugs in the future (Section 4.2). In addition, our study observes a significant portion of synchronization-related performance bugs in multi-threaded software. There will be more bugs of this type in the multi-core era.

Finally, our observations have been consistent across old software and new software (Chrome), old bugs (27 pre-2004 bugs) and new bugs (44 post-2008 bugs). Therefore, we are confident that these lessons will be useful at least for the near future.

[1] Two categories of the same aspect, such as the *Skippable Function* root cause and the *Uncoordinated Functions* root cause, usually have a highly negative correlation.

6. Rule-Based Performance-Bug Detection

6.1 Overview

Rule-based detection approach is effective for discovering functional bugs and security vulnerabilities [6, 15, 21, 30, 42]. Many functional bugs can be identified by comparing against certain function-call sequences that have to be followed in a program for functional correctness and security.

We hypothesize that rule-based bug detection is useful for detecting performance bugs based on our characteristics study:

Efficiency rules should exist. Those inefficient function-call sequences studied in Section 4.1 could all become rules. For example, random() should not be used by concurrent threads, and doTransact() in loops should be replaced by aggregateTransact().

Efficiency rules can be easily collected from patches, as most patches are small and follow regular fixing strategies (Section 4.4).

Efficiency rules could be widely applicable, as a misunderstanding of an API or workload could affect many places and lead to many bugs, considering how bugs are introduced (Section 4.2).

This section will test our hypothesis and provide guidance for future work on combating performance bugs.

6.2 Efficiency Rules in Patches

Terminology *Efficiency rules*, or *rules*, include two components: a *transformation* and a *condition* for applying the transformation. Once a code region satisfies the condition, the transformation can be applied to improve performance and preserve functionality.

We have manually checked all the 109 performance-bug patches. 50 out of these 109 patches contain efficiency rules, coming from all five applications. The other 59 do not contain rules, because they either target too specific program contexts or are too general to be useful for rule-based bug detection.

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

Table 3: Typical conditions in function rules

Most of these 50 rules, according to the *lift* correlation metric, are related to the *Uncoordinated Functions* root cause and the *Change Call Sequence* fix strategy. The conditions for applying these rules are composed of conditions on function-call sequences, parameter/return variables, and calling contexts, as shown in Table 3. For example, to apply the *Bookmark All* patch in Figure 4 elsewhere, one needs to find places that call doTransact inside a loop; to apply the patch in Figure 1 elsewhere, one needs to ensure that certain fields of the object pointed by the first parameter of apr_stat is not used afterward. There are also non-function rules, usually containing *Change Condition* transformation and other miscellaneous algorithm improvements.

6.3 Building Rule Checkers

Selecting Statically Checkable Rules Some rules' applying conditions are statically checkable, such as function f1 inside a loop; some are dynamically checkable, such as function f1 called by multiple threads at the same time; some are related to workload, such as having many large input files.

We check three largest application suites in our study: Apache, MySQL, and Mozilla. We find that 40 bug patches from them contain rules. 25 out of these 40 have applying conditions that are mostly statically checkable. Therefore, we have built checkers based on these **25** efficiency rules.

Checker Implementation We build 25 checkers in total. 14 of them are built using LLVM compiler infrastructure [27] for rules from C/C++ applications. LLVM works well for C++ software that troubles many other static analysis infrastructure [43]. It also provides sufficient data type, data flow, and control flow analysis support for our checking. The other 11 checkers are written in Python for 11 rules from Java, JavaScript, and C# applications.

The checker implementation is mostly straightforward. Each checker goes through software bitcode, in case of LLVM checkers, or source code, in case of Python checkers, looking for places that satisfy the patch-applying condition. We briefly discuss how our checkers examine typical conditions for function rules in the following.

Checking call-sequence conditions, exemplified in Table 3, involve mainly three tasks: (1) Differentiating functions with the same name but different classes; (2) Collecting loop information (loop-head, loop-exit conditions, loop-body boundaries, etc.); (3) Control flow analysis. LLVM provides sufficient support for all these tasks. Checkers written in Python struggle from time to time.

Checking parameter/return conditions, exemplified in Table 3, typically rely on data-flow analysis. In our current prototype, LLVM checkers conduct intra-procedural data-flow analysis. This analysis is scalable, but may lead to false positives and negatives. In practice, it works well as shown by our experimental results. Our current Python checkers can extract parameters of particular function calls, but can only do preliminary data-flow analysis.

6.4 Rule-Checking Methodology

We conduct all the experiments on an 8-core Intel Xeon machine running Linux version 2.6.18.

We apply every checker to the following software:

(1) The exact version of the software that the original patch was applied to, which is referred to as *original version*;

(2) The latest version of the software that the original patch was applied to, which is referred to as *original software*;

(3) The latest versions of software applications that are **different** from the one that the original patch was applied to, which is referred to as *different software*. This was applied to 13 checkers, whose rules are about *glibc* library functions, Java library functions, and some general algorithm tricks. We will refer to this as *cross-application checking*. For example, a C/C++ checker from MySQL will be applied to Mozilla and Apache HTTPD for cross-application checking; a Java checker from Apache TomCat server will be applied to the 65 other Java applications in the Apache software suite[2].

The checking results are categorized into three types: *PPPs*, *bad practices*, and *false positives*. As discussed in Section 1.3, a PPP is an inefficient code region that runs slower than its functionality-preserving alternate implied by the efficiency rule. A bad practice is a region prone to becoming inefficient in the future. We reported some PPPs to developers. Among those reported, 14 PPPs detected by 6 different checkers have been confirmed and fixed by the

[2] Development teams behind different Apache applications are different

developers. Other reported PPPs are put on hold due to lack of bug-triggering input information, which is unfortunately out of the scope of this work.

Finally, we have also changed each checker slightly to report code regions that follow each efficiency rule. We refer to these regions as *good practices*, the opposite of PPPs.

6.5 Rule-Checking Results

Overall Results As shown in Table 4, 125 PPPs are found in the *original version* of software. Programmers missed them and failed to fix them together with the original bugs.

113 previously unknown PPPs are found in the latest versions of the *original software*, including bugs inherited from the original version and bugs newly introduced. Figure 6 shows an example.

219 previously unknown PPPs are found in the latest versions of *different software*. An example is shown in Figure 6.

14 PPPs in the latest versions of Apache, Mozilla, and MySQL are already confirmed and fixed by developers based on our report.

These results confirm that performance bugs widely exist. Efficiency rules exist and are useful for finding performance problems.

PPPs In Original Versions 17 out of 25 checkers found new PPPs, 125 in total, in the original versions of the buggy software.

Some developers clearly tried to find all similar bugs when fixing one bug, but did not succeed. For example, in MySQL14637, after two buggy code regions were reported, developers found three more places that were similarly inefficient and fixed them altogether. Unfortunately, there were another 50 code regions that violated the same efficiency rule and skipped developers' checking, as shown in Table 4. Similarly, MySQL developers found and fixed 3 places that had the inefficiency pattern shown in Figure 6, but missed the other 15 places.

113 out of these 125 PPPs exist in different files or even different modules where the original bugs exist, which is probably why they were missed by developers. These PPPs end up in several ways: (1) 4 of them were fixed in later versions, which took 14–31 months; (2) 20 eventually disappeared, because the functions containing these PPPs were removed or re-implemented; (3) 101 still exist in the latest versions of the software, wasting computation resources 12–89 months after the original bugs were fixed.

Lesson The above results show that developers do need support to systematically and automatically find similar performance bugs and fix them all at once.

PPPs In The Latest Versions 2 of the 25 checkers are no longer applicable in the latest versions, because the functions involved in these checkers have been removed. The remaining 23 checkers are applied to the latest versions of corresponding software and find 113 PPPs. Among them, 101 PPPs were inherited from the original buggy versions. The other 12 were introduced later.

Lesson Developers cannot completely avoid the mistakes they made and corrected before, which is understandable considering the large number of bugs in software. Specification systems and automated checkers can prevent developers from introducing old bugs into new code.

PPPs In Different Software Applications An exciting result is that 8 out of 13 cross-application checkers have successfully found previously unknown PPPs in the latest versions of applications that are different from where the rules came from.

Most of these checkers reflect common pitfalls in using library functions. For example, Figure 6 shows a pitfall of using String::indexof(). Apache-Ant developers made this mistake, and we found Apache-Struts developers also made a similar mistake.

Apache32546 checker presents an interesting case. In the original bug report, developers from Apache-Slide recognized that a small buffer size would severely hurt the performance of java.io.InputStream.read (byte buffer[]) for reasonably large input (e.g., larger than 50KB). Replacing their original 2KB buffer with a 200KB buffer achieved **80 times** throughput improvement in WebDav server. We first confirmed that this rule is still valid. Our checker then found 135 places in the latest versions of 36 software applications where similar mistakes were made. These places use small buffers (1KB – 4KB) to read images or data files from disk or web, and are doomed to performance losses.

Some checkers reflect algorithm improvements and are also applicable to many applications. For example, algorithm improvements for string operations proposed by MySQL developers (MySQL14637 and MySQL49491) also apply for Mozilla and Apache HTTPD.

Cross-application checking also helps validate efficiency rules. For example, by comparing how java.util.zip.Deflater.deflate() is used across applications, we found that Ant developers' understanding of this API, reflected by their discussion, was wrong. They fixed Apache45396 by coincidence.

Lesson The above results show that there exist general inefficiency patterns that go beyond one application, just like that for functional bugs [21]. Maintaining specifications and checkers for these general patterns can significantly save developers' effort, and allow them to learn from other developers and other software. We can even discover performance bugs in a software where no performance patch has ever been filed.

Bad Practices Other than PPPs, some code regions identified by the checkers are categorized as bad practices. For example, there are code regions very similar to the MySQL PPP shown in Figure 6, except that the calculation of end is not completely useless as end is used in places other than the invocation of ismbchar. Clearly this practice is more likely to cause performance problems in the future than directly using mysqlcs→mbmaxlen as the parameter for ismbchar function.

Good Practices Code regions that have well followed the efficiency rules are also identified by slightly changed checkers. For example, we found that in 13 places of various applications developers do use InputStream.read (byte buffer[]) in a performance efficient way: buffer has a configurable size or a large size that suits the workload (e.g., 64K in some Hadoop code).

Lesson Violations to efficiency rules are not always rare comparing with good practices. Previous techniques that use statistical analysis to infer functional rules [13, 30] may not work for efficiency rules.

False Positives Our PPP detection is accurate. On average, the false-positive-vs-PPP rate is 1:4. The false positives mainly come from three sources.

First, Python checkers have no object-type information. Therefore, some rules are applied to functions with right function names but wrong classes (e.g., Mozilla490742 and Apache32546). This is not a problem in LLVM checkers.

Second, some non-function rules are difficult to accurately express and check, which leads to false positives in MySQL14637.

Third, accurately checking some efficiency rules requires runtime and/or workload information, which inevitably leads to false positives in our static checkers. False positives in Apache44408 and Apache48778 mostly belong to this category. These false positives can be largely eliminated by run-time checkers.

Performance Results Our checkers are efficient. Each Python checker finishes checking 10 million lines of code within 90 seconds. Our LLVM checkers are mainly applied to MySQL, Mozilla Firefox, and Apache HTTPD. It takes 4 – 1270 seconds for one LLVM checker to process one application.

We tried unit testing on PPPs. The performance difference is significant. For example, for programs that read images and files

ID	Orig. Buggy Version				Lastest Version of Same Softw.				Latest Version of Diff. Softw.				
	PPP	BadPr	F.P.	GoodPr	*PPP*	BadPr	F.P.	GoodPr	*PPP*	BadPr	F.P.	GoodPr	
Mozilla 35294	*5*	0	10	/	-	-	-	/	-	-	-	/	C++
Mozilla103330	*2*	0	0	117	*0*	0	0	7	-	-	-	-	C++
Mozilla258793	*1*	0	2	0	*0*	1	1	2	-	-	-	-	C++
Mozilla267506	*6*	0	0	9	*3*	0	0	19	-	-	-	-	C++
Mozilla311566	*26*	0	7	0	*25*	0	8	2	-	-	-	-	C++
Mozilla104962	*0*	0	0	1	*3*	0	0	12	*0*	0	0	0	C#
Mozilla124686	*0*	1	0	14	*0*	0	0	1	*0*	0	0	0	C#
Mozilla490742	*1*	0	3	5	*0*	0	0	4	-	-	-	-	JS
MySQL14637	*50*	0	11	/	*49*	0	11	/	*46*	0	31	/	C/C++
MySQL15811	*15*	20	5	5	*16*	20	7	7	-	-	-	-	C++
MySQL38769	*0*	0	1	5	-	-	-	-	-	-	-	-	C++
MySQL38941	*1*	4	0	2	*1*	4	0	2	*3*	5	2	0	C/C++
MySQL38968	*3*	0	1	38	*2*	0	2	43	-	-	-	-	C/C++
MySQL39268	*7*	0	0	4	*7*	0	0	18	-	-	-	-	C++
MySQL49491	*1*	0	0	0	*1*	0	0	2	*3*	0	0	0	C/C++
MySQL26152	*0*	0	0	0	*0*	0	0	0	*0*	0	1	4	C#
MySQL45699	*0*	2	0	0	*0*	0	0	0	*9*	0	0	45	C#/Java
Apache33605	*0*	2	0	/	*0*	2	0	/	*0*	5	0	/	C
Apache45464	*3*	0	0	47	*3*	0	0	67	-	-	-	-	C
Apache19101	*1*	0	0	1	*1*	0	0	0	-	-	-	-	Java
Apache32546	*1*	0	0	0	*1*	0	0	0	*135*	24	9	13	Java
Apache34464	*0*	0	0	3	*0*	0	0	2	*1*	0	0	12	Java
Apache44408	*1*	0	1	1	*0*	0	1	2	*3*	1	2	2	Java
Apache45396	*0*	0	0	0	*0*	0	0	1	*0*	0	0	1	Java
Apache48778	*1*	0	0	0	*1*	0	0	0	*19*	14	1	17	Java
Total	*125*	29	41	252	*113*	27	30	191	*219*	49	46	94	

Table 4: Checking results (BadPr: bad practice; F.P.: false positives; GoodPr: good practices. More detailed definitions are presented in Section 6.4. '-': not applicable. '/': good-practice checker does not exist.)

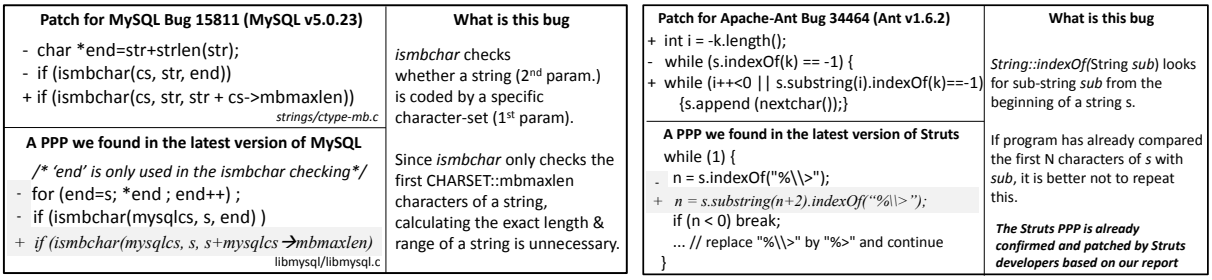

Figure 6: PPPs we found in latest versions of *original* and *different* software (the gray area shows how these two PPPs should be fixed)

using InputStream.read(byte buffer[]) with a 4KB-buffer parameter, we can stably get 3 times throughput improvement through a 40K-buffer parameter. When we feed the unit test with a 50MB file, which is a quite common image-file workload these days, the file operation time decreases from 0.87 second to 0.26 second, a definitely perceivable difference. As another example, the Struts code shown in Figure 6 is from a utility function used for processing JSP files. Our unit testing with a 15K JSP file shows that the simple patch can decrease latency by 0.1 second, a perceivable difference in interactive web applications.

Whole system testing turns out to be difficult, as suggested by our characteristics study (Section 4.3). No PPP detected by our checkers belongs to the always-active category. Future performance-oriented input-generation tools will significantly help performance testing and identify truly severe PPPs. Execution frequency information can also help future static performance-bug detectors to rank the severity of PPPs.

6.6 Discussions

Effectiveness of rule-based performance-bug detection

Effort saving Rule-based detection not only identifies problems, but also suggests alternative implementations with better efficiency. These alternative implementations often have small sizes and regular patterns, as shown in Figure 6, making PPP validation and fixing easy. It is also conceivable to enhance our checkers for automated PPP fixing.

Improving performance These PPPs showed significant performance improvement than their alternative implementations in our unit testing. Without fixing these PPPs, these unit-level performance losses could aggregate into intolerable performance problems that are difficult to diagnose. This is especially significant considering that many performance bugs are difficult to catch using other approaches.

Maintaining code readability Like those 109 patches studied earlier, most PPPs detected by us can be fixed through changes to a few lines of code, as shown in Figure 6. Even for the few complicated PPPs, wrapper-functions or macros can easily address the patch-readability issue.

Other usage Rules and checkers can serve as performance specifications for future software development. They can aid in code maintenance when software evolves. Developers can also save PPPs to an inefficiency list for future performance diagnosis.

Of course, this is only a starting point for rule-based performance-bug detection. We expect our experience to motivate future work on automatically generating rules, checkers, or even patches.

Can these problems be detected by other tools?

Copy-paste detectors Most PPPs that we found are **not** from copy-paste code regions and cannot be detected by text-matching tools [16, 28], as we can see in Figure 6. Rule violations are not rare. When developers misunderstand an API, they tend to make mistakes whenever they use this API. As a result, these mistakes usually go beyond copy-paste code regions.

Compiler optimization None of the bugs that provided the efficiency rules could be optimized away by compilers used in Apache, MySQL, and Mozilla. Many PPPs involve library functions and algorithmic inefficiency, and are almost impossible for a compiler to optimize (Figure 6). Even for the few cases where compiler optimization might help (Figure 1), the required inter-procedural and points-to analyses are not scalable for real-world large software.

General rule-based bug detectors Ideas for detecting functional bugs can greatly benefit and inspire future research on performance bug detection. However, many approaches cannot be directly applied. Tools that automatically infer functional correctness rules [13, 30, 33] may not be suitable for efficiency rules, because rule violations are not rare, as shown in Table 4. In addition, many efficiency rules either involve only one function or discourage multiple functions to be used together, making them unsuitable for tools that focus on function correlations.

7. Related Work

High performance computing (HPC) Performance is a central topic in the HPC community. Many tools for performance profiling and visualization have been developed [20, 23, 35, 38]. However, the performance issues encountered there differ from those in mainstream software. For example, the inputs of HPC are more regular; it is relatively easy to achieve high code coverage in testing HPC programs; load balancing and parallelism are much more important in HPC world.

Performance diagnosis A lot of progress has been made on profiling-based performance diagnosis [10, 19, 25, 39]. A better understanding of the common root causes, hidden locations, and propagation patterns of performance bugs can help profiling research to save manual effort in performance diagnosis and bug fixing.

Performance debugging in distributed systems has also received much attention. Many previous works tried to isolate bottlenecks in (semi-) blackbox systems [1], correlate system metrics with performance behaviors [8, 48], avoid performance problems at runtime [52], and diagnose problems through logs [59]. In distributed systems, a performance problem in a single node could spread into the entire system. Of course, there are also many performance problems unique to distributed environment.

Performance testing Performance testing includes load tests, stress tests, and soak tests. Most existing performance-testing tools [24] treat software as a blackbox. They mostly rely on developers to manually design test cases and interpret results, or simply conduct stress testing by increasing the number of concurrent users and requests. Consequently, testing is costly and ineffective [9]. It is widely acknowledged in the industry that better performance testing techniques are needed [40].

Tools for functional correctness Many ideas used for fighting functional bugs would also work for fighting performance bugs. The semantic patch project [43] provides a special language for developers to write patches in and automatically pushes each patch to multiple locations. We, however, cannot directly use semantic patches for this work, because the existing semantic patch framework does not provide sufficient data-type information and data-flow analysis support for our checkers. It also only works for C programs at this point. Of course, it is conceivable to extend the idea of semantic patches to cover performance bugs, which coincides with the goal of our work. We want to demonstrate the significance of performance bugs, and explore the challenges and opportunities faced when tackling performance bugs.

Our work only looks at one side of performance problems. It complements works on other critical issues, such as code optimization, architecture design, system resource management, mitigating false sharing [32], and configuration problems [2]. We omit further discussion due to the space constraints.

8. Conclusions

Performance bugs have largely been ignored in previous research on software defects. Facing the increasing significance of performance bugs, this paper provides one of the first studies on real-world performance bugs based on 109 bugs collected from five representative software suites. The study covers a wide spectrum of characteristics, and provides guidance for future research on performance-bug avoidance, performance testing, bug detection, etc. Guided by this study, we further explore rule-based performance-bug detection using efficiency rules implied by patches, and find many previously unknown performance problems. This work is only a starting point for understanding and fighting performance bugs. We expect it to deepen our understanding of performance bugs and bring more attention to performance bugs. More information of this work is available at `https://www.cs.wisc.edu/users/shanlu/performance-bugs`.

Acknowledgments

We would like to thank the anonymous reviewers for their invaluable feedback. We thank Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, Mark Hill, Ben Liblit, Barton Miller, and Michael Swift for many helpful discussions and suggestions. Shan Lu is supported by a Claire Boothe Luce faculty fellowship, and her research group is supported by NSF under grants CCF-1018180 and CCF-1054616.

References

[1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *SOSP*, 2003.

[2] M. Attariyan and J. Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *OSDI*, 2010.

[3] W. Baek and T. M. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *PLDI*, 2010.

[4] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[5] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: a platform for in-vivo multi-path analysis of software systems. In *ASPLOS*, 2011.

[6] A. Chou, B. Chelf, D. Engler, and M. Heinrich. Using meta-level compilation to check FLASH protocol code. In *ASPLOS*, 2000.

[7] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler. An empirical study of operating system errors. In *SOSP*, 2001.

[8] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. S. Chase. Correlating instrumentation data to system states: a building block for automated diagnosis and control. In *OSDI*, 2004.

[9] M. Corporation. *Performance Testing Guidance for Web Applications*. Microsoft Press, 2007.

[10] A. Diwan, M. Hauswirth, T. Mytkowicz, and P. F. Sweeney. Traceanalyzer: a system for processing performance traces. *Softw. Pract. Exper.*, March 2011.

[11] B. Dufour, B. G. Ryder, and G. Sevitsky. A scalable technique for characterizing the usage of temporaries in framework-intensive java applications. In *FSE*, 2008.

[12] R. F. Dugan. Performance lies my professor told me: the case for teaching software performance engineering to undergraduates. In *WOSP*, 2004.

[13] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *SOSP*, 2001.

[14] M. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *ICSE*, 2000.

[15] Fortify. HP Fortify Static Code Analyzer (SCA). https://www.fortify.com/products/hpfssc/source-code-analyzer.html.

[16] M. Gabel, J. Yang, Y. Yu, M. Goldszmidt, and Z. Su. Scalable and systematic detection of buggy inconsistencies in source code. In *OOPSLA*, 2010.

[17] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *PLDI*, 2005.

[18] S. Gulwani, K. K. Mehra, and T. Chilimbi. Speed: precise and efficient static estimation of program computational complexity. In *POPL*, 2009.

[19] M. Hauswirth, A. Diwan, P. F. Sweeney, and M. C. Mozer. Automating vertical profiling. In *OOPSLA*, 2005.

[20] J. K. Hollingsworth, R. B. Irvin, and B. P. Miller. The integration of application and system based metrics in a parallel program performance tool. In *PPOPP*, 1991.

[21] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *OOPSLA*, 2004.

[22] InfoWorld. Top 10 open source hall of famers. http://www.infoworld.com/d/open-source/top-10-open-source-hall-famers-848.

[23] R. B. Irvin and B. P. Miller. Mapping performance data for high-level and data views of parallel program performance. In *ICS*, 1996.

[24] JMeter. Java desktop application designed for load tests. http://jakarta.apache.org/jmeter.

[25] M. Jovic, A. Adamoli, and M. Hauswirth. Catch me if you can: performance bug detection in the wild. In *OOPSLA*, 2011.

[26] C. Killian, K. Nagaraj, S. Pervez, R. Braud, J. W. Anderson, and R. Jhala. Finding latent performance bugs in systems implementations. In *FSE*, 2010.

[27] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, 2004.

[28] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. In *OSDI*, 2004.

[29] Z. Li, L. Tan, X. Wang, Y. Zhou, and C. Zhai. Have things changed now?: an empirical study of bug characteristics in modern open source software. In *ASID*, 2006.

[30] Z. Li and Y. Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *FSE*, Sept 2005.

[31] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn. Flikker: saving DRAM refresh-power through critical data partitioning. In *ASPLOS*, 2011.

[32] T. Liu and E. D. Berger. Precise detection and automatic mitigation of false sharing. In *OOPSLA*, 2011.

[33] V. B. Livshits and T. Zimmermann. Dynamine: Finding common error patterns by mining software revision histories. In *FSE*, 2005.

[34] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes – a comprehensive study of real world concurrency bug characteristics. In *ASPLOS*, 2008.

[35] J. Mellor-Crummey, R. J. Fowler, G. Marin, and N. Tallent. Hpcview: A tool for top-down analysis of node performance. *J. Supercomput.*, 23(1):81–104, 2002.

[36] N. Meng, M. Kim, and K. S. McKinley. Systematic editing: generating program transformations from an example. In *PLDI*, 2011.

[37] Microsoft. MSDN SAL annotations. http://msdn2.microsoft.com/en-us/library/ms235402.aspx.

[38] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The paradyn parallel performance measurement tool. *Computer*, 28(11), 1995.

[39] N. Mitchell. The diary of a datum: an approach to modeling runtime complexity in framework-based applications. In *ECOOP*, 2006.

[40] I. Molyneaux. *The Art of Application Performance Testing: Help for Programmers and Quality Assurance*. O'Reilly Media, 2009.

[41] G. E. Morris. Lessons from the colorado benefits management system disaster. www.ad-mkt-review.com/publichtml/air/ai200411.html, 2004.

[42] G. Muller, Y. Padioleau, J. L. Lawall, and R. R. Hansen. Semantic patches considered helpful. *Operating Systems Review*, 40(3):90–92, 2006.

[43] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller. Faults in linux: ten years later. In *ASPLOS*, 2011.

[44] S. E. Perl and W. E. Weihl. Performance assertion checking. In *SOSP*, 1993.

[45] T. Richardson. 1901 census site still down after six months. http://www.theregister.co.uk/2002/07/03/1901_census_site_still_down/.

[46] C. J. Rossbach, O. S. Hofmann, and E. Witchel. Is transactional programming actually easier? In *PPoPP*, 2010.

[47] A. Shankar, M. Arnold, and R. Bodik. Jolt: lightweight dynamic analysis and removal of object churn. In *OOPSLA*, 2008.

[48] K. Shen, C. Stewart, C. Li, and X. Li. Reference-driven performance anomaly identification. In *SIGMETRICS*, 2009.

[49] R. L. Sites. Identifying dark latency.

[50] C. U. Smith and L. G. Williams. Software performance antipatterns. In *Proceedings of the 2nd international workshop on Software and performance*, 2000.

[51] Stefan Bodewig. Bug 45396: There is no hint in the javadocs. https://issues.apache.org/ bugzilla/show_bug.cgi?id=45396#c4.

[52] C. Stewart, K. Shen, A. Iyengar, and J. Yin. Entomomodel: Understanding and avoiding performance anomaly manifestations. In *MASCOTS*, 2010.

[53] M. Sullivan and R. Chillarege. A comparison of software defects in database management systems and operating systems. In *FTCS*, 1992.

[54] L. Torvalds. Sparse - a semantic parser for c. http://www.kernel.org/pub/software/devel/sparse/.

[55] J. S. Vetter and P. H. Worley. Asserting performance expectations. In *Supercomputing*, 2002.

[56] wikipedia. Chi-squared test. http://en.wikipedia.org/wiki/Chi-squared_test.

[57] G. Xu, M. Arnold, N. Mitchell, A. Rountev, and G. Sevitsky. Go with the flow: profiling copies to find runtime bloat. In *PLDI '09*, 2009.

[58] G. Xu, N. Mitchell, M. Arnold, A. Rountev, E. Schonberg, and G. Sevitsky. Finding low-utility data structures. In *PLDI*, 2010.

[59] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *SOSP*, 2009.

[60] A. Zeller. Isolating cause-effect chains from computer programs. In *FSE*, 2002.