An Empirical Study on Performance Bugs for Highly Configurable Software Systems

Xue Han, Tingting Yu
Department of Computer Science
University of Kentucky
Lexington, KY, 40506, USA
xha225@g.uky.edu, tyu@cs.uky.edu

ABSTRACT

Modern computer systems are highly-configurable, complicating the testing and debugging process. The sheer size of the configuration space makes the quality of software even harder to achieve. Performance is one of the key aspects of non-functional qualities, where performance bugs can cause significant performance degradation and lead to poor user experience. However, performance bugs are difficult to expose, primarily because detecting them requires specific inputs, as well as a specific execution environment (e.g., configurations). While researchers have developed techniques to analyze, quantify, detect, and fix performance bugs, we conjecture that many of these techniques may not be effective in highly-configurable systems. In this paper, we study the challenges that configurability creates for handling performance bugs. We study 113 real-world performance bugs, randomly sampled from three highly-configurable open-source projects: Apache, MySQL and Firefox. The findings of this study provide a set of lessons learned and guidance to aid practitioners and researchers to better handle performance bugs in highly-configurable software systems.

CCS Concepts

•Software and its engineering → Software notations and tools;

Keywords

Empirical Study, Performance, Configuration

1. INTRODUCTION

Modern software systems are highly-configurable, allowing users to customize a large number of configuration options while retaining a core set of functionality. The environment to which the application deploys has become increasingly more complex. The application can frequently interact with other system components such as shared libraries, environment variables, and kernel modules. The complexity of the configuration space and the sophisticated constraints among configuration settings could easily cause performance bugs. Unlike a functional bug that typically leads to system crashes or incorrect results, a performance bug can cause significant performance degradation, leading to problems such as poor user experience, long response time, and low system throughput [6, 16, 20].

Compared to functional bugs, performance bugs are substantially more difficult to handle because they often manifest themselves only with large inputs and specific execution environments [20, 22]. As such, traditional testing such as coverage-based approaches may not be effective. To address these problems, numerous research efforts have been made to analyze, detect, and fix performance bugs [7,13,16,17,21,22]. For example, rule-based pattern matching has been used to detect performance anomalies under normal executions [16]. Several test case generation techniques have been proposed to generate large workload test inputs [7, 23].

However, most existing work assumes default configurations while ignoring the influence of various configurations. The tasks of performance testing and debugging can still be challenging without considering the complex combinations of configuration options and environment factors. A typical configurable system may have thousands of options, and this generates numerous possible configurations. For example, Apache has more than 1000 possible configuration options [8]. Therefore, we believe that answering research questions related to how configurations can influence system performance can guide the design of, and improve techniques for addressing performance bugs. In addition, understanding configurations can facilitate performance optimization, such as finding a best set of configurations for system performance [29].

A natural question to ask is to what extent do performance bugs have the potential to go undetected if tested under default environments. If only a few performance bugs involve configurations, then developers may use existing performance testing and modeling techniques and do not need to be overly concerned about dealing with configurations. On the other hand, if performance bugs are indeed sensitive to configurations, it is also worth exploring the characteristics of such configurations. For example, one may ask if certain configuration options (e.g., cache settings) are more likely to trigger performance bugs than others. If configuration-related performance bugs differ from those general performance bugs, developers may not use existing configuration-aware techniques.

In this paper, we perform a characteristic study on performance bugs related to configurations. We consider both

ESEM '16, September 08-09, 2016, Ciudad Real, Spain
 ISBN 978-1-4503-4427-2/16/09…$15.00
DOI: http://dx.doi.org/10.1145/2961111.2962602

performance bugs caused by misconfigurations and those that can manifest under specific configurations. We aim to uncover and quantify the extent to which the performance problem exists on real-world highly-configurable software systems. Specifically, we manually inspect 193 performance bugs from three popular and highly-configurable open-source applications: Apache, MySQL and Firefox. We then perform a deep analysis on 113 configurations-related performance bugs. We see this study as a way to share with researchers and practitioners the performance issues that the configurability brings, a set of lessons learned, and a roadmap for developing configuration-aware techniques to address performance issues.

In this paper we have made the following contributions:

- We examine the prevalence of configuration issues that have led to performance bugs. We find that more than half of the performance bugs (59%) are due to configuration problems.
- We classify configurations into three types: parameter, hidden, and system-specific configurations. We find that a majority of configuration-related performance bugs (78% – 92%) are related to configuration parameter settings. However, a non-trivial portion of studied bugs (8% – 17%) are due to system-level configurations, motivating the need to consider system properties when addressing performance problems.
- We study different causes of configuration-related performance bugs. We find that performance bugs can be detected by observing internal symptoms, such as high CPU usage, low cache hit, and long synchronization time. These symptoms are also related to different domains of configuration options (e.g., memory, concurrency, and network). We also find that configuration-related performance bugs are caused by a small subset of the entire configuration space, motivating the use of configuration space reduction techniques to handle performance bugs.
- We examine the fixes of configuration-related performance bugs. We find that a majority of studied bugs (88%) require fixing source code instead of just changing values in configuration options. In addition, the patches for the examined bugs involve over 30 lines of code in average, which are more complex than the patches for general performance bugs (less than 10 lines) [16].
- Finally, we provide a set of lessons learned that will help practitioners and researchers better understand and address performance bugs in highly-configurable software systems.

The rest of the paper is organized as follows. We first present motivating examples in Section 2. We then describe our methodology for choosing subject applications, the bugs selected to study, and the threats to validity in Section 3. Our results are demonstrated in Section 4, followed by discussions in Section 5. We present related work in Section 6, and end with conclusions in Section 7.

2. MOTIVATION

We define a configurable system as a software system with a core set of functionality and a set of variable features, which are defined by a set of configuration options – including user provided options (or parameters) and environment settings (e.g., libraries, components, etc). Changes to the value of configuration options affect programs' behavior in some way. We use the term "configuration options" and "configuration parameters" interchangeably.

We use six motivating examples from the bugs we have studied to answer the following questions: 1) Do performance bugs require specific configurations to manifest? 2) Are configuration-related performance bugs different from configuration-related general bugs for testing, debugging, and fixing? 3) Do developers need special tools to handle performance bugs?

Wasted loop computation. Some users occasionally experience a slowdown during the Apache graceful restart as stated in Apache bug 54852. To trigger this bug, user needs to start Apache with a relatively larger number for the `StartServers` option (e.g, `StartServers` = 60). In this case, the server takes more time to restart than usual. Code inspection pinpoints the root cause: a wasted loop computation. When the children servers already exited, there is no need to iterate these servers. Figure 1 shows the fix for this bug. By checking the existence of the children server processes first, the time-consuming `dummy_connection` function is skipped if the server no longer exists. This example helps answer the first question as exposing the bug requires a specific configuration option (i.e., `StartServers`).

Page not cleaned. The page cleaner feature (i.e., the `innodb_page_cleaners` option) can improve MySQL performance scalability by spawning multiple threads to flush dirty pages from buffer pool instances. In MySQL bug 72703, the database server experienced a slowdown during the shutdown phase. The root cause of this bug is due to the wrong implementation of the page cleaner feature. Figure 2 shows the fix associated with this option. This example indicates that incorrect implementation of configurations can negatively impact application performance.

Lock contention. In MySQL bug 77094, when the configuration option `innodb_flush_log_at_trx_commitis` is set to 2, the two logging functions (commit and write) both use the `log_sys->mutex` lock to write to a buffer. This inevitably causes lock contention that hurts the application performance. The fix is to use two different buffers, as shown in Figure 3, so that commit and write functions can be concurrently executed.

Cache not purged. In Apache bug 46749, a developer reports that when the proportion of `LDAPSharedCacheSize` to `LDAPCacheEntries` is too small (e.g., {`LDAPSharedCacheSize` 200K, `LDAPCacheEntries` 1024 }), the old cache entries will not get purged. Hence, the cache hit rate drops rapidly and degrades the system performance. This performance bug requires both `LDAPSharedCacheSize` and `LDAPCacheEntries` set to specific values to manifest. The fix is to change the default setting of `LDAPSharedCacheSize` (i.e., 200K) to 500k. In addition, as shown in Figure 4, the source code is patched by inserting log statements (i.e., `ap_log_error`) as extra checks to help developers debug the code. For example, if the system fails to allocate memory for a new entry after the purge, events are recorded and the code returns NULL to the caller.

```
ap_mpm_pod_killpg(ap_pod_t *pod, int num){
   for (i=0;i<num && rv==APR_SUCCESS;i++) {
+      if(ap_scoreboard_image->
+      servers[i][0].status!=SERVER_READY
+      || ap_scoreboard_image->servers[i][0].pid
+      == 0)
+      continue;
+   rv=dummy_connection(pod);
   }
}
```

Figure 1: Apache bug 54852

```
/*storage/innobase/srv/srv0start.cc*/
+for (i = 1; i < srv_n_page_cleaners; ++i){
+   os_thread_create(buf_flush_page_
+   cleaner_worker, NULL, NULL);
}
```

Figure 2: MySQL bug 72703

```
/*Add one more buffer*/
byte* buf_pair_ptr[2];
        ...
/** Acquire the log sys mutex. */
#define log_mutex_enter_all() do {
+   mutex_enter(&log_sys->w_mutex);
    mutex_enter(&log_sys->mutex);
} while (0)
        ...
/** Release the log sys mutex. */
#define log_mutex_exit_all() do {
+   mutex_exit(&log_sys->w_mutex);
    mutex_exit(&log_sys->mutex);
} while (0)
```

Figure 3: MySQL bug 77094

Abnormal termination. In Apache bug 42829, configuring multiple listening ports (e.g., {`Listen 80`, `Listen 8080`}) causes Apache server to hang during a graceful restart. This bug happens when a child process receives a software signal that closes the listening sockets before the process starts polling the sockets using the `apr_pollset_poll` function. The `apr_pollset_poll` function is only called when multiple listening ports are present. Since no listening sockets are available (closed by the signal), the calling child process waits indefinitely. Figure 5 shows the fix for this bug. In the code patch, the timeout is updated to 10 seconds to avoid waiting `apr_pollset_poll` function without bound. This bug involves 64 posts over the course of 4 years to close since opened in 2007.

Slow autocomplete feature. In Firefox, the autocomplete in the URL bar is expected to expand URLs immediately as users type. In Firefox bug 415489, a user reports that the autocomplete becomes extremely slow. The bug is due to misconfigurations of `browser.urlbar.search.chunkSize` and `browser.urlbar.search.timeout` preferences, where the first option defines the number of chunks must be collected from SQLite database, and the second option defines the search timeout value. When the chunk size is set too small, the data query completes fast and waits for the search timeout to return and thus leading to performance degradation. The fix is to balance the chunk size and the timeout value (Figure 6).

The above six bugs help motivate this study and answer some earlier questions about configuration-related performance bugs in the following ways: 1) Performance bugs are triggered by not only inputs, but also specific configurations. 2) These bugs have shown similarities to general configuration bugs. For example, changing values of configuration options can affect the system performance. And performance issues may also require more than one configuration option to manifest. On the other hand, these bugs are also different from general configuration bugs: the configuration options relate much more to memory (Figure 4), synchronization (Figures 2 and 3), and network (Figures 1 and 5) operations. 3) Existing configuration testing or performance testing may not be effective to handle these bugs. For instance, performance testing tools unaware of configuration settings may miss all of the six bugs; whereas when applying traditional configuration-aware testing techniques, sampling large configuration space may still ending up with options not relevant to performance issues.

```
node = (util_cache_node_t *)util_ald_alloc(cache,
sizeof(util_cache_node_t));
if (node == NULL) {
-   return NULL;
+   ap_log_error("LDAPSharedCacheSize is too small...");
+   if (cache->numentries < cache->fullmark){
+   cache->marktime = apr_time_now();
+   }
+   util_ald_cache_purge(cache);
+   if (node == NULL){
+      ap_log_error("Could not allocate memory ...");
+      return NULL;
+     }
}
```

Figure 4: Apache bug 46749

```
for (;;) {
+  status = apr_pollset_poll(pollset,
-           -1, ...); /*wait forever*/
+           apr_time_from_sec(10), ...);
+  if (APR_STATUS_IS_TIMEUP(status)
+     continue;
}
```

Figure 5: Apache bug 42829

```
- pref("browser.urlbar.search.chunkSize", 100);
- pref("browser.urlbar.search.timeout", 100);
+ pref("browser.urlbar.search.chunkSize", 1000);
+ pref("browser.urlbar.search.timeout", 50);
```

Figure 6: Firefox bug 415489

3. CASE STUDY

Our study has two main objectives. First, we intend to understand the complexity of performance bugs in highly-configurable systems. Second, we want to understand what are the challenges that we will face as we apply configuration-aware techniques to performance issues. Therefore, we consider the following research questions.

RQ1: How prevalent are performance bugs related to configurations?

RQ2: What are the types of configurations that can influence performance?

RQ3: What are the causes of configuration-related performance bugs?

RQ4: How complicated is it to fix configuration-related performance bugs?

3.1 Data Sets

3.1.1 Studied Subjects

We chose three large, mature and popular open-source software projects: Apache, MySQL and Firefox. With millions of lines of publicly accessible code and well maintained

Table 1: Subjects And Their Characteristics

Application	*Sampled Bugs*	*Used Bugs*	*Config Bugs*	*# of Opts.*	*Studied Opts.*	*Configuration Options Appeared More Than Once*
Apache Suite HTTPD server	323	63	60	1,145	35	`KeepAlive` (5), `MaxClients` (3), `Listen` (3), `ThreadsPerChild` (2), `RequestHeader` (2), `ProxyPass` (2), `BalancerMember` (2), `SSLVerifyClient` (2), `AuthLDAPURL` (2)
MySQL Suite Databser Server	241	77	41	1,429	31	`query_cache_size` (4), `innodb_flush_log_at_trx_commit` (3) `innodb_flush_method` (2), `innodb_thread_concurrency` (2) `innodb_buffer_pool_size` (2), `read_buffer_size` (2) `sort_buffer_size` (2), `gtid_mode` (2) `query_cache_type` (2), `profiling` (2), `sync_binlog` (2)
Mozilla Suite Firefox	323	53	12	1,650	24	`Places` (5), `browser.urlbar.search.chunkSize` (2) `browser.urlbar.search.timeout` (2) `browser.urlbar.maxRichResults` (2) `Collusion` (2), `flash plugin` (2)
Total	887	193	113	3,700	90	

bug repositories, these subjects have been widely used by existing bug characteristic studies [16, 37, 39]. The selected programs are listed in Column 1 of Table 1. The subject programs cover various application spectrums - the world's most used HTTP server, the world's most popular database engine, and a leading web browser. All three projects started in the early 2000's and each has over ten years of bug reports.

3.1.2 Data Collection

Bugs. We picked configuration-related performance bugs from two sources: bug repositories and changelogs. We searched bug databases using a set of performance-related keywords ("slow", "performance", "latency", "throughput", etc.). We filtered out unconfirmed reports, which yielded a total of 887 bugs (Column 2 of Table 1). During the manual inspection, we follow those reports that have sufficient details in bug descriptions and discussions posted by commentators, and decide if the inspected bug is a performance bug or not, and whether the bug is related to configuration or not. These bug reports often describe the configuration options for reproducing the performance bugs. We collected both performance bugs due to misconfigurations and bugs that are triggered by specific configurations.

To ensure the correctness of our results, the manual inspections were performed independently by three graduate students. For bugs where the results from three inspections differ, the authors and the inspectors discussed to reach a consensus. As such, the examination yielded a total of 193 performance bugs, and 113 of them are related to configurations (Column 3-4 in Table 1).

Configurations. To answer our research questions, we also need to know the configuration space for each subject. We collected such information by studying all artifacts that are publicly available to users, including documents (e.g., user manuals and online help pages), configuration files, and source code. In Firefox, we also utilized the APIs that have been provided to programmatically manipulate internal data structures that hold configuration information, as well as studied the `about:config` page (a utility for modifying configurations). This process yielded the total number of configuration options for the three subjects (Column 5 of Table 1). Column 6 of Table 1 lists the total numbers of configuration options for the 113 studied bugs. The last column lists the configuration options and the number of bugs that each option is associated with (indicated in the parenthesis). We list only options that appeared in more than one studied bugs. These options will be discussed in the next section.

3.2 Threats to Validity

The primary threat to external validity for this study involves the representativeness of our subjects and bugs. Other subjects may exhibit different behaviors. Data recorded in bug tracking systems and code version histories can have a systematic bias relative to the full population of bug fixes [5] and can be incomplete or incorrect [2]. However, we do reduce this threat to some extent by using several varieties of well studied open source code subjects and bug sources for our study. A second source of potential threat involves the age of bug reports. Since the sampled bug reports span over ten years, the number of configurations and their options may change as time passes by. This may somewhat affect the methodological consistency. We have examined these bug reports and found that such changes are minor.

The primary threat to internal validity involves the use of keyword search and manual inspection to identify the configuration-related performance bugs. To minimize the risk of incorrect results given by manual inspection, bug were labeled as performance bugs independently by three people. The recall of our approach is estimated to be 50%. To compute this recall, we randomly sampled 227 bugs and manually inspected each of them. We found 6 performance bugs, of which only 3 were found by the keyword search and manual inspection. Such an approach is also used by Nistor et al. [20]. The risk of not analyzing all performance bugs cannot be fully eliminated. However, combining keyword search and manual inspection is an effective technique to identify bugs of a specific type from a large pool of generic bugs, which was successfully used in prior studies [16,20,37].

The primary threat to construct validity involves the dataset and metrics used in the study. To mitigate this threat, we used bug reports from the bug tracking systems of the three subjects, which are publicly available and generally well understood. We have also used well known metrics in our data analysis such as the number of bugs and the number of lines in the patch, which are straightforward to compute.

4. RESULTS

We now present our results for each of the four research questions [1].

4.1 RQ1: Prevalence of Bugs

To answer RQ1, we turn to Figure 7. A total of 193 performance bugs are classified into configuration and non-configuration bugs (solid grey area), where configuration bugs are further classified into three categories (described in

[1]Artifacts and experimental data are available at http://cs.uky.edu/~tyu/research/perfconf

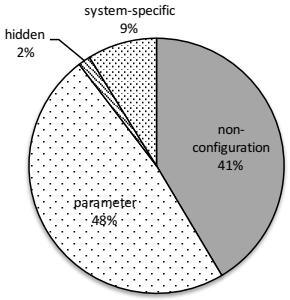

Figure 7: Bug classification

RQ2). For the 80 non-configuration performance bugs, defects are normally caused by semantic bugs in code that are independent of configurations. For instance, in MySQL bug 15935, an update query takes a very long time to complete. The degradation of performance is caused by the wasted search of the entire table indices, though only a smaller range of search is necessary. While the defect numbers can be potentially influenced by our sampling strategy, among all four categories, configuration-related performance issues contribute to 59% of the studied cases.

Finding 1: *A significant percentage of performance bugs are related to configuration issues (59%). Compared to general configuration bugs (27%) from a previous study [37], if these results generalize, performance bugs are more relevant to configurations.*

4.2 RQ2: Configuration Types

We begin by answering RQ2 by classifying the configurations of the examined 113 bugs into three general categories: parameter configurations, hidden configurations, and system-specific configurations. The parameter configurations refer to the configurations whose values can be changed by end users through configuration files. The six motivating examples in Section 2 fall into this category. The hidden configurations refer to program variables embedded in the source code which can be set only if developers are aware of them. The system-specific configurations refer to configurations related to hardware, system topology, and the choice of system core libraries. Table 2 indicates the categories and the total number of bugs falling into each category.

Finding 2: *The parameter configurations account for a majority of the examined configurations (78% to 92%). This ratio is similar to the finding by Yi et al. [37], where up to 85.5% of general bugs are related parameter configurations*

Finding 3: *However, the system-specific configurations account for a non-trivial portion (8% to 17%), and should be of particular concern for performance.*

We next describe the three configuration categories in more detail.

4.2.1 Parameter Configurations

Given the prevalence of parameter configuration bugs, we have studied them from three perspectives: types of configurations, the number of parameters, and the problem domains.

Types of parameters. We look at *illegal* and *legal* configurations. The illegal configurations refer to violations of configuration rules related to format, syntax, or semantics.

Table 2: Types of Configurations

Project	Parameter	Hidden	Sys-spec	Total
Apache	50	1	9	60
MySQL	32	2	7	41
Firefox	11	0	1	12

Table 3: Parameter Configurations

Project	*one option*	two options	*> 2 options*	Total
Apache	36	11	3	50
MySQL	23	7	2	32
Firefox	8	2	1	11

Such configurations are unacceptable to the examined system. For example, a directory name option `datadir` can point to the wrong directory causing MySQL not able to start (MySQL bug 65022). However, we have not found any performance bugs related to illegal configurations. Our further investigation exhibits two reasons. First, illegal configurations often lead to functional bugs, such as startup failures and error messages. Second, long term applications usually have established built-in syntax checkers as the first line of defense to guarantee the correctness of configuration files. For instance, after issuing a graceful restart in Apache server, it validates the changed configuration file before booting up.

On the other hand, legal configuration options can still cause performance issues. For instance, some performance bugs, although not directly caused by misconfigurations, require certain configuration options to manifest, such as the example in Figure 1. Others can be fixed by updating the configuration values as seen in Figure 3 and Figure 4. Performance bugs like these are difficult to detect as it requires users' experience to select the right configurations to expose them. To address this problem, it is possible to leverage some code-level analysis techniques [4, 40].

Finding 4: *All studied parameter configuration bugs results from legal options. The ratio of the number of bugs caused by legal configuration values over all the studied configuration performance bugs is much higher than the finding (46.3% - 61.9%) for general configuration bugs by Yi et al. [37].*

Numbers of incorrect parameters. Existing configuration-aware techniques mostly focus on single and two configuration options (e.g., 2-way combinatorial testing [8]). There have also been many debugging techniques that focus on only single configuration options [40]. In this study, we examine the percentage of performance bugs involved in different numbers of configuration options – one option, two options and more than two options.

Columns 1-3 in Table 3 show the number of parameters involved in configurations that lead to performance bugs. Our analysis indicates that about 27% to 28% of the parameter configuration bugs involve multiple options. For instance, the cache purge example in Figure 4 shows a bug case where two options are involved.

Finding 5: *The majority (72% to 73%) of studied parameter configuration bugs is related to only one option, whereas about 27% to 28% of the examined parameter configuration bugs involve two and more options.*

Problem domains of configurations. We also study under which problem domain each performance bug falls. We propose five domains based on the functionality of the in-

Table 4: Types of Configurations

Project	*Memory*	*Network*	*I/O*	*Concurrency*	*Graphics*
Apache	2	27	1	1	0
MySQL	12	12	8	4	0
Firefox	1	0	0	0	3
Total	15	39	9	5	3

volved parameters: memory, network, I/O, concurrency, and graphics. Table 4 indicates the domains and the total number of bugs falling into each category. Other studied bugs do not fall into these categories hence purposefully left out. For instance, the example in Figure 4 (i.e., `LDAPSharedCacheSize`) falls into the memory domain. The example in Figure 5 (i.e., `Listen`) falls into the network domain. In fact, 87% of parameter configurations in Apache fall into the Network domain, and 67% of parameter configurations in MySQL are equally distributed to Memory and Network domains.

The concurrency domain involves options that manipulate threads and synchronizations (e.g., the `skip-thread-priority` in MySQL bug 37536). The I/O domain involves options that affect I/O operations. For example, in MySQL Bug 61818, performance can be adjusted by setting the option `innodb_flush_log_at_trx_commit` which controls the timing of flushing log files to disks. The graphics domain involves the control of the display. For instance, in Firefox bug 820247, a noticeable lagging occurs when rendering the Bookmarks submenu. Graphics rendering is much faster after turning off the Hardware Acceleration (GPU) in the preferences ("Use hardware acceleration when available").

Finding 6: *The majority of the examined parameters fall into the Memory and Network domains. However, the distribution depends on the characteristics of applications (e.g., Firefox is more relevant to the graphics domain).*

4.2.2 Hidden Configurations

There are hidden configurations found only in the source code and related files. These hidden configurations are difficult to handle by existing techniques such as reverse engineering [26], since many applications are written in multiple programming languages such as C++, Java, and JavaScript, and often use aliases to refer to configuration option names. Neither of which are supported by existing techniques. Jin et al. [15] has reported that up to 44% options are hidden in the code and user manuals.

However, our observation is that only a small portion (3 cases) of hidden configurations are related to performance bugs. All three configurations are found in the source code. The first case, MySQL bug 20876, shows the default value of `FILE_SYSTEM_HASH_SIZE` is set to a small value, causing the CPU to spike to 100%. The second case found in MySQL bug 64258 concerns the default read timeout value (`WAIT_FOR_READ`) for storage devices, which is set to be unnecessarily long. The last one is in Apache bug 58091, where `MC_DEFAULT_SERVER_TTL` is set too short for the connection timeout.

Finding 7: *A small portion (3%) of the examined parameter configuration bugs involve hidden options.*

4.2.3 System-level Configurations

The system-specific configurations refer to the misconfigurations of the external environment under which an application executes, such as system topology (environment setup), hardware choice, and incompatible components/libraries. Table 5 shows the three types and the total number of bugs falling into each type.

Table 5: System-specific Configurations

Project	*Environment*	*HW*	*Components*
Apache	1	0	8
MySQL	1	2	4
Firefox	0	1	0
Total	2	3	12

While a complex environment setup serves its purposes, it could also be a source of performance issues, as we found in the Apache bug 45834. Apache is used to host SVN on a RedHat 5 server with a firewall that sits in between the hosting server and the LDAP server. The authentication process (`mod_authnz_ldap`) takes a long time to complete without doing any useful work. The root cause of this bug is that the firewall breaks the established connection between the SVN server and the LDAP server, and hence causes the retransmitting of TCP packets. Changing the timeout value in the LDAP component fixes the bug.

Another cause of performance bugs involve choosing inappropriate hardware. This can happen when customers choose between different hardware settings (e.g., a local file system v.s. a network storage system). For instance, in MySQL bug 21947, the storage area network (SAN) is used as the storage device that hosts the file system. When the option `innodb_flush_method` is set to `O_DIRECT`, it could slow down the SQL select statement by a factor of three. As another example shows, some Intel CPU models equipped with Hyper-Threading (HT) technology [14], are supposed to boost throughput on threaded applications. Nonetheless, from MySQL bug 15815 we have learned that concurrency performance is much worse when HT is enabled due to lock contentions.

Software incompatibility is another major cause of performance bugs. The application may work well under a standard set of components, yet the performance problems can occur when the components are improperly customized. For example, in the Apache bug 40010, the server becomes unresponsive on a FreeBSD OS with Jails installed. The reason is that with Jails installed, the IP address used by the Apache server is not mapped to the appropriate IPv6 address on FreeBSD, causing the connection to be rejected. The rejected connection freezes the server.

Finding 8: *There are various system-level misconfigurations that can cause performance bugs. The majority (70%) of the examined system-specific configurations involve incompatible components/libraries.*

4.3 RQ3: Causes of Bugs

4.3.1 When are Bugs Introduced?

We examine the stages where configuration-related performance bugs are introduced. We categorize these cases into (1) first-time use, (2) runtime reconfiguration, and (3) application upgrade. Table 6 shows the three stages and the total number of bugs falling into each category.

We define the first-time use as configuration files initially read from persistent storage (e.g., hard disk) into memory. All motivating examples except the one in Figure 6 in Section 2 fall into this category. The causes for the configuration bugs during first-time use can be inadequate domain knowledge, design defects of the system, user mistakes, or even inconsistent manuals [27].

Table 6: Stages of Bugs

Project	*First-time*	*Runtime*	*Upgrade*
Apache	60	0	4
MySQL	37	0	4
Firefox	11	12	2
Total	108	12	10

Table 7: Causes of Bugs

Project	*Memory*	*CPU*	*Synchronization*
Apache	24	11	1
MySQL	13	3	15
Firefox	2	5	0
Config Domain	Memory	Memory, I/O, Graphics, Network	Concurrency

In the case of runtime reconfiguration, when applications are running, a user can modify the configuration files directly. Only in Firefox will a modification take effect immediately and be written back to the preference files. The configuration options `browser.urlbar.search.chunkSize`, and `browser.urlbar.search.timeout` in the example of Figure 6 can be reconfigured at runtime. In contrast, in Apache and MySQL, the dynamic memory is not updated. The changed configurations are held in temporary memory and will take effect at the next startup.

Performance bugs can also be introduced when upgrading an application even if there are no changes in the configuration files. For example, in Apache bug 58037, after Apache server updates from version 2.2 to 2.4, the time to complete authentication grows as the number of authentication connection increases, though all configuration values remain unchanged. This bug is due to the `LDAPConnectionPoolTTL` option, whose default value is zero in the old version. However, in the new version, the server only unlocks connection when this option is not set to zero. As a result, the size of the connection entries grows rapidly. By the time a new connection is issued, all the locked connections will be checked, the time spent on checking connections that should have been unlocked explains the increased processing time. This bug can be avoided by setting the `LDAPConnectionPoolTTL` option to a non-zero value.

Finding 9: *The majority (95%) of studied performance bugs are related to the first-time use. However, in applications that allow runtime reconfiguration, the performance bugs caused by runtime misconfiguration are non-negligible (11% in Firefox).*

4.3.2 Why do Performance Bugs Happen?

We further examine the causes of performance bugs based on *observable* internal symptoms. In contrast to external symptoms, where the application simply hangs or slows down, the internal symptoms reflect the possible presence of performance bugs. We consider three major internal symptoms: 1) memory usage, 2) CPU usage, and 3) synchronization. These symptoms can be observed by profiling and system monitoring tools. Table 7 indicates the three categories, the number of bugs, and the configuration domains falling into each category.

The memory problem can refer to memory leak where the program fails to release memory when no longer needed, or low cache hits. The inefficient memory usage can negatively impact the performance of the systems [35]. For instance, in Apache bug 44975, when both `mod_ssl` and `mod_deflate` are enabled, a per connection memory leak is triggered by the client who initiates an SSL handshake with compression algorithm support. The example in Figure 4 shows the symptom of low cache hits. The high CPU usage means that the running programs hold a lot of CPU resources, which is an indicator of performance anomalies [34]. In the MySQL bug 55629, with a large read buffer (`read_buffer_size`), the wrong error code returned during I/O cache flushing from a select query causes the database server to loop indefinitely and results a high CPU usage.

Table 8: Bug Fix

Project	*Options*	*Source*	*Option&Source*	Total
Apache	5	54	1	60
MySQL	2	39	0	41
Firefox	5	6	1	12

The inefficient synchronization usage refers to the scenario when multiple threads contend to reach a synchronization point. Such synchronization bottlenecks can lead to performance bugs. For example, in MySQL bug 37536, the `skip-thread-priority` option is used to change thread priorities. However, this option can cause significant performance degradation depending on the thread count and number of processors.

Finding 10: *A majority of performance bugs involves inefficient memory usage (up to 35%), and a significant portion of performance bugs are caused by high CPU utilization (up to 17%) and synchronization (up to 14%).*

4.3.3 What configuration options are culprits?

As the last column in Table 1 shows, in all studied bugs, some configuration options appear more than once (nine in Apache, 11 in MySQL, and six in Firefox). We conjecture that these options are more likely to relate to performance bugs. The anecdotal evidence from the recent release of MySQL [31] also indicates that only 17 configuration options are the most common causes for MySQL performance degradation. The 17 options are also covered by the 31 configuration options studied in MySQL.

Finding 11: *Performance bugs are caused by a small subset of configuration options.*

4.4 RQ4: Bug Fixes

Finally, we examine the fix of performance bugs in highly-configurable systems. We focus on how to fix the bugs and the complexity of the fixes.

4.4.1 How to Fix Performance Bugs?

Fixing performance bugs can be done by (1) changing configuration values, (2) patching source code, and (3) fixing both configuration values and source code. Table 8 shows the three categories and the number of bugs that fall into each category.

We find that only a few cases require fixing solely configuration options. In contrast, 88% of performance bugs need to be fixed at the code level, such as the examples in Figures 1, 3, and 5. About 2% of performance bugs require fixing both configurations and the code. The example in Figure 4 is such a case.

Finding 12: *A dominate majority (88%) of performance bugs involve fixing the code.*

4.4.2 Are Patches Complex?

Jin et al. [16] study general performance bugs and their

results indicate that performance bugs can be fixed by simple patches (e.g., 8 lines on average). We find that fixing performance bugs in highly-configurable systems is not that simple. In fact, up to 61% of studied bugs require fixing over 30 lines of code.

Finding 13: *Fixing configuration-related performance bugs is more complex than fixing general performance bugs.*

5. DISCUSSION

Our study motivates further research on performance testing – performance bugs cannot be exposed easily without specific configurations. In this section, we summarize the implications learned from our study. The first part is geared towards practitioners, since they reflect the state-of-the-art practices. The second part provides a roadmap for researchers who plan to develop new tools and techniques for addressing performance issues in highly-configurable applications.

5.1 Implications to Practitioners

Performance testing should consider key configurations. Configuration-aware testing has been widely adopted in industrial environment [24] (e.g., combinatorial interaction testing [1]). Configuration-aware testing can be expensive because the space of possible unique configuration combinations grows exponentially with the configuration options. To address this problem, testers often evaluate a representative sample of all possible configurations [25, 36].

We return to the results in Section 4. Given a large configuration space, certain configuration options are more likely to trigger performance bugs than the others (Finding 11). In contrast to the standard configuration-aware testing [25], when doing performance testing, developers can prioritize configuration options that are more relevant to performance. Our results also suggest that performance testing can focus on one or two configuration options (Finding 5).

Although source code is the ground truth [15] of the configuration space, it may not be available to developers who want to test hidden configurations. Fortunately, our results indicate that only a small portion of configuration options are hidden options, and hence it is less likely to have caused performance issues (Finding 7) .

System-level configurations are important. As our results have shown, a significant portion of performance bugs are related to system-specific configurations (Finding 3). This implies that developers need to test their applications under different system settings that can closely resemble production. In addition, when a performance bug occurs, developers should not limit their searching efforts restricted by the target application scope but the underlying executing environment as well.

Profiling is helpful for identifying misconfigurations. Profiling is frequently used to bootstrap performance diagnosis and allows developers to collect various system statistics (internal symptoms) such as CPU utilization, cache hit/miss rate and synchronization time. Given the results of our study, we have provided insights about linking each configuration domain with its internal symptoms (see Table 7). Such information could be of help to pinpoint configuration options when using profilers to observe anomalous system statistics (e.g., the low cache hit rate may be associated with the configuration options in the memory domain).

5.2 Implications to Researchers

Testing and debugging tools are needed. As we have seen, the current state of research in testing for performance bugs consider two major aspects – test inputs and test oracles [21, 23]. Yet this is not realistic for highly-configurable applications. We have seen in many cases, exposing bugs require both specific inputs and configuration options. We need, therefore, new configuration-aware techniques to test for and debug performance bugs.

One possibility is to leverage existing static analyses [19, 26] to identify performance-sensitive configuration options based on code patterns. Such options can be used to guide performance testing. In the example of Figure 4, the configuration options related to `util_ald_alloc` are performance-sensitive. Therefore, effective techniques should be developed to unify the mapping of configuration items back to the code base.

To facilitate performance debugging and diagnosis, we need to link the erroneous behavior to the buggy code, inputs and configuration options. Most existing performance debugging techniques assume that inputs and configurations are available and only identify buggy code that leads to performance issues [30]. However, when a performance bug occurs, users may not even think of configuration as a cause of their problems. While existing configuration debugging and diagnosis techniques [4, 40] can address general functional bugs, real-world performance bugs are more difficult to handle [16, 30]. We need, therefore, new performance debugging techniques that can isolate bugs caused by misconfigurations and link the bug to specific configuration options.

Configuration-aware regression testing is needed. As software evolves, new performance bugs can be introduced (e.g., the Apache bug 58037 in Section 4.3). Regression testing is used to perform re-validation of evolving software. To date, most regression testing research has focused on selecting, reducing, prioritizing, and augmenting test inputs [38] while treating software systems as if they possessed a single homogeneous configuration. In addition, existing regression testing techniques do not target performance bugs. It is possible that we can first apply static impact analysis to identify configurations that are affected by performance-sensitive code changes. We could then apply regression testing techniques to detect performance regressions.

Building performance-influence configuration models. A performance-influence model can be used to describe how configuration options and their interactions influence the performance of a software system. There has been a great deal of research on building such models to help developers predict performance [12, 18, 29] through various techniques such as sampling and machine learning. We believe that these black-box techniques can be improved if performance-sensitive configuration options and their associated code elements can be better understood. First, in a system with enormous configuration options, minimizing learning set is still a challenge. Identifying key configuration options will mitigate such problems. Second, black-box techniques only approximate the performance influence induced by various combinations of configuration options. As such, more sophisticated techniques are needed to analyze these options,

such as the dependency analysis. Finally, system-specific configuration options can be used to model environment for building performance models.

Fixing and avoiding performance bugs. Many software systems require continued operation even if an erroneous condition is met. For example, self-adaptive software systems provide adaptation mechanisms that allow continued operation when the system environment changes. In the case of performance bugs, a self-adaptive system should be able to reconfigure its settings to meet the performance requirements. If we return to our results of the study, the performance bugs always show certain internal symptoms such as high CPU utilization and low cache hits (Finding 10). Monitors can be used to capture such information, so that the application is reconfigured to certain settings if a performance bug symptom manifests. Research has shown that workarounds can be found to adjust the runtime configurations [32], so it is possible that we can leverage some of those ideas for this work.

Extracting new configurations. We have seen two cases where the bug fix involves introducing new configuration options, such as the example in Figure 2. While current reverse engineering techniques can extract existing configuration options from the code [26], they cannot infer new configuration options. In order to extract performance-influence configuration options, we need to understand how a certain performance-related feature has been implemented. It is possible that we could leverage ideas of feature localization [9] to derive correspondences between configuration options and computational units, yet this may also yield unnecessary options such that changing their values will not impact performance. Therefore, we need new techniques that can both infer performance features from the code, and determine which features are useful as configuration options.

6. RELATED WORK

There has been a great deal of work on configuration-aware techniques [3, 26, 37, 40]. For example, Yin et al. [37] study a number of configuration bugs to understand the configuration errors in commercial and open source systems. Rabkin et al. [26] propose a static analysis technique to extract configuration options from Java code. There has been a large body of work in the testing community that demonstrates the need for configuration-aware testing techniques and proposes methods to sample and prioritize the configuration space [25, 36]. Zhang et al. [40] have proposed a technique to diagnose crashing and non-crashing errors related to software misconfigurations. There has also been recent work that uses configurability as a way to avoid failures through self-adaptation [32]. However, none of these work deals with performance bugs.

There has been some work on empirical study for performance bugs [16,20,39]. For example, Zaman et al. [39] study the bug reports for performance and non-performance bugs in Firefox and Chrome. Their study found that performance bugs are more difficult to handle than non-performance bugs. Jin et al. [16] study 109 performance bugs from five software projects. While the above work provides insights and guidance on addressing performance bugs in general, it does not study in depth about performance bugs in highly-configurable software systems.

There has been recent work on testing, debugging, fixing and avoiding performance bugs [7, 13, 17, 21, 22]. For example, Nistor et al. [21] identify loops whose computation has repetitive memory-access patterns. StackMine mines call stack traces to discover call sequences with a high performance impact [13]. Pradel et al. [23] generate performance test cases. Grechanik et al. [11] select test cases for performance testing. While the above techniques are inspiring and effective, they assume the default configurations and do not consider performance bugs caused by configurations. Foo et al. [10] use ensemble learning techniques to detect performance regressions due to environment-specific variations. Their work focus on system-specific configurations, whereas we have studied in depth a wide range of configurations.

From the performance modeling perspective, there has been much work on constructing performance models for various purposes [12, 18, 28, 29, 33], such as using learning approach to find performance influential configurations [29], creating performance models by profiling [18], and performance modeling by static and dynamic program analysis techniques [33]. All these techniques provide good insights about factors involved in the performance models. However, our study analyzes more thoroughly on how performance bugs are related to configurations, and thus complementary.

7. CONCLUSION

We have performed a comprehensive characteristic study on 113 configuration-related performance bugs collected from three popular open source projects. With the increasing significance of performance bugs, this paper provides the first study on real-world performance bugs in highly-configurable software systems. Our study covers a wide spectrum of characteristics, including types, causes, symptoms, and fixes. The study provides guidance for future research on performance testing and debugging. We intend to help developers and practitioners to extend and improve tools that can address performance issues in highly-configurable applications. This work is only a starting point for understanding performance bugs related to configurations. In the future, we will extend our study on more subject programs and propose techniques to handle these bugs.

8. ACKNOWLEDGMENTS

This work was supported in part by NSF grant CCF-1464032.

9. REFERENCES

[1] Automated Combinatorial Testing for Software, 2016. http://csrc.nist.gov/groups/SNS/acts/index.html.

[2] J. Aranda and G. Venolia. The secret life of bugs: Going past the errors and omissions in software repositories. In *ICSE*, pages 298–308, 2009.

[3] M. Attariyan, M. Chow, and J. Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *OSDI*, pages 307–320, 2012.

[4] M. Attariyan and J. Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *OSDI*, pages 1–11, 2010.

[5] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and balanced?: Bias in bug-fix datasets. In *ESEC*, pages 121–130, 2009.

[6] Bugzilla keyword descriptions, 2016. https://bugzilla.mozilla.org/describekeywords.cgi.

[7] J. Burnim, S. Juvekar, and K. Sen. Wise: Automated test generation for worst-case complexity. In *ICSE*, pages 463–473, 2009.

[8] M. B. Cohen, M. B. Dwyer, and J. Shi. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *TSE*, 34(5):633–650, 2008.

[9] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *TSE*, 29(3):210–224, 2003.

[10] K. C. Foo, Z. M. J. Jiang, B. Adams, A. E. Hassan, Y. Zou, and P. Flora. An industrial case study on the automated detection of performance regressions in heterogeneous environments. In *ICSE SEIP*, pages 159–168, 2015.

[11] M. Grechanik, C. Fu, and Q. Xie. Automatically finding performance problems with feedback-directed learning software testing. In *ICSE*, pages 156–166, 2012.

[12] J. Guo, K. Czarnecki, S. Apel, N. Siegmund, and A. Wasowski. Variability-aware performance prediction: A statistical learning approach. In *ASE*, pages 301–311, 2013.

[13] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie. Performance debugging in the large via mining millions of stack traces. In *ICSE*, pages 145–155, 2012.

[14] Intel Hyper-Threading (HT) Technology, 2016. http://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html.

[15] D. Jin, X. Qu, M. B. Cohen, and B. Robinson. Configurations everywhere: Implications for testing and debugging in practice. In *ICSE SEIP*, pages 215–224, 2014.

[16] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. In *PLDI*, pages 77–88, 2012.

[17] M. Jovic, A. Adamoli, and M. Hauswirth. Catch me if you can: Performance bug detection in the wild. In *OOPSLA*, pages 155–170, 2011.

[18] Y. Kwon, S. Lee, H. Yi, D. Kwon, S. Yang, B.-G. Chun, L. Huang, P. Maniatis, M. Naik, and Y. Paek. Mantis: Automatic performance prediction for smartphone applications. In *USENIX ATC*, pages 297–308, 2013.

[19] M. Lillack, C. Kästner, and E. Bodden. Tracking load-time configuration options. In *ASE*, pages 445–456, 2014.

[20] A. Nistor, T. Jiang, and L. Tan. Discovering, reporting, and fixing performance bugs. In *MSR*, pages 237–246, 2013.

[21] A. Nistor, L. Song, D. Marinov, and S. Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *ICSE*, pages 562–571, 2013.

[22] O. Olivo, I. Dillig, and C. Lin. Static detection of asymptotic performance bugs in collection traversals. In *PLDI*, pages 369–378, 2015.

[23] M. Pradel, M. Huggler, and T. R. Gross. Performance regression testing of concurrent classes. In *ISSTA*, pages 13–25, 2014.

[24] E. Puoskari, T. E. J. Vos, N. Condori-Fernandez, and P. M. Kruse. Evaluating applicability of combinatorial testing in an industrial environment: A case study. In *JAMAICA*, pages 7–12, 2013.

[25] X. Qu, M. B. Cohen, and G. Rothermel. Configuration-aware regression testing: An empirical study of sampling and prioritization. In *ISSTA*, pages 75–86, 2008.

[26] A. Rabkin and R. Katz. Static extraction of program configuration options. In *ICSE*, pages 131–140, 2011.

[27] E. Reisner, C. Song, K.-K. Ma, J. S. Foster, and A. Porter. Using symbolic evaluation to understand behavior in configurable software systems. In *ICSE*, pages 445–454, 2010.

[28] E. M. Rodrigues, R. S. Saad, F. M. Oliveira, L. T. Costa, M. Bernardino, and A. F. Zorzo. Evaluating capture and replay and model-based performance testing tools: An empirical comparison. In *ESEM*, pages 9:1–9:8, 2014.

[29] N. Siegmund, A. Grebhahn, S. Apel, and C. Kästner. Performance-influence models for highly configurable systems. In *ESEC*, pages 284–294, 2015.

[30] L. Song and S. Lu. Statistical debugging for real-world performance problems. In *OOPSLA*, pages 561–578, 2014.

[31] 17 Key MySQL Config File Settings, 2015. http://www.speedemy.com/17-key-mysql-config-file-settings-mysql-5-7-proof/.

[32] J. Swanson, M. B. Cohen, M. B. Dwyer, B. J. Garvin, and J. Firestone. Beyond the rainbow: Self-adaptive failure avoidance in configurable systems. In *FSE*, pages 377–388, 2014.

[33] A. Tarvo and S. P. Reiss. Automated analysis of multithreaded programs for performance modeling. In *ASE*, pages 7–18, 2014.

[34] A. Wert, J. Happe, and L. Happe. Supporting swift reaction: Automatically uncovering performance problems by systematic experiments. In *ICSE*, pages 552–561, 2013.

[35] G. Xu, N. Mitchell, M. Arnold, A. Rountev, and G. Sevitsky. Software bloat analysis: Finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In *FoSER*, pages 421–426, 2010.

[36] C. Yilmaz, M. B. Cohen, and A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *TSE*, 29(4), July 2004.

[37] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *SOSP*, pages 159–172, 2011.

[38] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *STVR*, 22(2):67–120, Mar. 2012.

[39] S. Zaman, B. Adams, and A. E. Hassan. A qualitative study on performance bugs. In *MSR*, pages 199–208, 2012.

[40] S. Zhang and M. D. Ernst. Automated diagnosis of software configuration errors. In *ICSE*, pages 312–321, 2013.