Postprint of article in The 2015 Stephen S. Yau Academic Symposium (SSYAS '15), *Proceedings of the IEEE 39th Annual Computers, Software and Applications Conference Workshops* (*COMPSACW '15*), IEEE Computer Society (2015)

Slope-based Sequencing Yardstick for Analyzing Unsatisfactory performance of multithreaded programs

An SSYAU Trend Estimation Approach to Performance Bug Localization*

W.K. Chan
Department of Computer Science
City University of Hong Kong
Tat Chee Avenue, Hong Kong
wkchan@cityu.edu.hk

T.H. Tse
Department of Computer Science
The University of Hong Kong
Pokfulam, Hong Kong
thtse@cs.hku.hk

Shangru Wu, Y.T. Yu†
Department of Computer Science
City University of Hong Kong
Tat Chee Avenue, Hong Kong
shangru.wu@my.cityu.edu.hk
csytyu@cityu.edu.hk

Zhenyu Zhang
State Key Laboratory of Computer Science
Institute of Software
Chinese Academy of Sciences
Beijing, China
zhangzy@ios.ac.cn

***Abstract*—As users are increasingly concerned about energy efficiency, they are also increasingly intolerant of performance anomalies of programs that may cause significant energy waste. Bug localization is a bottleneck in the development of multithreaded programs. Although both static and dynamic performance bug localization techniques have been proposed, they cannot handle performance anomalies with unforeseen patterns, and cannot work well if the concept of performance anomaly is fuzzy or evolves over time for the same program. We propose a novel model-based approach to performance bug localization. The approach is based on curve fitting and trend estimation over program executions with performance data. We describe our trend estimation model and illustrate it with the result of a case study on locating three real-world performance bugs in *MySQL*.**

Keywords—performance bug, model-based approach, multithreaded program, bug localization

I. INTRODUCTION

Bug localization is a difficult task in program development. Ineffective bug localization severely affects the development schedule. As energy efficiency is becoming a major concern in the implementation of many programs [23], users are increasingly intolerant of program performance anomalies causing significant energy waste. This situation places heavy pressure on developers to locate performance bugs in programs. In particular, many important real-world programs such as *MySQL* [14] and Mozilla *Firefox* [6] are multithreaded. It is thus imperative to develop effective techniques to locate performance bugs in them. This paper proposes an innovative model-based trend estimation approach known as Slope-based Sequencing Yardstick for Analyzing Unsatisfactory program performance (SSYAU) to address this problem.

A *performance bug* [18] is a defect in a program such that its activation causes *poorer performance* (such as more time or resources taken) to compute outputs than expected, irrespective of the *functional correctness* of the execution.

Suppose we have a "performance-anomaly-free" version v1 of a program in mind and a performance metric such that a higher metric value indicates better performance. We say that an execution of the program version v2 is associated with a *performance anomaly* with respect to that performance metric if the metric value of the execution of v2 over an input is lower than that of v1 over the same input.

The performance of a program is subject to natural variations due to inherent non-determinisms in the program or its execution environment. A performance anomaly is not easily revealed unless or until it manifests itself with a conspicuous symptom, such as when an execution exhibits 10-fold slowdown or 10-fold resources consumption. In many cases, it is simply hard to deem a program execution to be unassociated with any performance anomaly. But ignoring executions without obvious performance anomalies may miss the opportunity to locate detectable performance bugs. In the era of big data, we should both (1) avoid making premature decisions to label whether a program execution incurs a performance anomaly and (2) explore the possibility of extracting from any execution invaluable information that may be useful in a later stage.

* The acronym of our approach is "SSYAU" by design to celebrate Professor Stephen S. Yau's 80th birthday. For ease of understanding, however, we will simply use "trend estimation" as the name of the model in the paper.

† Corresponding author.

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

(a) A performance bug in Apache http server *httpd*

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 function.
	Using it inside **fastmutex** causes 40X slowdown in users' experiments.

(b) A performance bug in *MySQL*

Mozilla Bug 66461 and 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**.

(c) A performance bug in Mozilla *Firefox*

Figure 1. Three sample performance bugs presented in [10]. In each code listing, the line with "–" on the left is in the buggy version, the line with "+" in the left is in the bug-fixed version that replaces the "–" line. Other lines are in both the buggy and bug-fixed versions.

Hence, we contend that it is not the best option to locate performance bugs by either (a) investigating only the executions explicitly labeled with performance anomalies or (b) investigating only the executions that exhibit behavior not following the norm. We argue that performance bug localization approaches should be able to deal with the presence of (large and small) natural variations in executions.

MySQL [14] and *Chrome* [3] are widely used software applications whose architectures include multithreaded components. An earlier survey [10] confirms that popular multithreaded programs *do* exhibit many performance anomalies.

An execution of such programs involves numerous *threads* [7], each exercising a potentially nested sequence of functions. These threads often interleave with one another. Any thread may incur performance anomalies, and improper coordination among them may introduce another layer of such anomalies.

Fig. 1 shows three examples of performance anomalies in multithreaded programs presented in the literature [10]. Fig. 1(a) illustrates a performance bug when invoking the function apr_stat() for the Apache http server [1]. To retrieve the status of a file, the original function accepts an input parameter specifying the location of the file information. After upgrading to a new version, apr_stat() requires the calling module to specify the file type so as to return a result more efficiently, but also allows a caller to skip the file type for compatibility with the previous version. Although the *functional correctness* of a calling module that does not fulfill the latest interface requirement will not be adversely affected, retrieving the status of a file without specifying its type will be much slower than doing so with its type specified precisely. This *performance bug* is **sequential** in nature, that is, it manifests itself in sequential code execution and is not unique to multithreading. Performance bug locators should not ignore this kind of bug.

Fig. 1(b) shows a code excerpt from *MySQL* that uses a function random() to compute a random number. The use of random() is, however, not restricted to a single thread. To make it thread-safe, a shared lock is kept within random() to regulate the use of the critical section of the function. In *MySQL*, many threads may use random() in parallel, resulting in frequent lock contentions among multiple threads that queue up to acquire the shared lock. This bug is **concurrent** in nature, because it manifests itself during concurrent thread executions. Moreover, it causes some program threads to wait instead of doing computations, unnecessarily slowing down the execution.

Fig. 1(c) illustrates a performance bug in a multithreaded program that renders the shape of an object invisible on the canvas. The illustrated bug repair adds a variable check so that a short-circuited path is taken when the checking result is true. Locating this performance bug requires a semantic understanding that bypassing the short-circuited computation will improve performance while maintaining functional correctness. Performance bug locators should be able to deal with **semantic** bugs.

Both static and dynamic performance bug localization techniques have been proposed [4][5][8][9][10][12][22][24]. These techniques *either* use predefined access patterns (or rules) to mine the source code or executions of a program *or* compare between executions with and without performance problems to identify program entities that correlate with performance anomalies. They cannot handle applications having performance anomalies with unforeseen patterns, and cannot work well if the concept of performance anomaly is fuzzy or evolves over time for the same application. Unfortunately, the list of such patterns is open, and the same pattern may prompt a bug localization technique to report a true performance bug for one program but a false positive for another. Also, separating the above two types of runs can be unsystematic: runs without per-

formance anomaly may exercise the same code fragments that cause performance anomalies in other runs.

We propose a novel approach to localizing performance bugs in multithreaded programs. The base model was outlined in Zhang et al. [25]. This paper extrapolates it to performance bug localization. The key idea is to model execution with performance statistics of each function using trend estimation. Such a trend can be a line or a higher order polynomial. From our case study on locating three real-world performance bugs in *MySQL*, we observe that a performance bug tends to be associated with a lower order polynomial with a small fitting error than a higher order polynomial with the best fitting error.

The main contribution of this paper is twofold. First, it is the *first work* that presents a trend-estimation approach to performance bug localization for multithreaded programs. Second, it presents an exploratory case study that shows the feasibility of locating performance bugs using trend estimation.

The paper is organized as follows: We first review closely related literature in Section II. Then, we present the trend estimation model in Section III followed by a case study in Section IV. Section V concludes the paper.

II. Related Work

We classify related work into three board categories.

The **first category** consists of *pattern-based techniques* [5] [10]. They mine the source code or execution of a program and match it against a set of predefined patterns (or rules), each indicating the possible presence of a performance anomaly. *FindBugs* is a classic example with a repository of 27 performance bug patterns [5]. We found, however, that these patterns are rather low-level, such as "HSC: huge string constants are duplicated across multiple class files" (which locates a constant string copied from one class to another through sequences of function calls). Clearing all these anomalies still fails to locate the kinds of performance bugs (such as those illustrated in Section I) in real-world multithreaded programs.

Jin et al. [10] defined 50 manually crafted patterns based on a study of real-world performance bugs in multithreaded programs. There were also attempts to apply patterns identified from one program to locate performance bugs in another program. Xiao et al. [20] proposed to identify code fragments in loops (such as user interface threads) that may slow down owing to the execution of workload-heavy tasks.

Pattern-based techniques generally rely on manual identification and formalization of concrete patterns. Early recognition of performance bugs are impractical unless until a pattern has been identified or the usage profile has been changed significantly. Moreover, pattern-based approaches in general are well known to incur many false positives.

The **second category** consists of *performance profiling techniques* with or without code-based analysis, such as GNU *gprof* [9] and *StackMine* [8]. A performance trace is a time-ordered sequence of performance details. Each performance detail contains values of performance metrics or memory access metrics that characterize some aspects of the program performance. For instance, GNU *gprof* [9] generates such traces, which programmers may use to spot and revise parts of the code that are time-consuming. Unlike static pattern-based techniques, GNU *gprof* uses a dynamic approach to collect, for example, the execution count and other performance statistics. However, it does not have any extrapolation capability. As such, the bug in Fig. 1(b) is unlikely to be identified.

Algorithmic profiling [24] is the latest dynamic technique that estimates an empirical cost function for a data structure based on the performance data collected when the executions use the instances of this data structure. A key limitation, as pointed out by the authors [24], is that this technique has not been generalized to handle all parts/types of the program code. Their research group also developed a performance trace analyzer framework [4], on top of which customized analyzers using the framework primitives can be built to locate bugs by correlating program locations with a certain type of performance anomaly in mind. Their work does not address the concurrent aspect of multithreaded programs, however.

Some research work adds various criteria to identify performance bugs. One way is to integrate with the mining approach or specific patterns (say, those obtained by techniques in the first category). For instance, *StackMine* [8] has been applied to diagnose performance traces of the Microsoft Windows 7 production system. *StackMine* mines the call stacks over a set of performance traces to find patterns that may adversely correlate with the performance anomalies in the majority of the traces. It shows the feasibility of trace comparisons to locate bugs in multithreaded programs.

MacePC [12] first uses the executions with performance anomalies to extract performance anomaly patterns. It then applies model checking to explore the execution space similar to the former executions but without such patterns. It looks for the point in the earliest abstract state in the model checker that diverges into executions with and without the patterns.

Yan et al. [22] statically analyze a codebase to identify statements that unnecessarily manipulate the data structures by tracking how object references are passed through functions. TODDLER [15] monitors whether the code fragment of a loop may perform computations with repetitive or partially similar memory-access patterns across its iterations to spot performance anomalies.

The **third category** consists of *statistical correlation techniques*. Using various suspiciousness formulas, these techniques aim to locate faults in program entities by analyzing the distributions of passed and failed executions that exercise individual program entities (such as statements). One major class of techniques in this category is *spectrum-based fault localization* (*SBFL*). Well-known examples include *Tarantula* [11], *Ochiai*, *CBI*, *CP* [26], and *SOBER*. Theoretical analysis [21] has been performed to explain why one subclass of SBFL techniques can be more effective than another. SBFL is under active research to locate *semantic and functional* bugs in sequential programs, and has lately been adapted to locate concurrency functional bugs (such as *Falcon* [16] and *Racon* [13]). For example, *Tarantula* counts the proportions of failing executions (denoted by x) and passed executions (denoted by y) that exercise the same program statement. Then, it rates the statement associated with the largest value of $x/(x+y)$ as the most suspicious to be faulty. It is clear from the formula $x/(x+y)$ that *Tarantula* is blind to multiple bugs in a program (possibly exposed by the executions already). Moreover, most SBFL techniques require the availability of passed executions (or fragments of executions) to be effective [25]. Furthermore, the issue of coincidental correctness [19] (which means that a

passed execution may go through a buggy statement) severely limits the accuracy of SBFL techniques. Our experiment [19] on functional bugs reveals that SBFL techniques may become less effective than a random guess in some coincidental correctness scenarios.

Performance bug localization research is emerging. Intuitively speaking, for the three performance bugs illustrated in Fig. 1, an SBFL technique may count the number of times that certain statements within apr_stat() in Fig. 1(a) and within Draw() in Fig. 1(c) are executed, and the number of threads queuing to acquire the shared lock within random() in Fig. 1(b). These numbers are counted for both executions with and without performance anomalies and then correlated with the dissimilarity in the time taken to complete the executions.

Furthermore, the criteria to deem a particular execution to be suffering from performance anomaly may be fuzzy and are likely to vary across different applications. Jin et al. [10] pointed out that a piece of code may perform well until the usage of an application gradually changes over the years. For example, a transparent bitmap image is more extensively used in recent years to serve as the space separator to beautify the layout of a webpage when compared with 10 years ago.

III. Our Model

A. Basic Idea

The basic idea of our trend estimation model rests with the following bug hypothesis (H1) as a starting point: The more times that the execution of a multithreaded program invokes a suspicious program entity[1], the more likely is the execution related to performance anomalies.

A clear difference between our model and other SBFL techniques for functional bugs is that we do not distinguish between passed and failed runs, as our interest is in performance bugs. The model is based on the general statistical method of the same name for time series analysis. A widely adopted approach is to identify the best fit regression line using least-squares fitting, showing the tendency of the samples under study. Likewise, based on samples of the numbers of times that different executions exercise the same program entity, our base model identifies a best fit regression line. It then calculates the signal-to-noise ratio [17] from the slope of the regression line and the value of the fitting error. This ratio is used to estimate the relevance of the program entity to a performance bug.

Our approach generalizes individual performance data samples for a program entity into an intuitively neat function, which opens up a brand new direction for theoretical analysis. We note that *algorithmic profiling* [24] produces an empirical cost function but such a function does not have a clear trend.

B. Modeling

Consider a program modeled by a list of *program entities* such as statements and functions. Suppose we are given a set of overall performance metric values of the program executions (such as the processing time of each execution). Given any program entity, our trend estimation model divides the set of program executions into disjoint *partitions* such that each execution in the same partition invokes the program entity exactly the same number of times (say, c times). For the theoretical development of our model, we further presume a certain value of the *anomaly rate* $F(c)$, that is, the proportion of executions with performance anomalies in each partition. Interestingly, the need for this presumed value in our model will be eliminated at the end.

It is important to clarify that in the context of performance bug localization, the term "anomaly rate" is related to executions with performance anomalies rather than the exhibition of output incorrectness, unlike the term "failure rate" in the testing of functional properties or the localization of functional faults.

Note also that one of our targets is to establish a means of locating performance bugs *without* having to determine the anomaly rate. In the present phase, we discuss the problem from a *theoretical* perspective to formulate the key notions and model the program entities involved, *assuming* that the anomaly rate is known. In a later phase of our model (see Section III.E), we will present how to eliminate the need.

C. Recalibration

Given a program entity s, $F(0)$ denotes the anomaly rate of the partition in which all the executions in the partition never invoke s. Conceptually, if none of the executions ever invokes s, the latter should not be related to any performance anomaly exhibited by the executions in question. Hence, if $F(0)$ for s is nonzero, it should be reset to zero. Similarly, other partitions for the same s may have also overestimated their anomaly rates by the amount $F(0)$. To compensate for this systematic error, our trend estimation model calculates a *recalibrated anomaly rate* $G(c) = F(c) - F(0)$, which gives a more realistic estimate of the probability that the program entity exhibits performance anomalies when exercised *exactly c times*.

Intuitively, if the program entity in question is within the performance bug region and yet $F(0)$ is nonzero, it may indicate the existence of at least one other performance bug that our trend estimation model can be iteratively applied to locate. We will leave the reporting of the iterative strategy for locating multiple performance bugs to future work.

D. Trend Fitting

In the fitting phase, our model estimates the trend for each program entity to exhibit performance anomalies according to the recalibrated anomaly rates. Given any program entity, when $G(c)$ is defined, we may model $\langle c, G(c)\rangle$ as a point in two-dimensional space. Following the bug hypothesis H1, for a problematic program entity, $G(c)$ should intuitively possess the characteristics of a discrete *monotonically increasing* function. Our base model estimates the relevance of a program entity to a performance bug using a best fit regression line in two-dimensional space.

To establish the base model, we consider the probability that *invoking s* ***exactly c times*** *does not lead to a performance anomaly*. This probability is given by $(1-p)^c$, where p is the probability that *invoking s* ***only once*** *results in a performance anomaly*. The same probability can also be computed directly as $1 - G(c)$. (Note that we have not lifted the theoretical assumption on the identification of non-problematic executions with respect to the performance anomalies.) Equating these two probabilities, we obtain $G(c) = 1 - (1-p)^c$.

[1] Such as a function to retrieve the status of a file using apr_stat() in Fig. 1(a), a lock acquisition call within random() in Fig. 1(b), and a call to render an image using Draw() in Fig. 1(c).

Under suitable mathematical conditions, a function $f(x)$ may be represented by an infinite Maclaurin series[2]

$$f(x) = f(0) + f^{(1)}(0)\frac{x}{1!} + f^{(2)}(0)\frac{x^2}{2!} + \cdots$$

where $f^{(i)}(0)$ is the *i*th derivative of $f(x)$ at the point $x = 0$. In the base model, we use a linear function for trend estimation. Thus, the recalibrated anomaly rate can be approximated by $G(c) = G(0) + G^{(1)}(0)\frac{c}{1!} = -\log(1-p)\cdot c$, or simply as $G(c) = \ell \cdot c$, where $\ell = -\log(1-p)$ is a constant. In this way, the recalibrated anomaly rate can be fitted by a regression line passing through $\langle 0, 0\rangle$ with slope ℓ.

In general, the function $G(c)$ may be a higher order polynomial. Indeed, our preliminary case study to be presented in Section IV shows that when performance anomalies are present, the corresponding curve may be nonlinear.

For the moment, let us continue to take the linear form of the polynomial function to present this part of trend estimation. Our model applies least-squares analysis to minimize the error in regression line fitting. For a given program entity, the mean slope $\overline{\ell}$ and standard deviation σ are given by

$$\overline{\ell} = \sum_{c\in D}\big(c \cdot G_i(c)\big) \,/\, \sum_{c\in D} c^2$$

$$\sigma = \sqrt{\sum_{c\in D}\big(G(c)\big)^2 - \big[\sum_{c\in D}\big(c\cdot G(c)\big)\big]^2 / \sum_{c\in D} c^2}$$

where D is the set of possible numbers of times that any program execution may invoke the given program entity.

However, the number of times that an execution invokes a program entity is only a relative concept and may be weighted differently among various program entities. Hence, the mean slope $\overline{\ell}$ for each specific program entity *s* should be normalized by a factor $c_{\max,s}$ before comparison, where $c_{\max,s}$ is the largest possible number of times that any execution may invoke *s*.

Inspired by the notion of the signal-to-noise ratio [17], for each program entity, our base model determines *ranking score* *R* as the ratio of the mean to the standard deviation, thus:

$$R = \overline{\ell} \,/\, \sigma$$

The ranking score estimates the relevance of each program entity to performance bugs when executions with and without performance anomalies can be distinguished. The higher the value of *R*, the more suspicious will be the program entity *s*.

The range of the ranking score *R* is $[-\infty, +\infty]$. In other words, both $-\infty$ and $+\infty$ are valid ranking scores. If a program entity *s* comes with no sample point, it means that none of the executions with performance anomalies has invoked *s* before resulting in a performance problem. Hence, *R* is assigned a value of $-\infty$, meaning that *s* is the least suspicious with respect to any performance anomalies demonstrated by the executions. If a program entity comes with only one sample point, the slope $\overline{\ell}$ is undefined, and we tentatively assign a value of 0 to *R* because the program entity has not yet been demonstrated to exhibit performance anomalies in more than one execution. We will investigate whether this assumption holds and rectify the value assignment accordingly. If the standard deviation is zero, we have two sub-cases: (1) If *s* comes with one sample point, *R* is tentatively assigned a value of 0 just as before. (2) If *s* comes with more than one sample point, *R* is computed by taking the limit, resulting in a value of $+\infty$.

We will leave the formulation of the ranking score for higher order polynomial functions to future work.

E. Eliminating the Dependency on Anomaly Rate

In this section, we would like to eliminate the need to find the number of executions without performance anomalies as presented in last section. Again, we illustrate the process using the best fit regression line model.

For any program entity *s*, let $N(c)$ be the number of program executions such that each execution invokes *s* exactly *c* times. We first determine the mean number of executions $\overline{N}$ irrespective of the value of *c*. We then replace every $N(c)$ by $\overline{N}$ in the calculation of *R* to obtain an estimator $\tilde{R}$ as follows:

$$\tilde{R} = \frac{c_{\max,s}\sum_{c\in D}(c\cdot(Y(c)-Y(0)))\,/\sum_{c\in D}c^2}{\sqrt{\sum_{c\in D}(Y(c)-Y(0))^2 - [\sum_{c\in D}(c\cdot(Y(c)-Y(0)))]^2/\sum_{c\in D}c^2}}$$

where $Y(c)$ is the number of runs with performance anomalies such that each run invokes *s* exactly *c* times.

$\tilde{R}$ only depends on executions *with* performance anomalies. As we have mentioned in Section I, it is not practical enough to deem which execution may incur a performance issue before applying a performance bug localization technique. To tackle this issue, we will further explore the technique design space. For instance, one may fit the points $\langle c, N(c)\rangle$ by a polynomial function $h(x)$ with a monotonically increasing trend. Then, each $Y(c)$ in $\tilde{R}$ is replaced by $h(c)$.

The elimination phase is optional if the set of executions without performance anomalies can be clearly identified. Note that this phase depends on the calculation of *R* using the signal-to-noise ratio concept as presented in the last section, which is only a linear approximation in our base model.

IV. Exploratory Case Study

We have conducted an exploratory case study on the application of our trend estimation model to bug localization. The aim of our case study is to explore the possible setting in the design and solution spaces rather than verifying the proposed model. The empirical results are summarized in this section.

The first phase of the case study [25] is on *functional* bugs. It shows that, in most cases, the use of only failed executions to locate functional bugs is as effective as existing SBFL techniques that use both passed and failed executions. A clear message is that if there is indeed valuable information in the passed runs, current SBFL techniques have not been fully successful in utilizing it.

The second case study is on *performance* bugs. We used the keyword "performance" to search the *MySQL* bug repository and picked three closed bug reports, each describing a bug-fix.

We executed *MySQL* 5.5 and *MySQL* 5.0, respectively, over 10 randomly selected test cases taken from the *MySQL* repository. Our experiment was conducted on Ubuntu Linux 10.04 configured on a 3.16GHz Intel Core 2 Duo processor with 3.25GB physical memory. Following [2], we measured the times needed to execute the test cases using the time command of the Linux system. We plotted the execution times against the numbers of invocations of the functions described in the bug reports. The results are shown in Fig. 2.

[2] A Taylor series represents $f(x)$ by an infinite series in terms of the *i*th derivative of $f(x)$ at the point $x = a$ for some constant *a*. A Maclaurin series is a special case of a Taylor series such that $a = 0$.

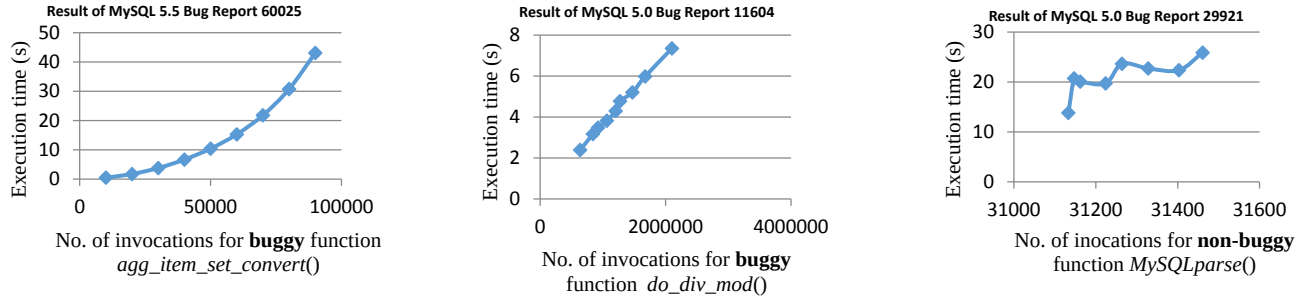

Figure 2. Performance versus execution count graphs for functions in three real-world MySQL bug reports. Both subfigures (a) and (b) show the buggy functions that have been fixed in bug reports 60025 and 11604 while subfigure (c) shows the non-buggy function mentioned in bug report 29921.

The two functions in Figs. 2(a)–(b) are *buggy* whereas the function in Fig. 2(c) is *non-buggy* but simply mentioned in the bug report. In each case, we make a best-fit polynomial. We observe that the buggy functions in Figs. 2(a)–(b) closely resemble a polynomial of lower order than the curve in Fig. 2(c). Moreover, the curve in Fig. 2(c) also disagrees with our base model that hypothesizes that a fitted curve resembles a monotonically increasing function.

By the nature of curve fitting, it is possible to fit the data points in Figs. 2(a)–(b) by a polynomial of much higher order. We did make such exploratory attempts but found that the fitted curves would not monotonically increase. Moreover, the corresponding curve fitting errors were not further reduced by any significant amount (and certainly not by an order of magnitude). By contrast, when we attempted to fit the data points in Fig. 2(c) by a polynomial of much lower order, the curve fitting error increased by at least an order of magnitude.

From the case study, it appears feasible to use polynomial fitting and *relative* polynomial orders to assess whether a program entity can be more relevant to performance anomalies.

V. Conclusion

We have presented a trend estimation approach to performance bug localization. We have outlined an exploratory case study on three real-world performance bugs that were present in *MySQL*. As ongoing and further work, we will continue to enhance the effectiveness and applicability of our approach.

Acknowledgment

This research is supported in part by the General Research Fund of the Research Grants Council of Hong Kong (project nos. 11201114, 125113, 716612, and 717811) and the National Natural Science Foundation of China (project no. 61379045).

References

[1] *Apache HTTP Server Project*, http://httpd.apache.org/.

[2] Y. Cai and W.K. Chan, "MagicFuzzer: Scalable deadlock detection for large-scale applications," *Proceedings of the 34th ACM/IEEE International Conference on Software Engineering* (*ICSE '12*), IEEE Computer Society, 2012, pp. 606–616.

[3] *Chromium*, http://www.google.com/chrome.

[4] A. Diwan, M. Hauswirth, T. Mytkowicz, and P.F. Sweeney, "TraceAnalyzer: A system for processing performance traces," *Software: Practice and Experience*, vol. 41, no. 3, 2011, pp. 267–282.

[5] *FindBugs Bug Descriptions*, University of Maryland, http://findbugs.sourceforge.net/bugDescriptions.html.

[6] *Firefox*, Mozilla, http://www.mozilla.org/firefox/.

[7] C. Flanagan and S.N. Freund, "FastTrack: Efficient and precise dynamic race detection," *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI '09*), ACM, 2009, pp. 121–133.

[8] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie, "Performance debugging in the large via mining millions of stack traces," *Proceedings of the 34th International Conference on Software Engineering* (*ICSE '12*), IEEE Computer Society, 2012, pp. 145–155.

[9] M. Honeyford, *Speed your code with the GNU profiler*, IEEE DeveloperWorks Linux Technical Library, 2006, http://www.ibm.com/developerworks/library/l-gnuprof.html.

[10] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI '12*), ACM, 2012, pp. 77–88.

[11] J.A. Jones, M.J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," *Proceedings of the 24th International Conference on Software Engineering* (*ICSE '02*), ACM, 2002, pp. 467–477.

[12] C. Killian, K. Nagaraj, S. Pervez, R. Braud, J.W. Anderson, and R. Jhala, "Finding latent performance bugs in systems implementations," *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (*FSE '10*), ACM, 2010, pp. 17–26.

[13] B. Lucia, B.P. Wood, and L. Ceze, "Isolating and understanding concurrency errors using reconstructed execution fragments," *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI '11*), ACM, 2011, pp. 378–388.

[14] *MySQL*, http://www.mysql.com/.

[15] A. Nistor, L. Song, D. Marinov, and S. Lu, "Toddler: Detecting performance problems via similar memory-access patterns," *Proceedings of the 2013 International Conference on Software Engineering* (*ICSE '13*), IEEE Computer Society, 2013, pp. 562–571.

[16] S. Park, R.W. Vuduc, and M.J. Harrold, "Falcon: Fault localization in concurrent programs," *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering* (*ICSE '10*), vol. 1, ACM, 2010, pp. 245–254.

[17] E. Säckinger, Broadband Circuits for Optical Fiber Communication, John Wiley, 2005.

[18] L. Song and S. Lu, "Statistical debugging for real-world performance problems," *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (*OOPSLA '14*), ACM, 2014, pp. 561–578.

[19] X. Wang, S.C. Cheung, W.K. Chan, and Z. Zhang, "Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization," *Proceedings of the 31st International Conference on Software Engineering* (*ICSE '09*), IEEE Computer Society, 2009, pp. 45–55.

[20] X. Xiao, S. Han, D. Zhang, and T. Xie, "Context-sensitive delta inference for identifying workload-dependent performance bottlenecks," *Proceedings of the 2013 International Symposium on Software Testing and Analysis* (*ISSTA '13*), ACM, 2013, pp. 90–100.

[21] X. Xie, T.Y. Chen, F.-C. Kuo, and B. Xu, "A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization," *ACM*

Transactions on Software Engineering and Methodology, vol. 22, no. 4, 2013, pp. 31:1–31:40.

[22] D. Yan, G. Xu, and A. Rountev, "Uncovering performance problems in Java applications with reference propagation profiling," *Proceedings of the 34th International Conference on Software Engineering* (*ICSE '12*), IEEE Computer Society, 2012, pp. 134–144.

[23] C. Yang, C. Jia, W.K. Chan, and Y.T. Yu, "On accuracy-performance tradeoff frameworks for energy saving: Models and review," International Workshop on Software Quality and Management (SQAM '12), *Proceedings of the 19th Asia-Pacific Software Engineering Conference* (*APSEC '12*), vol. 2, IEEE Computer Society, 2012, pp. 58–65.

[24] D. Zaparanuks and M. Hauswirth, "Algorithmic Profiling," *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI '12*), ACM, 2012, pp. 67–76.

[25] Z. Zhang, W.K. Chan, and T.H. Tse, "Fault localization based only on failed runs," *IEEE Computer*, vol. 45, no. 6, 2012, pp. 64–71.

[26] Z. Zhang, W.K. Chan, T.H. Tse, B. Jiang, and X. Wang, "Capturing propagation of infected program states," *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering* (*ESEC '09/FSE-17*), ACM, 2009, pp. 43–52.