Performance mutation testing: hypothesis and open questions

Ana B. Sánchez[a,*], Pedro Delgado-Pérez[b,*], Sergio Segura[a], Inmaculada Medina-Bulo[b]

[a] *ETS Ingeniería Informática, Universidad de Sevilla, Spain*
[b] *Escuela Superior de Ingeniería,Universidad de Cádiz, Spain*

Abstract

Performance bugs are common, costly, and elusive. Performance tests aim to detect performance bugs by running the program with specific inputs and determining whether the observed behaviour is acceptable. There not exist mechanisms, however, to assess the effectiveness of performance tests. Mutation testing is a technique to evaluate and enhance functional test suites by seeding artificial faults in the program under test. In this new idea paper, we explore the applicability of mutation testing to assess and improve performance tests. This novel approach is motivated with examples and open questions.

Keywords: performance testing, mutation testing, performance bugs

1. Introduction and motivation

Performance bugs are those programming errors that lead to significant performance degradation while preserving the program functionality [1]. Identifying these bugs are of primary importance: they cause poor user experience, low system throughput and waste of resources, which can easily frustrate users and eventually result in considerable loss of money. For instance, consider the performance bug found in the JFreeChart software shown in Figure 1. The bug arises because the code traverses a dataset to compute a value called $xxWidth$. However, if the dataset is not modified between successive calls to $drawItem$, the recomputation of $xxWidth$ in each call is redundant and, therefore, exhibits a serious performance bug that should be fixed [1]. Such bugs are surprisingly common: 7,603 performance bugs were identified in Mozilla Firefox, and 510 in Google Chrome [2].

Performance bugs are different from functional bugs and require special testing care. In general, the definition of performance defects is imprecise due to

☆This work was partially supported by the European Commission (FEDER) and the Spanish Goverment projects BELI TIN2015-70560-R and DArDOS TIN2015-65845-C3-3-R.

*Corresponding authors (anabsanchez@us.es,pedro.delgado@uca.es)

```
public boolean render(Graphics2D g2, Rectangle2D dataArea, ...){
        ...
        for(item=first; item <= last; item++){
                renderer.drawItem(dataset, series, item...);
        }
        ...
}

public void drawItem(XYDataSet dataset, int series, int item, ...) {
        ...
        for(int i=0; i < itemCount; i++){
                xxWidth=Math.min(xxWidth,Math.abs(pos-last));
        }
        ...
}
```

Figure 1: Real performance bug in JFreeChart.

the oracle problem (e.g., how slow a computation should be to be considered a performance bug). As such, they need much more time and effort to be detected and fixed than functional bugs [3]. Performance bugs are typically detected by running the program under test with specific inputs and checking whether the observed performance (e.g., execution time) is within the expected boundaries by using test cases and profilers. However, the selection of suitable inputs and the assessment of the observed performance are challenging tasks [4, 5]. Also, there is a lack of mechanisms to evaluate and improve the quality of performance tests, which allow many performance bugs to remain unrevealed [5].

Mutation testing is nowadays deemed as a powerful method to measure the fault-revealing ability of test suites. It is based on the injection of faults (*mutations*) through some predefined rules (*mutation operators*) that are helpful to assess the adequacy of a test suite in detecting plausible coding errors. This technique has been mainly focused on source-code transformations to uncover existing deficiencies in functional test suites. Even though this injection of mutations has also been extrapolated to many other software engineering activities [6], its use on non-functional properties is limited. An exception to this could be those systems where properties like the execution time are especially relevant and treated as a part of the functional specification. For example, in a related paper, timed automata models –representing timeliness properties of the program under test– were mutated to detect timing issues [7], instead of testing them at the source-code level. Also, software refactoring based on code mutations was applied to optimize the performance of a program [8], which is different from our goal of assessing the quality of tests in detecting performance issues. In this paper, we present different novel approaches to the generation of mutants oriented to testing non-functional properties on the basis of the following hypothesis:

> ***Hypothesis:*** *Performance mutation testing can help to evaluate and enhance the fault detection capability of performance tests.*

2. Performance mutation testing

In this section, we propose two different strategies for the application of performance mutation testing, described below.

2.1. Performance mutants

This approach consists in defining specific performance mutation operators that introduce performance faults in the program under test, i.e., changes in the code that degrade the performance of the program without altering its functionality. We distinguish two different types of performance operators, depending on whether they require preprocessing the program code to preserve its functionality, or not, namely:

Context-dependent mutation operators. These operators introduce faults that mimic real bugs by cleverly mutating code fragments. For instance, as noted by Olivo et al. [1], the performance bug in Figure 1 can be fixed by previously calculating the value of $xxWidth$ and passing it as an argument to $drawItem$:

```
public boolean render(Graphics2D g2, Rectangle2D dataArea, ...){
        ...
        double xxWidth= calculateXxWidth(dataset);
        for(item=first; item <= last; item++){
                renderer.drawItem(dataset, series, item, ..., xxWdith);
        }
        ...
}
```

Inspired by this performance bug, we could define a mutation operator that, given the correct solution shown above, inverts the situation to recreate the code in Figure 1: the operator would move the statement to calculate $xxWidth$ into the $drawItem$ function. Similarly, Xu et al. [9] identified a common performance degradation caused by the unnecessary creation of $GregorianCalendar$ objects inside a loop. Based on this idea, we could define an operator that moves the object creation statements located before loops' openings into the loops:

```
/* Original program version */

GregorianCalendar g =
        new GregorianCalendar();
for (int i=0; i<l.size(); i++){
    ...
}
```

```
/* Mutant program version */

for (int i=0; i<l.size(); i++){
    GregorianCalendar g =
            new GregorianCalendar();
    ...
}
```

The mutant creates many objects inside the loop instead of just one outside the loop. Theoretically, a performance mutant should be functionally equivalent to the original program, i.e., it should preserve the program functionality. Going back to the previous examples, before generating the mutants, we should check that the value of $xxWidth$ and the object g keep the same state between successive iterations in the *for* loop. This is somehow connected with program optimizations performed by compilers, e.g. loop-invariant code motion. This leads to the following open questions:

Question set 1: *How can the program be preprocessed to ensure its functionality is not altered by the mutation? What is the cost of such preprocessing? What is the connection with program optimization techniques?*

Question set 2: *How many context-dependent performance mutations operators can be identified? How often can they be applied?*

Context-independent mutation operators. This approach proposes the definition of mutation operators that are not dependent on the context, thereby preventing that functional errors arise. As an example, we could define an operator that randomly inserts a noticeable delay into the code by calling a *sleep* method into a loop (similar mutation operators have been proposed for multithreaded applications but with the goal of modifying the program's functionality). Operators like this simulate the effect that real coding errors would produce. For instance, the invocation to *sleep* resembles whatever sequence of unnecessary instructions, such as in the previous examples, where a value or object was needlessly recomputed. This fact leads to the following question:

Question 3: *How close are context-independent performance mutants to modelling real performance bugs?*

Another type of performance mutation operators that can be considered are those based on programming "anti-patterns" or "bad practices". These habits often introduce defects that have a negative impact on the program performance (but not in the functionality). For instance, the following mutation example, based on a real defect [1], transforms a set collection (s) into a generic list (ls) before checking whether the collection contains certain elements. This change worsens the execution time because generic types of collections have a slow containment checking method that is invoked a linear number of times.

```
/* Original program version */

boolean f(HashSet<Foo> s,
          ArrayList<Foo> l){

for(int i=0; i < l.size(); i++){
    Foo elem = l.get(i);
    if(s.contains(elem)){
        return true;
    }
    return false;
}
```

```
/* Mutant program version */

boolean f(HashSet<Foo> s,
          ArrayList<Foo> l){
List<Foo> ls = new ArrayList<Foo>(s);
for(int i=0; i < l.size(); i++){
    Foo elem = l.get(i);
    if(ls.contains(elem)){
        return true;
    }
    return false;
}
```

Question 4: *How effective are mutation operators based on "bad practices" in evaluating and improving performance tests?*

2.2. Functionally-equivalent mutants

Functional mutants have been traditionally produced under the assumption that they only model functional errors. As such, surviving mutants are reviewed and those that have the same functionality as the original program are then tagged as equivalent (this is an undecidable problem and, therefore, it cannot be fully automated). However, we might have overlooked the possibility that those equivalent mutants do show a performance degradation. As an example, Delgado-Pérez et al. [10] noticed that the number of iterations in a loop could increase because of a mutation, but those additional iterations did not affect the functionality of the program; they resorted to a timeout to kill that mutant. As a result, we now propose going a step further and reusing functionally-equivalent mutants, which have always been regarded as a stumbling block in mutation testing. This is an innovative way to search for performance defects because, to our knowledge, the effect of these mutations on the performance has never been studied before. At this point, we wonder:

> ***Question 5:*** *How many functionally-equivalent mutants created with traditional operators can be reused to assess performance tests?*

The term *equivalence* leads to a question transversal to all these strategies:

> ***Question set 6:*** *What is an equivalent mutant in performance mutation testing? What criteria should be used to detect them? How many of them are generated and what is the cost of their identification?*

3. Conclusion

This paper presents open questions on performance mutation testing that have remained unexplored so far. On the one hand, we propose the generation of performance mutants based on real bugs seeking not to alter the semantics of the program. On the other hand, we suggest giving a new opportunity to mutants that are found to be functionally equivalent to search for performance defects. Previous research challenges need to be resolved and they are key to enhancing the ability of tests to reveal performance bugs.

4. References

[1] O. Olivo, I. Dillig, C. Lin, Static detection of asymptotic performance bugs in collection traversals, in: ACM SIGPLAN Conference on Programming Language Design and Implementation, 2015, pp. 369–378.

[2] S. Zaman, B. Adams, A. E. Hassan, A qualitative study on performance bugs, in: IEEE Working Conference on Mining Software Repositories, 2012, pp. 199–208.

[3] S. Segura, J. Troya, A. Durán, A. Ruiz-Cortés, Performance metamorphic testing: Motivation and challenges, in: International Conference on Software Engineering: New Ideas and Emerging Results Track, 2017, pp. 7–10.

[4] A. Nistor, T. Jiang, L. Tan, Discovering, reporting, and fixing performance bugs, in: Working Conference on Mining Software Repositories, 2013, pp. 237–246.

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

[6] M. Papadakis, M. Kintis, J. Zhang, Y. Le Traon, M. Harman, Mutation testing advances: An analysis and survey, Advances in Computers.

[7] R. Nilsson, J. Offutt, S. F. Andler, Mutation-based testing criteria for timeliness, in: Annual International Computer Software and Applications Conference, 2004, pp. 306–311 vol.1.

[8] W. B. Langdon, M. Harman, Optimizing existing software with genetic programming, IEEE Transactions on Evolutionary Computation 19 (1) (2015) 118–135.

[9] G. Xu, M. Arnold, N. Mitchell, A. Rountev, G. Sevitsky, Go with the flow: Profiling copies to find runtime bloat, in: ACM SIGPLAN Conference on Programming Language Design and Implementation, 2009, pp. 419–430.

[10] P. Delgado-Pérez, I. Medina-Bulo, J. J. Domínguez-Jiménez, A. García-Domínguez, F. Palomo-Lozano, Class mutation operators for C++ object-oriented systems, Annals of telecommunications 70 (3-4) (2015) 137–148.