# Search-based mutation testing to improve performance tests

Ana B. Sánchez
Universidad de Sevilla, Spain
anabsanchez@us.es

Pedro Delgado-Pérez
Universidad de Cádiz, Spain
pedro.delgado@uca.es

Inmaculada Medina-Bulo
Universidad de Cádiz, Spain
inmaculada.medina@uca.es

Sergio Segura
Universidad de Sevilla, Spain
sergiosegura@us.es

## ABSTRACT

Performance bugs are common and can cause a significant deterioration in the behaviour of a program, leading to costly issues. To detect them and reduce their impact, performance tests are typically applied. However, there is a lack of mechanisms to evaluate the quality of performance tests, causing many of these bugs remain unrevealed. Mutation testing, a fault-based technique to assess and improve test suites, has been successfully studied with func-tional tests. In this paper, we propose the use of mutation testing together with a search-based strategy (evolutionary algorithm) to find mutants that simulate performance issues. This novel approach contributes to enhance the confidence on performance tests while reducing the cost of mutation testing.

## KEYWORDS:

Search-based software engineering, evolutionary algorithm, mutation testing, performance testing, performance bugs.

## 1 INTRODUCTION

Slow and inefficient software caused by performance bugs can easily frustrate users and lead to significant loss of money. We refer to performance bugs as software defects that can produce significant performance degradation while preserving software functionality [4]. As an example of a real-world performance bug, consider the row one in Table 1. This bug caused more than 40 times slowdown in MySQL database servers [4]: developers implemented the method *fastmutex_lock* for fast locking, but this operation turned out to be much slower than normal locks.

| **MySQL Bug 38941 & Patch** | **Bug's description** |
|---|---|
| int fastmutex_lock(fmutex_t *mp){<br>...<br>- maxdelay += (double) random();<br>+ maxdelay += (double) park_rng();<br>... } | random() is a serialized global-mutex-protected glibc function. Using it inside *'fastmutex'* causes **40X slowdown** in users' experience. |
| **MNC mutant program** | **MNC's description** |
| int fastmutex_lock(fmutex_t *mp){<br>...<br>maxdelay += (double) **random();**<br>... } | MNC changes a method call with another compatible method call.<br><br>E.g.: **park_rng() ⟹ random()** |

**Table 1: A MySQL performance bug [4] and the mutation operator MNC [1]**

**Problem and motivation.** Performance bugs are different from functional bugs and require special testing care. They need much more time and effort to be detected and fixed than functional bugs [7], whose anomalous behaviours are clearly defined (e.g., a function returning an unexpected value). Performance test cases or profilers are typically used to detect performance bugs by running the program under test with specific inputs and checking whether the observed performance (e.g., execution time) is within the expected boundaries. However, the selection of suitable inputs and the assessment of the observed performance are challenging [5, 6]. Also, there is a lack of mechanisms to evaluate and improve the quality of performance tests, such as the generation of realistic performance bugs that simulate the degraded behaviour of a program [5].

**Proposal.** Mutation testing is a fault-based technique that could contribute to help us reproduce such performance issues. In this technique, some predefined faults, generated with different *mutation operators*, are purposely injected into the code with the aim of challenging the test suite to detect those faults. This happens when the output of the original program and the faulty version (called *mutant*) differs. In this case, we say that the mutant is *killed*. In this paper, we propose taking advantage of mutation testing to generate mutants with the same functional behaviour as the original program (i.e., mutants functionally-equivalent to the original version) but affecting different non-functional properties of the program. In this way, we have the opportunity to generate realistic coding errors, supported by the hypotheses behind mutation testing [3] (competent programmer and couple effect), instead of inserting trivial mutations into the code to simulate performance issues. For instance, the mutation operator *MNC* [1], shown in row 2 of Table

1, could help us recreate the kind of performance error illustrated in row 1 of Table 1.

However, mutation testing is known to be costly due to the high number of mutants that can be potentially produced from a piece of code, which undermines its utility in practice. As a solution, we propose the use of a search-based strategy, and more specifically, an evolutionary algorithm, to guide the search towards interesting mutants. As such, it is not necessary to generate and execute the whole set of mutants but only those selected by the algorithm. Namely, the evolutionary algorithm should search functionally-equivalent mutants that maximize the non-functional changes that the mutation causes with respect to certain performance properties. This algorithm is described in further detail in the next section.

## 2 ALGORITHM

Evolutionary Algorithms (EAs) process a set of candidate solutions in parallel that are combined and modified iteratively to obtain better solutions. Candidate solutions to the optimization problem play the role of individuals or chromosomes in a population, and objectives determine the fitness functions that measure the quality of the solutions. EAs are useful to solve the problem of mutant selection for the assessment of performance tests because not all mutants can serve this purpose but only a subset of them. Thus, we can use the evolutionary strategy to breed new mutants (individuals) derived from other valuable mutants, therefore sharing part of their information and increasing the probability to find interesting similar mutants. The properties that uniquely identify a mutation are the *mutation operator* that generates them (represented by a predefined code) and the *area of the code* in which they are injected (encoded by an integer in order of appearance).

The technique *Evolutionary Mutation Testing* (EMT) [2] makes use of an EA to generate a reduced subset of interesting mutants for the improvement of functional test suites. In broad terms, EMT searches for mutants that are not killed by the current test suite, which may induce the design of new test cases once they are reviewed. In this paper, we propose the use of an EA for mutant selection but, unlike EMT, we should guide the search in order to find useful mutants for the evaluation of performance tests. As a result, a new fitness function appropriate to this goal is required. In particular, our EA should look for functionally-equivalent mutants that maximize certain non-functional properties affecting the performance of the code, such as, the execution time or the memory usage. To this purpose, we initially propose two different objectives to be optimized simultaneously:

(1) **Search for functionally-equivalent mutants**: A mutant is considered functionally-equivalent when it is killed by no test cases. Therefore, to find this kind of mutants, we should *minimize* the number of test cases that kill the mutant. Let $f$ be a matrix of size $|M| \times |T|$ (number of mutants $\times$ number of test cases) with the functional results of executing each test case against each mutant, where $f_{mt}$ is 1 when the mutant $m$ is killed by the test case $t$, and 0 otherwise. The fitness for mutant $m$ with test suite $T$ would be:

$$\mathrm{F}_1(m, T) = \sum_{t=1}^{|T|} f_{mt} \tag{1}$$

This objective seeks mutants detected by few test cases to guide us on the generation of equivalent mutants.

(2) **Search for mutants that decrease the performance**: This means we should *maximize* the mutant execution result that shows the highest degradation with respect to the original program execution. Let $nfo$ be a vector of size $|T|$ with the non-functional results of executing each test case against the original program, e.g., test cases execution time. Let $nf$ be a matrix of size $|M| \times |T|$ with the non-functional results of running each test case on each mutant. The fitness for mutant $m$ and original program $o$ with test suite $T$ would be:

$$\mathrm{F}_2(m, o, T) = max(nf_{mt} - nfo_t), \text{ for } t = 1...|T| \tag{2}$$

This objective searches for mutants that degrade significantly the performance to help us derive new mutants that also affect the performance. Note that different non-functional measures can be considered in this fitness function. Also note that we do not sum the result of all test cases but we take the maximum of all of them to avoid the effect that similar test cases could have on the fitness.

By optimizing both objectives, it is expected that more functionally equivalent mutants with great impact on the performance are discovered instead of doing this selection merely in a random way. Hence, this can be solved applying a multi-objective search-based optimization or defining a mono-objective problem in which each objective is assigned a weight to determine the fitness function.

## 3 CONCLUSIONS

Equivalent mutants are not useful to evaluate functional tests and they increase the cost of mutation testing. However, testers have overlooked the opportunity that those mutants may offer to assess and enhance performance tests. This paper proposes using an evolutionary algorithm to drive the search towards interesting mutants without incurring a high cost. These mutants may be key to improve the ability of tests to detect performance issues.

## ACKNOWLEDGEMENTS

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.

## REFERENCES

[1] P. Chevalley. 2001. Applying mutation analysis for object-oriented programs using a reflective approach. In *Proceedings Eighth Asia-Pacific Software Engineering Conference*. 267–270.
[2] P. Delgado-Pérez, I. Medina-Bulo, and M. Núñez. 2017. Using Evolutionary Mutation Testing to improve the quality of test suites. In *Proceedings of the IEEE Congress on Evolutionary Computation, CEC'17*. 596–603.
[3] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer* 11, 4 (April 1978), 34–41.
[4] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. 2012. Understanding and Detecting Real-world Performance Bugs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 77–88.
[5] I. Molyneaux. 2009. *The Art of Application Performance Testing: Help for Programmers and Quality Assurance*. O'Reilly Media.
[6] A. Nistor, T. Jiang, and L. Tan. 2013. Discovering, reporting, and fixing performance bugs. In *Working Conference on Mining Software Repositories*. 237–246.
[7] S. Segura, J. Troya, A. Durán, and A. Ruiz-Cortés. 2017. Performance Metamorphic Testing: Motivation and Challenges. In *International Conference on Software Engineering: New Ideas and Emerging Results Track*. 7–10.