SlowCoach: Mutating Code to Simulate Performance Bugs

Yiqun Chen*, Oliver Schwahn†, Roberto Natella‡, Matthew Bradbury* and Neeraj Suri*
*School of Computing and Communications, Lancaster University, United Kingdom
†TU Darmstadt, Germany
‡University of Naples Federico II, Italy
y.chen101@lancs.ac.uk, os@cs.tu-darmstadt.de, roberto.natella@unina.it, m.s.bradbury@lancs.ac.uk, neeraj.suri@lancs.ac.uk

***Index Terms*—performance bugs, mutation testing, fault injection**

***Abstract*—Performance bugs are unnecessarily inefficient code chunks in software codebases that cause prolonged execution times and degraded computational resource utilization. For performance bug diagnostics, tools that aid in the identification of said bugs, such as benchmarks and profilers, are commonly employed. However, due to factors such as insufficient workloads or ineffective benchmarks, software defects related to code inefficiencies are inherently difficult to diagnose. Hence, the capabilities of performance bug diagnostic tools are limited and performance bug instances may be missed. Traditional mutation testing (MT) is a technique for quantifying a test suite's ability to find functional bugs by mutating the code of the test subject. Similarly, we adopt performance mutation testing (PMT) to evaluate performance bug diagnostic tools and identify where improvements need to be made to a performance testing methodology. We carefully investigate the different performance bug fault models and how synthesized performance bugs based on these models can evaluate benchmarks and workload selection to help improve performance diagnostics. In this paper, we present the design of our PMT framework, SLOWCOACH, and evaluate it with over 1600 mutants from 4 real-world software projects.**

I. INTRODUCTION

A program's performance is a software attribute that describes how quickly it can complete tasks or process inputs [1]. Performance is important when input sizes increase but program throughput does not scale accordingly. One of the causes of scalability issues is the presence of unnecessarily inefficient code within a program's codebase, which wastes computational resources when executed. For example, some function may not save the frequently needed result of an expensive computation, but instead recomputes it each time when needed, thereby wasting CPU cycles. Or, in a lock contention scenario, inefficient synchronization code may causes a program to wait unnecessarily for off-CPU events. Inefficient code chunks that could be optimized to increase program performance are often referred to as *performance bugs* [2]–[4].

An essential part of code optimization is to identify the inefficient code. The identification is usually carried out in two steps: 1) *detection*, i.e., determining whether performance issues exist in the first place, and 2) *localization*, i.e., pinpointing the code chunks causing the issues. Several approaches and diagnostic tools exist that assist in the detection or localization of performance bugs to guide performance diagnostics and optimization [3], [5]–[8]. Some syntax checkers [1], for example, find simple performance anti-patterns in code statically and thus help to avoid them. However, most performance bugs are too complicated for simple syntax rules to detect and must be analyzed with runtime information [3], [9], [10]. Given suitable workloads, benchmarks provide performance metrics that can be used as a comparison basis, while profilers provide runtime information that helps developers localize performance bugs. Still, neither benchmarks nor profilers can ascertain whether there are performance bugs without a proper specification or performance measurements from previous versions for comparison [11].

More sophisticated approaches [5], [12] aim to detect performance bugs by symptomatic analysis, e.g., tracing hardware/software events or memory accesses. Such approaches, however, lack a ground truth to be evaluated against. The symptoms on which they depend are not guaranteed to have observable performance degradation. This observability problems can be caused by many factors, such as insufficient workloads. For example, a vector in C++ reallocates memory when new elements are added and no memory is available to hold them. Such reallocation has a very small performance overhead, making it challenging to measure with profilers. But reallocation would cause significant performance degradation if it occurs often [9]. Thus, it is not always clear that the symptoms identified by such approaches are actually performance bugs, as the given workload may not be able to exercise the problematic code frequently enough.

Moreover, there is no simple way to evaluate performance diagnostic approaches beyond the small set of a priori known and reproducible performance bugs [10], [12], [13]. The lack of a standardized *corpus* of performance bugs and the lack of rules for synthetically creating performance bug instances to evaluate performance bug detection and localization approaches, motivate our interests in the synthesis of performance bugs.

Inspired by the idea of software fault injection, we adopt techniques from *mutation testing* (MT) [14], [15] to inject performance bugs to evaluate the quality of existing performance bug detection and localization approaches. MT intentionally injects synthetic faults, using code mutation, into the test subject's code to check if its test suite can find them. The ultimate goal is to quantify and improve the quality of test suites. The rules controlling how and where the source code is

mutated are known as *fault models*. After mutation, the source code is expected to produce functional deviations compared to the original code. A high-quality test suite is supposed to capture these deviations. In this paper, we employ *performance mutation testing* (PMT) to synthetically create performance bugs as an assessment for the performance testing. In stark contrast to MT, PMT requires that code mutations *do not* introduce functional deviations since performance bugs should be considered separately from functional bugs. Therefore, PMT requires the preservation of functional equivalence (FE). This makes PMT different from traditional MT techniques, which have the exact opposite requirement.

In this paper, we introduce SLOWCOACH, our novel PMT framework. We demonstrate its practical utility and show how PMT can help to improve performance bug diagnosis approaches. Particularly, we address the following research questions in this paper:

1) How does PMT relate to MT?
2) What are the different dimensions to consider for PMT fault models?
3) Can PMT produce enough useful mutants in practice?
4) How useful are synthetic performance bugs in practice?

II. RELATED WORK

The term *performance bug* was coined by Jin *et al.* [1], who investigated more than 100 performance bugs in real-world C/C++ projects and developed a tool for detecting these bugs. Chen *et al.* [16] surveyed and semantically categorized more than 700 performance bugs from real-world developer commits from 13 popular C/C++ projects. Sánchez *et al.* [17] investigated the performance bugs across multiple publications in the research community and Tizpaz-Niari *et al.* [18] surveyed performance bugs in machine learning libraries.

Many approaches based on the symptoms of performance bugs have been proposed. Su *et al.* [5], Chabbi *et al.* [12], and Wen *et al.* [13] detect performance bugs by processor event based sampling (PEBS), which samples hardware events, such as memory or cache accesses, on modern processors. This facility can be used to identify 1) *dead store*, where data are stored to memory but never loaded later, 2) *redundant load*, where data are loaded but never stored back to memory, and 3) *false sharing*, where memory accesses from different threads are close together, resulting in cache thrashing. To detect these bugs with PEBS, performance tests are needed so that dynamic memory accesses can be analyzed. Dynamic memory access patterns are also helpful to detect and localize redundant computation in loops [3]. Moreover, performance bottlenecks caused by off-CPU events [19] and lock contention [6], [10], [20] can be detected using dynamic runtime information. Attariyan *et al.* [4] propose a more generic state-of-the-art performance profiler to localize performance bugs. State-of-the-practice performance diagnostic tools, e.g., `perf` [7] and `lttng` [8], are capable of profiling both on-cpu and off-cpu events, as well as numerous additional kernel events.

Besides various performance bug diagnostic approaches in different domains, many researchers also consider *computational redundancy* as a significant indicator for performance bugs. Wen *et al.* [21] define the same return value being computed by repeatedly calling a function as a source of performance bugs. Song and Lu [3] propose a more generic approach to detect whether the results of each iteration are redundant in a loop. Besides logical approaches to detect computational redundancies, Della Toffola *et al.* [22] aim to find locations where the computational results can be cached by deep learning. The fixing strategies often applied to mitigate redundancy are to either skip the computation, e.g., by introducing a fast path, or cache the results of computation for future usage. Interestingly, these two strategies are among the dominating performance bug fixing patterns in real-world scenarios [16]. Despite many variants of PMT fault models, we carefully adopt the computational redundancy anti-patterns from the aforementioned works in this paper since computational redundancies are a stronger indicator for performance bugs in comparison to other metrics.

MT techniques are also well studied across the research communities. Papadakis *et al.* [14] as well as Jia and Harman [15] review the development of MT techniques covering various programming languages from 1970 to 2017. Chekam *et al.* [23] implement and evaluate an early mutation testing framework. Chekam *et al.* [24] develop a symbolic execution approach to search for the input to kill *hard-to-kill* mutants (please refer to Table I for definitions). Devroey *et al.* [25] propose an approach to identify equivalent mutants by NFA simulation. Recent research [26] also confirmed the efficacy of mutation testing in industry practices.

Natella *et al.* [27] provide an overview of mutation testing techniques in the context of assessing systems against software failures. Research in this area has been focused on the representativeness of injected faults, which is a indicator of whether artificial bugs are hard-to-kill, and thus subtle enough to cause realistic software failures [28]. However, simulating failures through code mutation can be cumbersome, as the mutated program may need to be recompiled, and the mutants have often no effect on the program (i.e., equivalent and hard-to-kill mutants, see Table I). Thus, previous studies have investigated how to efficiently perform mutations on binary code [29], and whether faults injected inside a software component (i.e., through code mutation) can be replaced by more convenient injections at software interface level (i.e., by corrupting data returned from a component) [30]. Other work has also developed a PMT framework with many MT operators [31]. As will be demonstrated in Sections IV and V, SLOWCOACH shows better results in terms of evaluating a performance testing environment by careful implementation of mutation operators and better interpretation of mutation scores. In this paper, we also investigate efficiency and representativeness issues in the context of performance bug injection, by exploring different approaches from the design space of PMT techniques.

III. Background

A. Performance Mutation Testing

Mutation testing (MT) is a technique to evaluate test suite quality [14], [15]. In MT, faults are injected through code mutations, commonly at the source code level. The code transformation rules that govern how to mutate the original source code are called *mutation operators*. The resulting copies of mutated code are termed *mutants*. A *fault model* describes what, where, when, and how to inject the buggy code. An example MT fault model could match all binary *and* operators (`&&`) in `if` statements and change them to binary *or* operators (`||`). If a software's test suite is not able to distinguish the generated mutants from the original software, then the test suite needs improvement as it fails to detect the injected faults. We compare MT and PMT in Table I and provide further definitions of MT concepts.

In the context of performance evaluation, a test suite is typically represented by benchmarks and workloads, which exercise the software under test with input data to reveal performance regressions and to diagnose bottlenecks. Similar to MT evaluating test suites, we adopt PMT as technique to evaluate performance benchmarks and performance diagnostics approaches. However, MT fault models aim to change the functional behavior of the original code, since these are the faults that test suites should detect. In practice, mutants that *do not* change functional behavior (equivalent mutants) are avoided or filtered out if possible. In contrast, PMT filters out mutants that *do* change functional behavior as it is only meaningful to compare the performance of functionally identical programs. Thus, an important requirement for PMT fault models is to retain the functional behavior for all mutants while introducing performance overheads. These concepts are detailed in Table I. As such, P-mutants are different from traditional mutants in terms of key MT concepts. To better distinguish these mutant types, we refer to such performance mutants as *P-mutants*.

The functional equivalence (FE) of P-mutants is defined as: *given a set of inputs, all P-mutants should produce the same set of outputs as the unmutated code*, i.e., all P-mutants should adhere to the same functional specification as the original code. Despite much proposed work addressing the program equivalence problem in the research community (e.g., [32], [33]), the preservation of FE in PMT cannot be trivially solved. The proposed formal equivalence checkers verify if two programs execute the same steps, while performance optimization in general involves two versions of a program that produce the same output by executing correspondingly fewer steps. Conversely, PMT fault models would lead to more steps being executed in a program. So, formal FE checkers cannot be used to effectively verify FE for PMT. An alternative approach proposed by Devroey *et al.* [25] detects equivalent P-mutants by the simulation of non-deterministic automata. However, none of these approaches can generally solve the FE problem in acceptable time for potentially thousands of mutants. As a practical compromise, we carefully select PMT mutation operators that are unlikely to affect functional behavior and check FE using classical functional tests.

B. PMT Fault Models

Performance bugs are often believed to be fixed by "relatively simple source code changes" [1]. However, they usually involve more complicated semantic changes in the real world [16], [17].

Jin *et al.* [1] identified performance bugs via detectors which relied on the *contextual information* of the code. For example, given function A invoked before function B would cause performance degradation, a detector finds all invocation pairs of function A before function B. The contextual information in this example is the relative invocation ordering of functions A and B. Some PMT fault models inject performance antipatterns derived from the detection strategies, which also requires domain knowledge. As an example, the code shown in Listing 1 demonstrates a performance optimization scenario in the real world [16]. The code snippet matches a string against a pattern[1], this algorithm uses Deterministic Finite Automata (DFA) or keywords searching to perform the matching. DFA usually matches patterns with wildcards faster than the keywords searching algorithm, if the inputs are unibyte and do not contain any back references. Hence in Listing 1 we search by the DFA if the condition `dfafast` is satisfied (line 9 and 19). This example will be the first case study discussed in Section V-D1.

The fault models derived from this example could be either to remove the `else if` block or to change the value of `dfafast`. Unfortunately, neither model can be described by simple syntactic rules without contextual knowledge about what the affected code blocks or variables are used for. Such fault models suffer from several drawbacks. Firstly, large human-in-the-loop efforts are required to understand the entire software project and to implement these mutation operators. Secondly, these fault models generate only a limited number of P-mutants. In our experiments on `grep` (discussed in Section V-D1), each mutation operator instance[2] generates about 1 to 2 P-mutants. PMT fault models that do not rely on domain knowledge are more generic than their counterparts that use contextual information. We classify the space of possible PMT fault models along two dimensions: how representative and how context dependent fault models are. Representativeness can be categorized as the fault models simulating performance bug effects and developers' errors. The other dimension specifies if a fault model is context dependent or independent. This is visualized in Fig. 1. Since the Listing 1 simulates developer errors, the described fault models fall into quadrant 2, or $\mathbb{Q}2$ in short, as they depend on contextual information. The context-independent fault models fall into $\mathbb{Q}4$ as they do not rely on contextual information when injecting faults.

An alternative to the simulation of developer errors is to simulate the effects of performance bugs. All performance bugs have a performance impact either on-CPU or off-CPU. For example, an on-CPU performance issue can be caused by

[1]The code is simplified for the discussion.

[2]SlowCoach embeds the contextual information into mutation operators. Each operator with the contextual information is an instance. (c.f. Section IV-A)

TABLE I
PERFORMANCE MUTATION TESTING VS. MUTATION TESTING [27]

Concept	Performance Mutation Testing	Mutation Testing
Test Suite	A fixed set of benchmarks and workloads yielding various performance metrics (e.g., execution time, memory usage, and execution paths) to be compared against. The workload carried out by the benchmark aims to identify those tests that have worse performance metrics than the unmutated baseline.	A test suite (test programs and inputs) to determine whether a program complies with its (functional) specification. Tests should *kill* (i.e., detect) mutants to demonstrate their fault detection capability.
Equivalent Mutants	All performance mutants must be functionally equivalent to the original version. Performance equivalent mutants are those whose performance results are statistically close to the original.	If a mutant is functionally equivalent to the original software, this mutant will never be killed by the test suite.
Hard-to-kill Mutants	Functionally equivalent mutants that can be killed only if mutated code is executed sufficiently frequently. We hypothesize that all code changes introduce performance overheads if not optimized out, while the overheads need repetitive execution to be observable.	Some mutants can only be killed by few, very specific test cases. These mutants help identify possible improvements of test suites.
Mutation Score	Identical to the mutation score by mutation testing. But since there are multiple performance metrics, there are correspondingly multiple mutation scores for the different perspectives of the metrics.	The mutation score grades the quality of a test suite. It is the percentage of nonequivalent mutants that can be killed by the test suite.

```
1 +bool dfaisfast (struct dfa *d) {
2 +   return !d->multibyte &&
3 +       d->has_no_backref();
4 +}
5
6  size_t EGexecute (char const *buf,
7         size_t size, size_t *match_size,
8         char const *start_ptr) {
9 +  bool dfafast = dfaisfast (dfa);
10   /* ... */
11   for (beg = end = buf; end < buflim; beg = end){
12       if (!start_ptr) {
13           if (kwset) { /* Slow path */
14                 do_kwset_search();
15                 /* ... */
16                 if (matched) return;
17             }
18 -         else
19 +         if (!kwset || dfafast) {
20               /* Fast path */
21               do_dfa();
```

Listing 1. Fast Path

inefficient algorithms that waste CPU cycles, and an off-CPU issue can be related to unnecessary file IO operations that cause wait times. The on-CPU overheads can be simulated by inserting useless operations into the code, while off-CPU overheads can be emulated by inserting useless `sleep()` operations. Although these simulations do not hamper the FE and potentially generate more P-mutants, the introduced code mutations do not resemble performance bugs found in real world software. The resemblance of synthesized bugs to those that occur in the real world is called the *representativeness* of software faults [28]. The simulation of performance bug impacts is less representative when compared to the simulation of developer errors as indicated on the y-axis in Fig. 1. The representativeness of MT fault models is a significant indicator for the efficacy of MT. Performance bug detection and localization approaches, however, are only concerned with the symptoms of performance bugs. In other words, a performance bug may itself be trivial, but it is not trivial to evaluate whether a particular benchmark or workload is capable of showing observable performance degradation.

Like other PMT fault models, the effect simulation may also be dependent on domain knowledge. For example, as there are no consistent interfaces among C/C++ software projects for low-level system operations, the function names of these operations are required for fault models. In the case of heap memory, allocations are performed with `malloc()` in standard C and with `new/new[]` in C++, but many projects adopt custom allocators, e.g., `kmalloc()` in the Linux kernel, and `ALLOC()` or `xmalloc()` in gnulib. We label the simulation of performance bug effects as $\mathbb{Q}1$ and $\mathbb{Q}3$ in Fig. 1 with and without context dependency correspondingly. In spite of the context dependency, the difference between $\mathbb{Q}1$ and $\mathbb{Q}3$ is often negligible in terms of simulating performance bug causalities, which is why we usually discuss $\mathbb{Q}1$ and $\mathbb{Q}3$ fault models together. As SLOWCOACH allows developers to encode contextual information in the fault models, this means there are limited differences between $\mathbb{Q}1$ and $\mathbb{Q}3$. Therefore, we use $\mathbb{Q}3$ to represent $\mathbb{Q}1/3$ mutants in the following discussions.

IV. SLOWCOACH: A PMT FRAMEWORK

A. Overview and Workflow

A general overview of SLOWCOACH's workflow is provided in Fig. 2. SLOWCOACH's primary inputs are the source code of the target software project and project specific configuration which must be provided by the user. The configuration allows customizing the PMT process for a specific project, such as function ignore or include lists, required for some mutations operators to work correctly. Listing 2 shows a configuration example to replace all local variables in the `main` function named `dfafast` (in Listing 1) with a value of `false`. One or more optional `caller` elements can be provided to limit

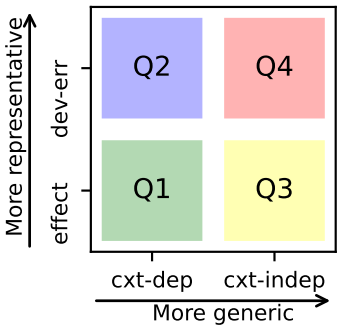

Fig. 1. PMT Fault Models

the scope of replace operations to a specific set of functions. The tool mutates the original source code to generate different mutants (as modified source code files), which are then applied to unique copies of the project. Both the original project and the mutated versions are then compiled to generate the executable programs. The programs are then executed with benchmarks using various workloads, or other performance diagnostic approaches are applied to the executables. The performance metrics the user is interested in are measured and recorded for the original and all mutated versions. Based on these metrics or their comparison across versions, the user can assess the quality of the used benchmarks, workloads, or performance diagnostic tools.

B. Mutation Operators

In this section, we take the operators from Table III in Section III to illustrate how PMT mutates the code.

1) $\mathbb{Q}4$ Operators (developer errors, context-independent):

a) $\mathbb{Q}4$-A – Loop Unbreaker: As discussed in Section III, our fault models simulate developer errors without contextual information. For SLOWCOACH, we select 2 representative mutation operators to demonstrate the concepts of $\mathbb{Q}4$ fault models and discuss two of them in this section. The first operator is called *loop unbreaker* and derives from a common performance optimization pattern found in many real-world projects [3], [16] that we call *loop breaker*. This pattern is similar to the fast path pattern in Listing 1. We consider that early termination (`break`) of a loop is a fast path, stopping the loop early when possible. Listing 3 shows an example for the loop breaker pattern. The `if` statement containing the `break` in Line 2 is the fast path that terminates the loop early. Our loop *un*breaker mutation operator removes these `if` statements.

The main drawback of loop unbreaker is that semantics cannot be asserted from the syntactic `if` statement. It is possible that the `if` break is necessary for functional behavior, e.g., a loop returning the first occurrence of an item in a list. Moreover, loops may rely on such an `if` break to terminate, so the removal of it may cause the program to hang. Although loop hangs can be detected by runtime monitoring [34], [35], it is hard or even impossible to predict statically during code mutation. To minimize the probability that the loop unbreaker operator causes program hangs, we adopt a naive heuristic to exclude loops without a terminating condition, e.g., `for(;;)` or `while(1)`, because these loops rely on a `break` statement to terminate.

b) $\mathbb{Q}4$-B – Oblivion: Our second mutation operator is called *oblivion*. It is derived from another performance optimization pattern called *cache memoization* [22].In this pattern the code records the result of some heavy computation and reuses it later without re-doing the computation again. A simple example is shown in Listing 4 at Lines 1 to 2, where the results of `foo()` are cached in variable `a` and re-used in `bar(a)`. To simulate performance bugs where cache memoization was omitted, our oblivion operator substitutes all variable references with their initializers, so that the result of the initializing function is redundantly computed. In Listing 4, the oblivion operator matches all occurrences of variable `a` and substitutes them with a function call to its initializing function `foo()` as shown on Line 3.

The oblivion operator, however, may alter the functional behavior if the relevant function updates or depends on the global state of the program. A notorious example is memory allocation, whose results depend on the global state and repeated calls to memory allocation functions lead to different results. To mitigate this, we adopt a straightforward function blacklisting approach, i.e., the code is mutated only if a local variable is declared with an initializer and the initializing function is not blacklisted. As a result, given a presumably side-effect free initializer, the oblivion operator adds a ternary operator as shown in Line 4 in Listing 4, where the local variable `a` is replaced with an expression of the form: `(a == foo()) ? foo() : a`. If the initializer function (`foo()`) returns a different value, variable `a` was changed since its initialization and we fall back to using variable `a` directly to not affect the original program semantics. While this approach amplifies the performance impact by calling `foo()` twice in the ternary operator, it increases the probability that the original functional behavior is retained if `foo()` is side-effect free. Despite these efforts, it is still possible that the functional behavior is changed, for example, if the initializer function is not side-effect free or takes arguments that change over time.

2) $\mathbb{Q}2$ Operators (developer errors, context-dependent): The performance bug dataset provided by Chen *et al.* [16] shows that almost all real-world performance bugs and optimizations depend on project specific context, which has been confirmed by other works on real-world performance bugs [3], [17].Therefore, additional contextual information is usually needed to synthesize representative performance bugs since semantic fault injection is not possible with purely syntactic rules in most MT approaches [14], [15], [23]. SLOWCOACH encodes the contextual information in the configuration as described in Section IV-A and shown in Fig. 2. Considering the fast path example in Listing 1, occurrences of the `dfafast` variable can be replaced with a `false` so that the fast path will never be taken. A mutation operator that removes fast paths can then match the variable reference with this name, its parent `if` statement, and its enclosing function `foo()`, and safely remove this particular fast path as defined by the user.

To demonstrate how context dependency is encoded into

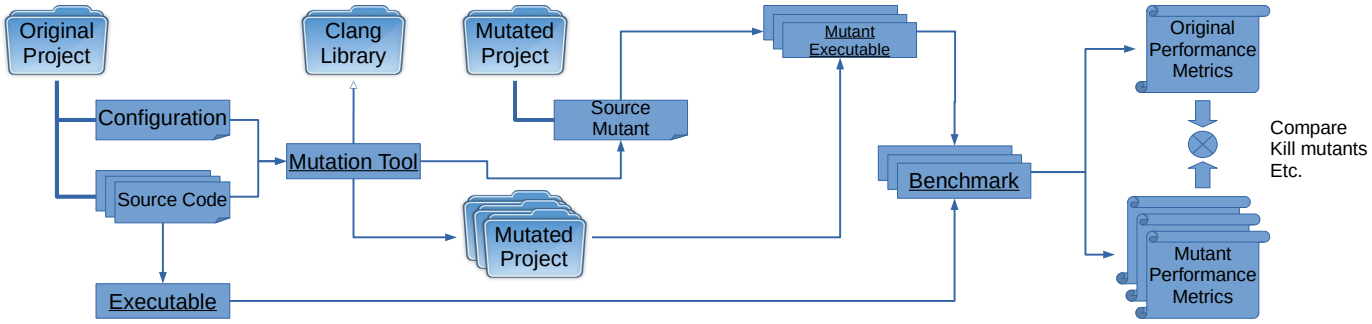

Fig. 2. SLOWCOACH Workflow

```
<local-var>
  <var-name>dfafast</var-name>
  <value>false</value>
  <caller>main</caller>
</local-var>
```

Listing 2. Configuration Example

```
for (int i = 0; i < 1024; i++) {
    if (some_cond(i)) break;
    do_something();
}
```

Listing 3. Loop Breaker optimization pattern (ℚ4-A in Table III)

```
int a = foo();
bar(a);
bar(foo());
bar((a == foo()) ? foo() : a);
```

Listing 4. Cache Memoization optimization pattern (ℚ4-B in Table III)

```
volatile int sum = 0, foo[ARR_LEN];
for(int i = 0; i < foo_len; i++) {
    sum += foo[i];
}
```

Listing 5. **1*** Loop (ℚ3). Produces useless results in each iteration.

PMT operators, we take two examples of code optimizations from the GNU `grep` project (one of the most popular text search utilities) and show how they can be reversed to produce performance bugs. Both examples are typical fast path performance bug fixes. In the first case[3], developers introduce a new function `dfaisfast()` (described in Listing 1) as the switch between the fast path and the slow path. In the second case[4], `fgrep_icase_available()` is added to the code, which functions like `some_cond` as well. SLOWCOACH provides a mutation operator that replaces a call to a given function with some other function, a variable, or a concrete value. This operator can replace the call to `dfaisfast()` with a fixed value of `0` to change the fast path condition, thereby permanently preventing the fast path from being executed. For both cases, ℚ2-A and ℚ2-B operators Table III are instantiated to replace occurrences of all calls to `dfaisfast()` and `fgrep_icase_available()` with a value of `0`.

3) ℚ1 & ℚ3 Operators (performance effects, context-dependent and -independent): As discussed earlier, mutating for performance while retaining the functional behavior of programs is generally a hard problem. ℚ4 operators require expensive static or dynamic analysis if unchanged functional behavior must be guaranteed. ℚ2 operators compromise on generality for a more accurate reincarnation of previously known performance bugs. Since most real-world performance bug fixes involve specific inputs, the lack of generality limits the usefulness of ℚ2 operators for evaluating larger sets of workloads. An alternative to ℚ4 and ℚ2 operators is to simulate the observable effects of performance bugs rather than the bugs themselves. A naive mutation operator could insert `sleep()` operations into the code, to extend the wall clock time of the execution. But inserting `sleep()` does not aid in improving performance benchmark design workload selection, because sleeping does not affect the CPU time, hence can be easily detected by checking the CPU utilization.

Loops are often considered as one of the main sources of performance bottlenecks [2], [3], [37], [38], but reversing loop-related performance optimizations may introduce functional behavior deviation. To simulate the effects of loop-related performance bugs, we develop mutation operators to synthesize inefficient loops. We derive the ℚ3 fault models from the inefficient loops classified by Song and Lu [3]. There are many types of inefficient loops, e.g. **1***, **0*1?** or **0*1?**. Since we are simulating the effects of performance bugs by ℚ3 operators, we do not discuss all types of loops in detail. We pick a typical inefficient loop known as the **1*** loop for our PMT study and evaluation. **1*** loops produce results (side effects) in each iteration, where these results are useless. A simplified example is the loop in Listing 5 which computes the sum of an integer array. Since the variable `sum` is written by the incremented value of `foo[i]`, there is a result in every loop iteration. However, these results (accumulated as `sum`) are not used after the loop, i.e., they are unnecessarily computed. Although most **1*** loops are semantically related to the loop context and much more complicated, ℚ3 mutation operators

[3]GNU `grep` repository [36] Git commit ID 3255bc
[4]GNU `grep` repository [36] Git commit ID 960ad3

TABLE II
EVALUATION SOFTWARE PROJECTS

Project	Application Area	PL	LoC	Mutants	T_m*	OH†
`astar`	Path-finding Algorithms	C++	3959	514	0.77	36.41
`bzip2`	Compression	C	7292	892	0.40	17.58
`mcf`	Combinatorial Optim.	C	2044	239	0.40	19.36
`grep`‡	GNU Text Utility	C	357 520	1532		

* Time in seconds to generate all source code P-mutants.
† Total time in seconds to generate, sample, compile sampled P-mutants.
‡ `grep` has 1532 $\mathbb{Q}3$ and $\mathbb{Q}4$ operators and 2 extra case study P-mutants. Overheads not applicable.

TABLE III
MUTATION OPERATORS.

Operator	Description	MPO_{O0}*	MPO_{O3}
$\mathbb{Q}2$-A	Replace `dfaisfast()` calls with `0`		
$\mathbb{Q}2$-B	Replace `fgrep_icase_avail()` calls with `0`		
$\mathbb{Q}3$-A	Prepend `sleep(1)` statement to loop bodies	1000000	1000000
$\mathbb{Q}3$-B	Prepend **1*** loop (10 000 iterations) to loop bodies	51.47	4.82
$\mathbb{Q}3$-C	Prepend **1*** loop (100 000 iterations) to loop bodies	514.55	48.25
$\mathbb{Q}3$-D	Prepend **1*** loop (1 000 000 iterations) to loop bodies	5137.72	479.16
$\mathbb{Q}4$-A	Apply *loop unbreaker*, remove early `break` from loops		
$\mathbb{Q}4$-B	Apply *Oblivion*, remove cache memoization		

* Time measured by Google microbenchmark, in microseconds.

could use a simple form (e.g., summing integer) to simulate the performance bug effects. Other loops like **0*1?** and **[0|1]*** with different memory access patterns can be easily implemented and encoded in SLOWCOACH. Due to high similarity in terms of introducing performance impacts for PMT, we only evaluate the mutation operators derived from **1*** loops in this paper, and apply various array lengths (`ARR_LEN`) to simulate different performance impacts. The `foo` array is allocated on stack to avoid randomness caused by dynamic allocators (line 1 in Listing 5). Since static arrays will be optimized by compilers, `volatile` was used on `foo` to prevent the compiler removing injected loops.

V. EVALUATION

In this section we evaluate SLOWCOACH by applying it to 4 real-word software projects. Our evaluation is driven by the following research questions.

RQ 1 What is SLOWCOACH's runtime overhead and how many P-mutants does it generate?
RQ 2 Which fraction of the generated P-mutants preserve the functional equivalence to the original version?
RQ 3 Does PMT assist in identifying issues with performance testing tools and the testing environment?

A. Experimental Setup

1) Implementation & Mutation Operators: We use our SLOWCOACH prototype implementation to conduct the experiments for this evaluation. The prototype targets C and C++ software as performance critical software is often implemented with these languages. The prototype itself is realized using C++ and Python. For its code mutation functionality, it builds upon the Clang C/C++ frontend in version 10.0.1. The currently supported mutation operators (cf. Section IV-B) that we apply in our experiments are described in Table III. Note that the $\mathbb{Q}2$ operators are only used in our `grep` case study in Section V-D due to their program specificity and the manual effort involved.

2) Evaluation Targets: Since our prototype targets C and C++ software, we select evaluation targets implemented in these languages. Additionally, the selected targets should be sensitive to performance issues, i.e., good performance should matter to their users. For example, the performance of a compression program such as `bzip2` is important to its users as excessive processing time wastes resources. Targets should also be selected from diverse application domains to avoid domain specific bias. Considering these aspects, we choose the SPEC CPU 2006 benchmark suites [39] as our primary source of evaluation targets. Table II provides an overview of the selected target programs along with a size estimation as number of lines of code (LoC). The target programs for our case study are `astar`, `bzip2`, and `mcf` from SPECint 2006, as well as `grep`, which is a well known real-world utility.

3) Workloads: To exercise the evaluation targets, we use workloads of different sizes. For the programs from SPEC, we use the standard workloads (inputs) that come with SPEC. For `grep`, we use the developer test suite.

4) Experiment Execution: We generate and build the mutants and run all experiments inside Docker containers on 4 *identical* machines, which are equipped with Intel® Core™ i7-4790 CPUs (3.60 GHz), 12 GiB RAM, and 256 GiB SSDs. The host runs Ubuntu 21.10 with Linux 5.13.0. Inside the Docker containers, we run an Ubuntu 20.04.3 LTS user-space.

Our workflow consists of the following steps: (1) we generate and store all P-mutants for all evaluation targets as source code, (2) we compile both the original programs (for the baseline) and the P-mutants using Clang with `O0` and `O3` optimization and store the resulting binaries, (3) we repeatedly execute both the original (baseline) and mutated binaries with their workloads and collect time measurements using GNU `time`. As shown in Table II and discussed in Section V-B, SLOWCOACH generates hundreds of P-mutants for our evaluation targets. To reduce these numbers to a manageable level for experiment execution, we randomly sample 50 $\mathbb{Q}4$ P-mutants[5], and 30 random locations on which four $\mathbb{Q}3$ operators ($\mathbb{Q}3$-A to $\mathbb{Q}3$-D in Table III) inject performance bugs. For each project in Table II, there are a total of 170 P-mutants being sampled. Since

[5]If $\mathbb{Q}4$ operators produce less than 50 P-mutants, all generated P-mutants will be used.

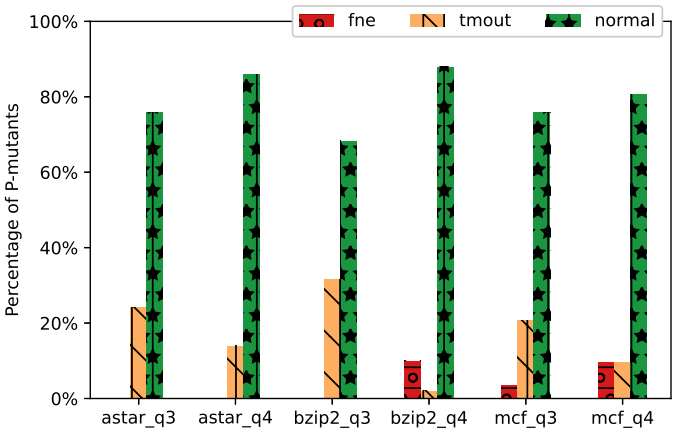

Fig. 3. Functional Equivalence by Operators and Programs

performance measurements are affected by external factors [39], we repeat each execution 30 times and report median values if not stated otherwise. The median values are used because they are robust against outliers. To mitigate potential experiment stalls, we assign a time budget of 30 minutes for each execution, which is twice as long as the longest baseline execution needs.

B. RQ 1: Mutant Generation and Overheads

We analyze the amount of P-mutants SLOWCOACH generates for our evaluation targets when we apply four different $\mathbb{Q}3$ (effect simulation) and $\mathbb{Q}4$ (dev errors, no context) mutation operators (cf. Table III). We omit $\mathbb{Q}2$ operators as they are reserved for our `grep` case study in Section V-D. For $\mathbb{Q}3$ operators, Table III provides the *minimal performance overhead* (MPO), which is the performance effect introduced in an individual execution of the mutated code, as measured with a microbenchmark on the mutated code chunks. The MPOs are given for both unoptimized (MPO_{00}) and optimized (MPO_{03}) compilation.

In total, SLOWCOACH produces 1645 $\mathbb{Q}3$ and $\mathbb{Q}4$ P-mutants for the three programs (`astar`, `bzip2` and `mcf`), as shown the *Mutants* column in Table II. The T_m column shows the time SLOWCOACH takes to produce all source code P-mutants. The *OH* column is the total time SLOWCOACH takes to generate all source code files for P-mutants, inject sampled P-mutant source code files into the original project copies (cf. Section V-A4) and build sampled P-mutants. The time to produce P-mutants is measured with the debug version of SLOWCOACH without compiler optimization. SLOWCOACH produces source code P-mutants in less than a second and builds all sampled P-mutants within one minute. This shows the scalability of SLOWCOACH, where it generates 1645 P-mutants in 1.57 seconds and builds 491 P-mutants in 73.35 seconds. In summary, SLOWCOACH produces large amounts of P-mutants, where the exact number and distribution depends on the target program's code structure and the presence of performance optimization patterns in the code.

C. RQ 2: Functional Equivalence

In this section, we analyze the proportion of P-mutants which preserve FE compared to the original program as only those preserving FE are suitable for PMT. Since FE is undecidable for arbitrary programs, we resort to comparing the *standard output* (`stdout`) and the *standard error* (`stderr`) streams of baseline (original program) and P-mutant executions. If the outputs from both `stdout` and `stderr` of a P-mutant are identical to those of the baseline, this P-mutant is considered functionally equivalent. Otherwise the P-mutant is not considered to preserve FE and will not be used further as part of the mutation score computation in Section V-D. Also, if a P-mutant terminates abnormally (exit with signals), this P-mutant is removed from further experiments. If a P-mutant does not finish within the assigned time budget, we consider it a timeout.

Fig. 3 summarizes the results of our FE analysis based on O0 binaries. Among all 483 sampled P-mutants, we observe a total of 12 functional deviations (fne), 101 timeouts (tmout), and 370 P-mutants without functional deviation (normal). All 12 functional deviations are captured by signal 11 (segmentation fault), while none of them have caused output deviation. 4 instances of the segmentation faults are caused by the $\mathbb{Q}3$-C and $\mathbb{Q}3$-D operators that perform 1×10^5 and 1×10^6 iterations. They are all located at `pbeampp` in `mcf`, where large numbers of stack allocations (1×10^5 and 1×10^6 integers) lead to memory errors. Another 3 FE cases are caused by $\mathbb{Q}4$-A operators, and $\mathbb{Q}4$-B triggers the last 5 cases.

Timeouts can be seen as the *fuzzy part* between functional and performance bugs [40]. Timeouts can be caused either by a program hang, e.g., infinite loops from removing `break` statements (cf. Section IV-B1), or by an excessively slow program. Without dedicated monitoring [34], [35], these cases are hard or impossible to distinguish.

In our experiments, about 20.9% of P-mutants yield timeouts, most notably by $\mathbb{Q}3$ operators. 94 out of 101 timeouts are fully optimized $\mathbb{Q}3$ P-mutants (`O3`), which do not timeout when unoptimized. From an example P-mutant in `bzip2`, we found that the unoptimized P-mutant finishes processing inputs in (144.65 ± 0.12) s, while the fully optimized P-mutant binary takes 40 hours to finish. Further investigation with Linux `perf` shows that 27.15% of the CPU time of the unoptimized (`O0`) P-mutant is spent on the function where the redundant loop is injected, which is already the second slowest function according to `perf` reports. The fully optimized P-mutant, on the other hand, has spent 83.78% CPU time on the same function in the first hour of execution. By analyzing the assembly code of the affected `O3` binary, we see that the injected redundant loop is causing 76.91% of all CPU time. This finding demonstrates the potential of PMT techniques to be extended to identify compiler optimization anomalies because we observed optimized binaries to run longer than unoptimized binaries. The mutation operators in SLOWCOACH could be applied as mutation approaches of compiler fuzzers [41], to facilitate the detection of compiler bugs or undefined behaviors.

We discard both P-mutants that functionally deviated or

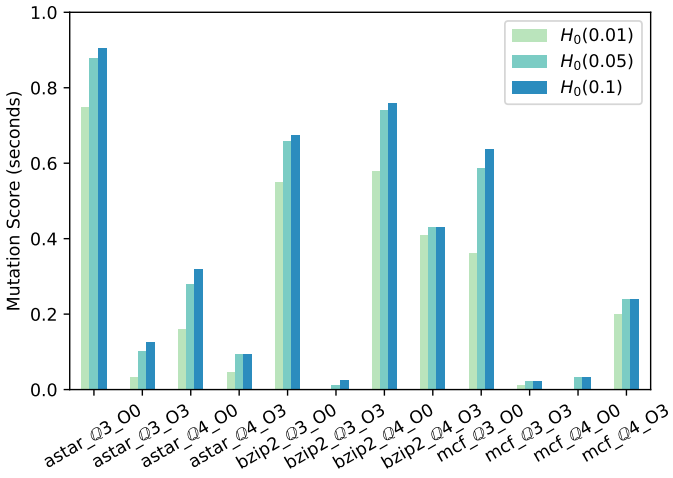

Fig. 4. Mutation Scores by H_0

timed out for mutation score analysis, but keep the `O0` P-mutants whose `O3` counterparts timed out due to the anomaly where unoptimized binaries finish execution earlier than optimized binaries. In summary, SLOWCOACH is capable of generating 76.6% valid mutants for mutation score analysis.

D. RQ 3: Mutation Score and Discussion

In this section, we evaluate SLOWCOACH by grading the performance testing setup (cf. Section V-A3) to help improve the quality of performance testing environments. The grade is defined to be the mutation score (as in classic MT) which is the number of P-mutants that can be killed. The wall clock time is used as metric to determine the performance of a mutant. As the wall clock time is susceptible to external noise, we repeat the execution 30 times and apply one-sided statistical testing to determine if a P-mutant can be killed.

There are many statistical approaches to compare the performance metric (wall clock time) results for a P-mutant P_m and for the baseline P_b, e.g., by arithmetic means or medians. But both means and medians are not robust enough to tolerate external noises. Delgado-Pérez *et al.* [31] used Mann-Whitney U tests to compare P_m and P_b, by rejecting the null hypothesis that *P_m and P_b are drawn from the same distribution*. More specifically, the null hypothesis states that the cumulative distribution function (CDF) $F(x)$ of P_m is the same as the CDF $G(x)$ of P_b ($F(x) = G(x)$). If we can reject H_0 that P_m is drawn from the same distribution as the baseline P_b, a P-mutant is killed when its Mann-Whitney U test p-value is lower than the significance level α (0.01, 0.05 and 0.1). As we are interested in results when the mutants perform slower than the baseline, this null-hypothesis could potentially be incorrectly rejected if we observe P_m which execute faster than P_b. Therefore, we adopt the one-sided KS-test [42] with the null hypothesis H_0, which states $F(x) \leq G(x)$. By rejecting H_0, we can assert that P_m is stochastically larger than P_b. If the P_m by any workload of a P-mutant is larger than P_b by the corresponding workload, then the P-mutant is killed.

Fig. 4 shows the mutation scores of the three programs by $\mathbb{Q}3$ and $\mathbb{Q}4$ operators and different optimization levels. The mutation score is computed as the percentage of killed P-mutants among all normal FE P-mutants without timeouts, defined as

$$score_{mut} = \frac{\text{number of killed P-mutants}}{\text{number of normal P-mutants}} . \quad (1)$$

In Fig. 4, there are three mutation scores for each significance level (0.01, 0.05 and 0.1). The mutation score is measured in the interval of $[0, 1]$, where 1 means all P-mutants are killed and 0 means no P-mutants are killed.

Among all unoptimized cases, 65% P-mutants are killed, while 11% optimized P-mutants are killed ($\alpha = 0.1$). $\mathbb{Q}3$ P-mutants, those in `astar` for example, show a large discrepancy of mutation scores by different optimization levels, where unoptimized P-mutants have a score of 0.9 and optimized ones have 0.05 (the first two bars in Fig. 4). It is expected as the MPO values of unoptimized $\mathbb{Q}3$ operators are 10 times slower than unoptimized ones (Table III). $\mathbb{Q}4$ P-mutants are less likely to be killed and are more likely to be killed when unoptimized as well, except those by `mcf`.

We argue that PMT differs from classic MT in that PMT assesses the performance testing as a whole, rather than a particular workload or a profiler. Performance testing has many facets, including compiler optimizations, workload selection, selected performance diagnostic tools, or the repetition of the profiling, etc. The performance impacts by $\mathbb{Q}3$ and $\mathbb{Q}4$ P-mutants are believed to be small, unless repeated enough. The relatively higher mutation scores by unoptimized P-mutants (0.42) show that the P-mutants produced by SLOWCOACH do introduce performance overheads. Since software development is usually carried out with compiler optimization disabled, this also demontrates the capability of SLOWCOACH to assess the quality of the performance testing during the software's evolution.

The low mutation scores (overall 0.05%) by fully optimized P-mutants convey potential limitations of our performance testing enviornment. Firstly, the default workloads do not exercise the injected code frequently enough. This may be a scalability issue for production software as the release version of software is usually fully optimized, as the existing workloads cannot exhibit the performance bugs injected. Secondly, broader range of profilers with finer granularity need to be used in performance testing. The generic performance tool we use (`time`) to compute the mutation score suffers from too much noise and is not precise enough to measure millisecond performance impacts. As the MPO values in Table III illustrate, a single run of the injected code yields merely several milliseconds (sometimes even hundreds of microseconds) of performance overheads. Given external noises, small overheads are challenging to detect and profilers like `Linux perf` could be used to detect such overheads by sampling CPU [7].

1) Case Study on Context-dependent $\mathbb{Q}2$ Operators: SLOWCOACH also supports $\mathbb{Q}2$ operators to simulate real-world performance bugs given contextual information. We investigate

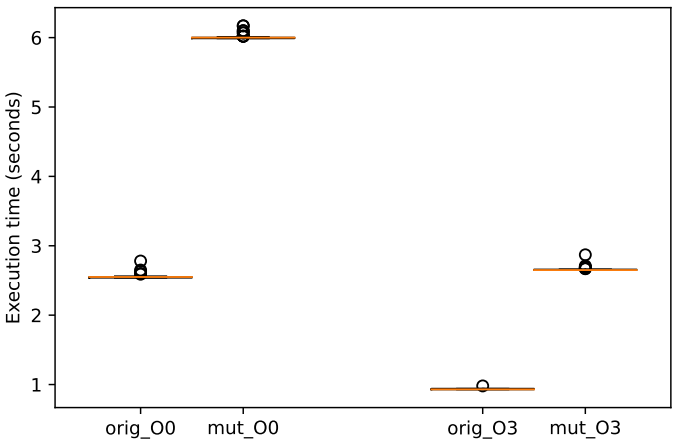

Fig. 5. Case 1 (ℚ2-A): The performance of the P-mutant whose `dfafast` is replaced by `false`.

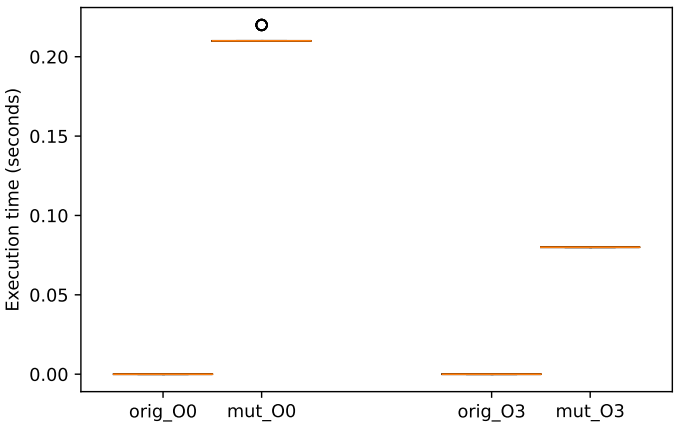

Fig. 6. Case 2 (ℚ2-B): The performance of the P-mutant whose function call to `fgrep_icase_available` is replaced by `false`.

the performance impact of such operators (cf. Section IV-B2) in a case study on `grep`. They each produces a P-mutant which reproduces the scenarios involving performance bugs [16] and preserves FE. The performance impacts of the P-mutants are shown in Figs. 5 and 6. The y-axis is the execution wall clock time of the testing program reported in seconds. Experiments are repeated 50 times, and Figs. 5 and 6 are the box plots of the performance values and their corresponding setup. Due to the small performance deviation, some boxes are overlapped with the median bar, and outliers are depicted as circles. Original program executions are labeled with *orig* and P-mutant executions with *mut*.

Since most real-world performance bugs involve performance bottlenecks for certain workloads, we use dedicated workloads for each mutant. In the first case (ℚ2-A), a 50 MiB file containing repeated "abcdabc" is searched for the pattern `'abcd.bd'`. The performance impact of the mutant in the `O0` case is with 3.45 s median difference. With `O3` optimization, the absolute impact is about 2.8 times smaller with 1.72 s. For the ℚ2-B case, a file with 600 capitalized strings is matched with its uncapitalized strings in a multibyte locale. The developer who originally fixed the bug that ℚ2-B re-introduces claims an overhead as large as 104 s[6]. However, we observe median runtimes of only 0.21 s for `O0` and 0.08 s for `O3`.

In summary, although ℚ2 operators produce representative mutants derived from real-world bugs, their specificity leads to limited applicability and the required domain knowledge entails manual effort.

E. Internal and External Validity

The validity of the conclusions we draw in this work may be affected by several factors. Our PMT operator selection could be biased as we focus our approach and evaluation on PMT operators derived from performance bugs that are widely discussed in the research community [3], [43], but there could be further classes of performance bugs that should be considered. Our random sample of P-mutants and mutation score based on statistical testing could be statistically biased. Our assessment of functional equivalence is based on program outputs, which assumes that the FE P-mutants yield identical outputs, which could be less robust. Bugs and mistakes in the implementation and data processing could also affect our conclusions. This is why we carefully tested and debugged our prototype implementation and all involved scripts and performed sanity checks on our collected data. We plan to make our implementation and data publicly available for review upon publication.

[6]https://www.mail-archive.com/bug-grep@gnu.org/msg06334.html

VI. Conclusion

In this paper, we presented and evaluated SlowCoach, a PMT framework. We subdivided the design space of PMT operators into four quadrants depending on whether they simulate effects of performance bugs or actual developer errors and identify functional equivalence as an key issue. We discuss concrete PMT operators from these quadrants and demonstrate how they can be derived from real-world performance optimizations and bugs. We demonstrate the applicability of our approach using 4 real-world software projects and show that our PMT operators can produce P-mutants that preserve functional equivalence and assess performance testing. We find that the mutation score based on one-sided statistical testing can provide reliable assessments of the quality of the performance testing, which could help provide suggestions on improving performance testing.

Acknowledgment

This research was supported, in part, by EC H2020 CONCORDIA GA No. 830927 and by the UKRI Trustworthy Autonomous Systems Node in Security [EPSRC grant EP/V026763/1].

Data Statement

The code used to generate the data analysed in this paper are both available at https://github.com/61OlkVq8/PMT.

References

[1] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and Detecting Real-World Performance Bugs," *SIGPLAN Not.*, vol. 47, no. 6, pp. 77–88, Jun. 2012, ISSN: 0362-1340. DOI: 10.1145/2345156.2254075.

[2] A. Nistor, P.-C. Chang, C. Radoi, and S. Lu, "Caramel: Detecting and Fixing Performance Problems That Have Non-Intrusive Fixes," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15, Florence, Italy: IEEE Press, 2015, pp. 902–912, ISBN: 9781479919345. DOI: 10.1109/ICSE.2015.100.

[3] L. Song and S. Lu, "Performance Diagnosis for Inefficient Loops," in *IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017, pp. 370–380. DOI: 10.1109/ICSE.2017.41.

[4] M. Attariyan, M. Chow, and J. Flinn, "X-Ray: Automating Root-Cause Diagnosis of Performance Anomalies in Production Software," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12, Hollywood, CA, USA: USENIX Association, 2012, pp. 307–320, ISBN: 9781931971966. DOI: 10.5555/2387880.2387910.

[5] P. Su, S. Wen, H. Yang, M. Chabbi, and X. Liu, "Redundant Loads: A Software Inefficiency Indicator," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19, Montreal, Quebec, Canada: IEEE Press, 2019, pp. 982–993. DOI: 10.1109/ICSE.2019.00103.

[6] N. R. Tallent, J. M. Mellor-Crummey, and A. Porterfield, "Analyzing Lock Contention in Multithreaded Applications," *SIGPLAN Not.*, vol. 45, no. 5, pp. 269–280, Jan. 2010, ISSN: 0362-1340. DOI: 10.1145/1837853.1693489.

[7] Perf Maintainers, *Perf Wiki*, Online: https://perf.wiki.kernel.org, Accessed on 2022-04-22.

[8] The LTTng Project, *LTTng: an open source tracing framework for Linux*, Online: https://lttng.org, Accessed on 2022-04-22.

[9] N. Rotem, L. Howes, and D. Goldblatt, *Warrior1: A Performance Sanitizer for C++*, 2020. arXiv: 2010.09583 [cs.SE].

[10] T. Yu and M. Pradel, "SyncProf: Detecting, Localizing, and Optimizing Synchronization Bottlenecks," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016, Saarbrücken, Germany, 2016, pp. 389–400, ISBN: 9781450343909. DOI: 10.1145/2931037.2931070.

[11] J. Chen, "Performance Regression Detection in DevOps," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*. New York, NY, USA: ACM, 2020, pp. 206–209, ISBN: 9781450371223. DOI: 10.1145/3377812.3381386.

[12] M. Chabbi, S. Wen, and X. Liu, "Featherlight On-the-Fly False-Sharing Detection," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '18, Vienna, Austria, 2018, pp. 152–167, ISBN: 9781450349826. DOI: 10.1145/3178487.3178499.

[13] S. Wen, X. Liu, J. Byrne, and M. Chabbi, "Watching for Software Inefficiencies with Witch," in *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18, Williamsburg, VA, USA, 2018, pp. 332–347, ISBN: 9781450349116. DOI: 10.1145/3173162.3177159.

[14] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman, "Mutation Testing Advances: An Analysis and Survey," *Advances in Computers*, 2018.

[15] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," *IEEE Transactions of Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011. DOI: 10.1109/TSE.2010.62.

[16] Y. Chen, S. Winter, and N. Suri, "Inferring Performance Bug Patterns from Developer Commits," in *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, Berlin, Germany, 2019, pp. 70–81. DOI: 10.1109/ISSRE.2019.00017.

[17] A. B. Sánchez, P. Delgado-Pérez, I. Medina-Bulo, and S. Segura, "TANDEM: A Taxonomy and a Dataset of Real-World Performance Bugs," *IEEE Access*, vol. 8, pp. 107 214–107 228, 2020. DOI: 10.1109/ACCESS.2020.3000928.

[18] S. Tizpaz-Niari, P. Černý, and A. Trivedi, "Detecting and Understanding Real-World Differential Performance Bugs in Machine Learning Libraries," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020, Virtual Event, USA, 2020, pp. 189–199, ISBN: 9781450380089. DOI: 10.1145/3395363.3404540.

[19] F. Zhou, Y. Gan, S. Ma, and Y. Wang, "Wperf: Generic off-cpu analysis to identify bottleneck waiting events," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'18, Carlsbad, CA, USA: USENIX Association, 2018, 527–543, ISBN: 9781931971478. DOI: 10.5555/3291168.3291207. [Online]. Available: https://doi.org/10.5555/3291168.3291207.

[20] M. M. u. Alam, T. Liu, G. Zeng, and A. Muzahid, "SyncPerf: Categorizing, Detecting, and Diagnosing Synchronization Performance Bugs," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17, Belgrade, Serbia, 2017, pp. 298–313, ISBN: 9781450349383. DOI: 10.1145/3064176.3064186.

[21] S. Wen, X. Liu, and M. Chabbi, "Runtime Value Numbering: A Profiling Technique to Pinpoint Redundant Computations," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, 2015, pp. 254–265. DOI: 10.1109/PACT.2015.29.

[22] L. Della Toffola, M. Pradel, and T. R. Gross, "Performance Problems You Can Fix: A Dynamic Analysis of

Memoization Opportunities," *SIGPLAN Not.*, vol. 50, no. 10, pp. 607–622, Oct. 2015, ISSN: 0362-1340. DOI: 10.1145/2858965.2814290.

[23] T. T. Chekam, M. Papadakis, and Y. Le Traon, "Mart: A Mutant Generation Tool for LLVM," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019, Tallinn, Estonia: ACM, 2019, pp. 1080–1084, ISBN: 9781450355728. DOI: 10.1145/3338906.3341180.

[24] T. T. Chekam, M. Papadakis, M. Cordy, and Y. L. Traon, "Killing Stubborn Mutants with Symbolic Execution," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 2, Jan. 2021, ISSN: 1049-331X. DOI: 10.1145/3425497.

[25] X. Devroey, G. Perrouin, M. Papadakis, A. Legay, P.-Y. Schobbens, and P. Heymans, "Model-based mutant equivalence detection using automata language equivalence and simulations," *Journal of Systems and Software*, vol. 141, pp. 1–15, 2018, ISSN: 0164-1212. DOI: 10.1016/j.jss.2018.03.010.

[26] G. Petrović, M. Ivanković, G. Fraser, and R. Just, "Does Mutation Testing Improve Testing Practices?" In *Proceedings of the 43rd International Conference on Software Engineering*. Madrid, Spain, 2021, pp. 910–921, ISBN: 9781450390859. DOI: 10.1109/ICSE43902.2021.00087.

[27] R. Natella, D. Cotroneo, and H. S. Madeira, "Assessing Dependability with Software Fault Injection: A Survey," *ACM Comput. Surv.*, vol. 48, no. 3, Feb. 2016, ISSN: 0360-0300. DOI: 10.1145/2841425.

[28] R. Natella, D. Cotroneo, J. A. Duraes, and H. S. Madeira, "On Fault Representativeness of Software Fault Injection," *IEEE Transactions on Software Engineering*, vol. 39, no. 1, pp. 80–96, 2013. DOI: 10.1109/TSE.2011.124.

[29] D. Cotroneo, A. Lanzaro, and R. Natella, "Faultprog: Testing the Accuracy of Binary-Level Software Fault Injection," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 1, pp. 40–53, 2018. DOI: 10.1109/TDSC.2016.2522968.

[30] R. Natella, S. Winter, D. Cotroneo, and N. Suri, "Analyzing the Effects of Bugs on Software Interfaces," *IEEE Transactions on Software Engineering*, vol. 46, no. 3, pp. 280–301, 2020. DOI: 10.1109/TSE.2018.2850755.

[31] P. Delgado-Pérez, A. B. Sánchez, S. Segura, and I. Medina-Bulo, "Performance mutation testing," *Software Testing, Verification and Reliability*, Jan. 2020. DOI: 10.1002/stvr.1728.

[32] S. Gupta, A. Saxena, A. Mahajan, and S. Bansal, "Effective Use of SMT Solvers for Program Equivalence Checking Through Invariant-Sketching and Query-Decomposition," in *Theory and Applications of Satisfiability Testing – SAT 2018*, O. Beyersdorff and C. M. Wintersteiger, Eds., Cham: Springer International Publishing, 2018, pp. 365–382, ISBN: 978-3-319-94144-8.

[33] N. P. Lopes and J. Monteiro, "Automatic equivalence checking of programs with uninterpreted functions and integer arithmetic," *International Journal on Software Tools for Technology Transfer*, vol. 18, no. 4, pp. 359–374, Feb. 2015. DOI: 10.1007/s10009-015-0366-1.

[34] D. Cotroneo, R. Natella, and S. Russo, "Assessment and Improvement of Hang Detection in the Linux Operating System," in *2009 28th IEEE International Symposium on Reliable Distributed Systems*, 2009, pp. 288–294. DOI: 10.1109/SRDS.2009.26.

[35] Y. Zhu, Y. Li, J. Xue, *et al.*, "What Is System Hang and How to Handle It," in *2012 IEEE 23rd International Symposium on Software Reliability Engineering*, 2012, pp. 141–150. DOI: 10.1109/ISSRE.2012.12.

[36] The GNU Project, *Savannah Git Hosting - grep.git*, Online: https://git.savannah.gnu.org/cgit/grep.git, Accessed on 2022-04-20.

[37] C. Lemieux, R. Padhye, K. Sen, and D. Song, "PerfFuzz: Automatically Generating Pathological Inputs," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018, Amsterdam, Netherlands: ACM, 2018, pp. 254–265, ISBN: 9781450356992. DOI: 10.1145/3213846.3213874.

[38] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana, "SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17, Dallas, Texas, USA: ACM, 2017, pp. 2155–2168, ISBN: 9781450349468. DOI: 10.1145/3133956.3134073.

[39] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, Sep. 2006, ISSN: 0163-5964. DOI: 10.1145/1186736.1186737.

[40] R. Atachiants, G. Doherty, and D. Gregg, "Parallel Performance Problems on Shared-Memory Multicore Systems: Taxonomy and Observation," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 764–785, 2016. DOI: 10.1109/TSE.2016.2519346.

[41] C. Cummins, P. Petoumenos, A. Murray, and H. Leather, "Compiler Fuzzing through Deep Learning," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018, Amsterdam, Netherlands: ACM, 2018, pp. 95–105, ISBN: 9781450356992. DOI: 10.1145/3213846.3213848.

[42] "Kolmogorov–Smirnov Test," in *The Concise Encyclopedia of Statistics*. New York, NY: Springer New York, 2008, pp. 283–287, ISBN: 978-0-387-32833-1. DOI: 10.1007/978-0-387-32833-1_214.

[43] S. Tsakiltsidis, A. Miranskyy, and E. Mazzawi, "On Automatic Detection of Performance Bugs," in *2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Ottawa, ON, Canada: IEEE, Oct. 2016, pp. 132–139. DOI: 10.1109/issrew.2016.43.