Evaluating Search-Based Software Microbenchmark Prioritization

Christoph Laaber, Tao Yue, and Shaukat Ali

***Abstract*—Ensuring that software performance does not degrade after a code change is paramount. A solution is to regularly execute software microbenchmarks, a performance testing technique similar to (functional) unit tests, which, however, often becomes infeasible due to extensive runtimes. To address that challenge, research has investigated regression testing techniques, such as test case prioritization (TCP), which reorder the execution within a microbenchmark suite to detect larger performance changes sooner. Such techniques are either designed for unit tests and perform sub-par on microbenchmarks or require complex performance models, drastically reducing their potential application. In this paper, we empirically evaluate single- and multi-objective search-based microbenchmark prioritization techniques to understand whether they are more effective and efficient than greedy, coverage-based techniques. For this, we devise three search objectives, i.e., coverage to maximize, coverage overlap to minimize, and historical performance change detection to maximize. We find that search algorithms (SAs) are only competitive with but do not outperform the best greedy, coverage-based baselines. However, a simple greedy technique utilizing solely the performance change history (without coverage information) is equally or more effective than the best coverage-based techniques while being considerably more efficient, with a runtime overhead of less than 1%. These results show that simple, non-coverage-based techniques are a better fit for microbenchmarks than complex coverage-based techniques.**

***Index Terms*—Software microbenchmarking, performance testing, JMH, search-based software engineering, multi-objective optimization, regression testing, test case prioritization.**

I. Introduction

REGRESSION testing comprises effective techniques for revealing faults in continuously evolving software systems [35], e.g., as part of continuous integration (CI) [25], [26], [33], [62]. To detect performance problems, particularly, in software libraries and frameworks, microbenchmarks are the predominantly used performance testing technique, similar to unit tests for functional testing [76]. However, their extensive runtimes inhibit their adoption in CI [40], [51], [54], [76]. In our previous work [54], we found that 15% of the *Java Microbenchmark Harness* (*JMH*) suites on GitHub run longer than three hours and 3% longer than 12 hours. Therefore, it is inevitable to employ performance regression testing techniques, such as software microbenchmark prioritization (SMBP) (test case prioritization (TCP) on software microbenchmarks), to capture important performance changes as early as possible.

Catching performance problems early is crucial for industry. Meta mentioned that performance regression testing is worth investigating [2], and MongoDB has created a sophisticated solution to run benchmarks as part of CI [13]. Though academic research on microbenchmarks has recently gained attention [15], [42], [51], [59], [76], performance regression testing on microbenchmark-level is still scarce, mostly focusing on regression test selection (RTS) for performance tests [3], [4], [11], [17]. Mostafa et al. [67] are the first to apply SMBP, introducing a technique based on a complex performance-impact model, which is, however, only suited for collection-intensive software and non-trivial to apply. In our previous work [56], we perform a large-scale study of coverage-based, greedy heuristics for SMBP, inspired by TCP [24], [64], [73], and found that the studied techniques are not nearly as effective on microbenchmarks as on unit tests and impose a relatively large runtime overhead of 17%.

A promising approach is search-based techniques, which are highly effective for various software engineering optimization problems [1], [5], [22], [27], [29], [41], [60], [65], [84]. Motivated by these successes, in this paper, we define search-based software microbenchmark prioritization (SBSMBP) problems and solve them with single-objective (SAs) and multi-objective evolutionary algorithms (MOEAs), with (up to) three objectives: (1) coverage (*C*), (2) coverage overlap among benchmarks (*CO*), and (3) historical performance change size (*CH*). Moreover, we define new *Greedy* SMBP techniques based on each of the three objectives as well as their combination (*C-CO-CH*).

We evaluate the SBSMBP (two single-objective SAs and six MOEAs) and *Greedy* SMBP techniques on 10 *JMH* suites having 1,829 distinct microbenchmarks with 6,460 distinct parameterizations across 161 versions, regarding (1) effectiveness measured by the average percentage of fault-detection on performance (*APFD-P*); and (2) efficiency measured as the runtime overhead. The study compares the SBSMBP and *Greedy* SMBP

Manuscript received 10 November 2022; revised 12 March 2024; accepted 20 March 2024. Date of publication 22 March 2024; date of current version 18 July 2024. This work was supported in part by The Research Council of Norway (RCN) under the project 309642 and in part by the Experimental Infrastructure for Exploration of Exascale Computing (eX3), which is supported by the RCN project 270053. Recommended for acceptance by A. Zaidman. *(Corresponding author: Christoph Laaber.)*

Christoph Laaber and Shaukat Ali are with Simula Research Laboratory, 0164 Oslo, Norway (e-mail: laaber@simula.no; shaukat@simula.no).

Tao Yue is with Beihang University, Beijing 100191, China (e-mail: yuetao@buaa.edu.cn).

Digital Object Identifier 10.1109/TSE.2024.3380836

techniques to two greedy, coverage-based baselines, i.e., *Total* and *Additional* [56], [67].

The results show that the best SBSMBP technique, i.e., the single-objective Generational Genetic Algorithm (*GA*) combining the three objectives with weighted-sum (*GA C-CO-CH*), performs competitively with the best greedy baseline, i.e., *Total*. However, it does not outperform *Total* regarding effectiveness, with both having an overall median *APFD-P* of 0.60, while only adding a minor additional overhead of 1% compared to below 1% for *Total* on top of the coverage extraction overhead. The best MOEA is Multi-Objective Cellular Genetic Algorithm (*MOCell*), which exhibits a lower median *APFD-P* of 0.58 than *GA C-CO-CH* and *Total*. Surprisingly, the *Greedy* technique that relies only on the historical performance change size (i.e., *Greedy CH*) has a higher median *APFD-P* of 0.65 than *Total*. Statistically, however, the best *Greedy* and SBSMBP techniques are only as effective as *Total*. Per project, *Greedy CH* is more effective than *Total* for one project and never worse, while having a significantly lower overhead of 1% consistently across the studied projects, whereas the coverage-based techniques have overheads ranging between 8% and 105% across the projects and 17% on average.

These results reveal that the *Greedy* technique relying solely on the performance change history (*CH*) performs the best and is arguably the simplest to implement. Hence, we recommend practitioners to employ the *Greedy CH* technique to achieve consistently earlier performance change detection, e.g., as part of CI, and researchers to investigate non-coverage-based SMBP techniques in the future.

To summarize, the main contributions of this paper are:

- the first paper to describe SMBP with search (SBSMBP);
- three search objectives (two of which are novel) to employ in SMBP algorithms;
- an experimental study showing the effectiveness and efficiency of the new *Greedy* SMBP and SBSMBP techniques compared to greedy, coverage-based baselines;
- an experimental study on the impact of change-awareness on the SMBP techniques; and
- an experimental study on the effectiveness and efficiency of two single-objective SAs and six MOEAs.

We provide all data and scripts in our replication package [58] and the SMBPs implementations in *bencher* `v0.4.0` [52].

II. Software Microbenchmarking With *JMH*

Software microbenchmarking is a performance testing technique that can be seen as the equivalent of unit testing for functional testing. A software microbenchmark, microbenchmark or benchmark for short and used thereafter, measures a performance metric, usually runtime or throughput, of small code units, such as statements and methods.

The *Java Microbenchmark Harness (JMH)* is the de facto standard for defining and executing Java microbenchmarks. They are defined in source code with annotations, similar to *JUnit*. Listing 1 shows a simplified example. A benchmark is a method annotated with `@Benchmark` (lines 8–15), which optionally takes parameters as input, i.e., an `Input` object

```
1  @Fork(3)
2  @Warmup(iterations = 10, time = 1, timeUnit =
       TimeUnit.SECONDS)
3  @Measurement(iterations = 20, time = 1, timeUnit =
       TimeUnit.SECONDS)
4  @BenchmarkMode(Mode.SampleTime)
5  @OutputTimeUnit(TimeUnit.NANOSECONDS)
6  public class ComputationSchedulerPerf {
7
8      @Benchmark
9      public void observeOn(Input input) {
10         LatchedObserver<Integer> o = input.
       newLatchedObserver();
11         input.observable
12             .observeOn(Schedulers.computation())
13             .subscribe(o);
14         o.latch.await();
15     }
16
17     @State(Scope.Thread)
18     public static class Input extends
       InputWithIncrementingInteger {
19         @Param({ "1", "1000" })
20         public int size;
21     }
22 }
```

Listing 1. Modified *JMH* example from *RxJava*

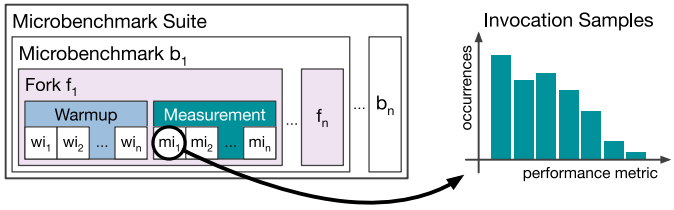

Fig. 1. JMH execution.

containing an instance variable annotated with `@Param` (lines 9 and 17–21), called the parameterization of a benchmark.

As measuring performance is non-deterministic and multiple factors can influence the measurement, one has to repeatedly execute each benchmark to get reliable results. *JMH* executes each combination of a benchmark method and its parameterization according to a specified configuration, set through annotations on class or method level (lines 1–5) or through the command line interface (CLI). Fig. 1 visualizes the repeated execution of a benchmark suite. Each (parameterized) benchmark is invoked as often as possible for a defined time period (e.g., 1 s, configured on lines 2 and 3), called an iteration, and the measured performance metrics are reported. To get reliable results, *JMH* first runs a set of warmup iterations (line 2 and "wi") to get the system into a steady state, for which the measurements are discarded. After the warmup, *JMH* runs a set of measurement iterations (line 3 and "mi"). To deal with non-determinism of the Java Virtual Machine (JVM), *JMH* repeats the sets of warmup and measurement iterations for a number of forks (line 1 and "f"), each in a new JVM instance. The result of a benchmark is then the distribution of all measurement iterations ("mi") of all forks ("f"). A microbenchmark suite usually contains many benchmarks, which *JMH* executes sequentially.

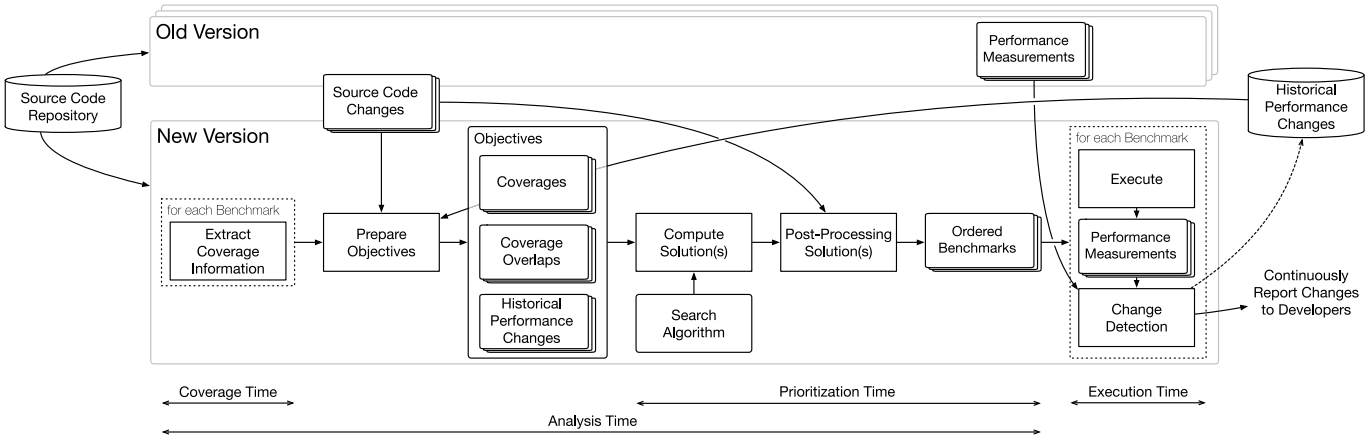

Fig. 2. Search-based software microbenchmark prioritization (adapted and extended from our previous work [56]).

III. Search-Based Prioritization

This section defines SBSMBP, which draws inspiration from search-based TCP [22], [27], [41], [60], [65] and greedy SMBP [56].

Fig. 2 depicts an overview of SBSMBP. Upon a new version, the source code is retrieved from a repository. First, the coverage information is extracted for every benchmark. This is different from TCP where coverage information is retrieved during the test execution, and the TCP technique uses the coverage of the old version for ranking the test execution of the new version. This is due to unit tests being usually only executed once. Because benchmarks are executed repeatedly for rigorous measurements, SBSMBP can leverage a single execution before the measurement to extract coverage information. Based on the coverage information, historical performance changes, and source code changes (in case the technique implements change-awareness), a pre-processing stage prepares the search objectives. Then, SBSMBP employs a SA to compute a SMBP ranking (solution), i.e., ordered benchmarks. SBSMBP can be parameterized with either a single-, multi-, or many-objective SA.

The post-processing stage has two tasks. (1) Select one solution from the SA results. In the single-objective SA case there is only one solution. In the multi-objective SA case, the SA returns the Pareto front containing multiple solutions to select from [28]. (2) Adjust the benchmark order based on the source code changes, when using a change-aware technique (see Section IV-C4). Finally, SBSMBP executes the whole benchmark suite in the optimized order. That is, it executes each benchmark repeatedly according to performance engineering best practice [30], measures the performance, and compares the measurements to the old version for change detection. The changes are then reported to the developers and stored for future versions, i.e., to compute the historical performance changes objective.

IV. Experimental Study

We conduct a laboratory experiment [77] to empirically study the effectiveness and efficiency of SBSMBP.

A. Research Questions

We pose the following research questions (RQs):

RQ1: How effective is SBSMBP?

- **RQ1a:** Which SBSMBP technique is most effective?
- **RQ1b:** How do SBSMBP techniques compare to greedy techniques?
- **RQ1c:** How does change-awareness impact effectiveness?

RQ2: How efficient is SBSMBP?

RQ1 addresses the effectiveness of SBSMBP by first investigating different SAs and objectives in RQ1a to find the most effective SBSMBP techniques, which we subsequently compare with greedy SMBP techniques in RQ1b, including the *Total* and *Additional* baselines. RQ1c assesses the impact of two source code change-aware techniques compared to the non-change-aware technique on SMBP effectiveness. Finally, RQ2 studies the runtime overhead that the SBSMBP techniques impose compared to the greedy techniques, which shows the practical feasibility.

B. Dataset

To perform our experimental evaluation, we require executions of (1) dedicated benchmarks (but not unit tests utilized as performance tests because their execution is not rigorous concerning performance testing and benchmarking best practice [30]); (2) full benchmark suites, as TCP/SMBP is defined by executing a full suite in a certain order [73], [74]; (3) multiple versions of the same project to perform regression testing; and (4) multiple projects to improve external validity. In addition, a rigorous performance evaluation demands high standards and compliance with the performance engineering best practice to reduce internal validity threats [30], [66]. We are only aware of one dataset that adheres to these criteria, which is from our previous work and was specifically created for SMBP [56], [57]. The dataset includes (1) *JMH* benchmark suite executions, (2) dynamic coverage information on method level for these benchmarks, and (3) coverage-based, greedy baselines for TCP

TABLE I
STUDY OBJECTS FROM THE DATASET [56]

Project	Versions	Benchmarks		Runtime [h]	
		Mean	Stdev	Mean	Stdev
Byte Buddy	31	30.74	± 8	0.26	± 0.069
Eclipse Collections	10	2,371.40	± 13	38.45	± 0.124
JCTools	11	126.91	± 52	1.15	± 0.481
Jenetics	21	49.24	± 6	0.42	± 0.053
Log4j 2	15	309.53	± 162	2.71	± 1.398
Netty	10	746.50	± 522	6.56	± 4.625
Okio	11	181.64	± 20	1.56	± 0.170
RxJava	19	842.63	± 228	7.81	± 2.113
Xodus	11	67.00	± 10	1.33	± 0.104
Zipkin	22	55.18	± 11	0.48	± 0.101

on benchmarks. Though there are a few papers on microbenchmarking and performance regression testing [10], [11], [17], [67], [78], no other dataset fits the above-mentioned criteria. Hence, we only select the dataset of our previous work [56], [57] for our experiment.

1) Study Objects: The dataset has 10 open-source software (OSS) Java projects hosted on GitHub, which contain *JMH* suites. The projects have 59,164 benchmarks in total and 6,460 distinct benchmarks across 161 versions. Distinct benchmarks are counted once across all versions they occur in. In this paper, we consider a benchmark to be the instantiation of a *JMH* benchmark method (annotated with `@Benchmark`) with concrete parameterization (using *JMH* parameters annotated with `@Param`). Table I provides an overview of these projects. Columns "Benchmarks" and "Runtime" show the arithmetic mean and standard deviation across the versions of each project, as different versions potentially have a different number of benchmarks and, hence, varying runtimes.

2) Benchmark Executions and Performance Changes: We executed the benchmarks in a controlled environment for a predefined number of repetitions, following the best practice [30], [66]. The executions used a unified configuration for all the benchmarks, i.e., 10 warmup iterations and 20 measurement iterations of 1 s each. In addition, we executed the full suites for 3 trials (note that a trial is different from a *JMH* fork, in that it is executed at different points in time and not back-to-back), leading to a total runtime for all the projects, versions, and repetitions of approximately 89 days.

Based on the executions, the performance changes are computed between adjacent versions with a Monte-Carlo technique to estimate the confidence intervals of the ratio of the means. The technique is based on Kalibera and Jones [45], [46] and employs bootstrap [16] with hierarchical random resampling with replacement [71] on three levels, i.e., trial, iteration, and benchmark invocation. It uses 10,000 bootstrap iterations [39] and a confidence level of 99%. For this, we used the *pa* tool [50].

3) Coverage Information and SMBP Baselines: The dataset provides the necessary coverage information for the SBSMBP, *Greedy* SMBP, and greedy baseline techniques. Dynamic coverage information, i.e., method coverage, is extracted per benchmark and version with *JaCoCo*[1], by executing a benchmark once (with *JMH*'s single-shot mode) and injecting the *JaCoCo* agent. Based on these coverages, we [56] studied coverage-based, greedy SMBP techniques, particularly *Total* and *Additional* — our baselines.

C. Independent Variables

Our experiment investigates four independent variables: (1) the prioritization strategies, (2) the SAs employed in the SB-SMBP techniques, (3) the search objectives used for solving the SBSMBP problem, and (4) the change-awareness of the technique, which are described below in detail.

1) Prioritization Strategies: The four different prioritization strategies are: (1) the coverage-based, greedy *Total* strategy, which ranks the benchmarks in a suite, based on their total number of covered units, from the one with the most to the one with the least; (2) the coverage-based, greedy *Additional* strategy, which ranks the benchmarks, based on the number of covered units not yet been considered for prioritization by previously ranked benchmarks; (3) a *Greedy* strategy which ranks the benchmarks according to either one or three of the search objectives; and (4) the search-based strategy described in Section III. The strategies relying on code coverage use dynamic method coverage, aligned with our previous work [56].

The *Total* and *Additional* strategies have their roots in unit testing [73], [74], are the standard coverage-based techniques with good effectiveness [34], [64], and were recently adapted for benchmarks [56], [67]. Our experiment parameterizes *Total* and *Additional* with a benchmark granularity for both the prioritization strategy and the coverage extraction on the parameter level, which we showed to be optimal [56]. The *Greedy* strategy based on the search objectives uses either coverage, coverage-overlap, performance change history, or a combination of these. The parameterization of the search-based strategy is mostly concerned with the employed SA and search objectives, which are distinct independent variables of our experiment that the next sections describe.

2) Search Algorithms (SAs): We study eight SAs: (1) Steepest Ascent Hill Climbing (*HC*), (2) Generational Genetic Algorithm (*GA*), (3) Indicator-Based Evolutionary Algorithm (*IBEA*) [85] using the hypervolume [7], (4) Multi-Objective Cellular Genetic Algorithm (*MOCell*) [68], (5) Non-Dominated Sorting Genetic Algorithm II (*NSGAII*) [19], (6) Non-Dominated Sorting Genetic Algorithm III (*NSGAIII*) [18], (7) Pareto Archived Evolution Strategy (*PAES*) [48], and (8) Strength Pareto Evolutionary Algorithm 2 (*SPEA2*) [47], [86]. The first two are single-objective SAs, and the other six are MOEAs. We select these to cover a wide range of algorithms that have been used in previous search-based test optimization research [22], [27], [60], [82] and are supported by *jMetal* [69].

Solution Encoding A SBSMBP solution is encoded as an integer permutation, where its length corresponds to the number

[1]https://www.jacoco.org/jacoco/

of benchmarks in a suite, and each element corresponds to an integer identifier mapping to a unique benchmark in that suite. We choose this encoding because the SMBP problem has the constraint that each benchmark exists exactly once in every solution. SBSMBP then executes the benchmark suite in the order of the solution.

Algorithm Parameters Our experiment uses the same parameter settings across the SAs where possible to facilitate a fair comparison, which are also in line with previous research on multi-objective TCP [22], [27], [41], [61], [65]. The algorithm parameters are set to:

- population size: 250;
- selection: binary tournament selection;
- crossover: PMX-Crossover with a probability $p_c = 0.9$;
- mutation: SWAP-Mutation with a probability $p_m = 1/n$, where n is the number of benchmarks to prioritize;
- maximum number of generations: 100; and
- maximum evaluations: population size times the maximum number of generations, i.e., 25,000.

Note that not all SAs make use of all the parameters. Archive-based MOEAs, i.e., IBEA, *MOCell*, and PAES, use an archive size equal to the population size, which is in line with the papers that introduced them [48], [68], [85] and the *JMetal* defaults. *MOCell* is an exception, which requires a population size that is the power of 2 of an integer; hence, we set *MOCell*'s population size to 256, which is the closest to the population size of the other MOEAs, i.e., 250, and its maximum number of generations to 25,600.

***HC* Neighborhood**. We define the neighborhood of a solution as the benchmark orderings that swap the first benchmark with each of the other benchmarks. The neighborhood size is $n - 1$, where n is the number of benchmarks in the suite. This definition is in line with previous research to keep *HC* scalable [60]. Other definitions would also be valid, e.g., swapping any two benchmarks; however, this would not scale, as for every iteration *HC* would need to check $\mathcal{O}(n^2)$ neighbors.

3) Search Objectives: The ideal search objective would be the actual performance changes (faults) the benchmark suite detects upon a new release, which, however, are only known after the execution (see Section II for details). Hence, the search requires objectives that are good proxies for performance changes, which are available before the repeated suite execution.

Search-based TCP often relies on coverage metrics [22], [27], [60], such as average percentage element coverage (APEC) [60] inspired by average percentage of fault-detection (*APFD*) [22], [60]. APEC requires $\mathcal{O}(mn)$, where m and n are the number of elements and tests, because it considers the additionally covered elements ("additional coverage") for each test (not covered by an already ranked test) [27], as it is a better proxy for fault detection than "total coverage" [63], [64].

Differently for SMBP, "total coverage" leads to more effective rankings than "additional coverage" [56]. Hence, we refrain from using APEC and employ objectives that favor "total coverage", i.e., average percentage total elements coverage (APTEC) in Eq. (1) inspired by *APFD-P* [56], [67].

$$APTEC = \frac{\sum_{b=1}^{n} \frac{prev(b-1)+elements(b)}{m_{total}}}{n} \tag{1}$$

where m_{total} is the number of total covered elements; n is the number of benchmarks; $prev()$ is the cumulative element coverage until the previous benchmark $b-1$ in a SMBP ranking; and $elements(b)$ is the number of covered elements by benchmark b. Note, m_{total} potentially contains duplicate elements; e.g., if two benchmarks cover the same element, it is counted twice in m_{total}. APTEC only requires $\mathcal{O}(n)$ to compute instead of $\mathcal{O}(mn)$ for APEC, as we can use memoization in $prev$: $prev(b) = prev(b-1) + elements(b)$, where $prev(0) = 0$. What constitutes an element depends on the specific search objective.

We define **three objectives** based on APTEC:

Coverage (C, maximize): This objective is akin to code coverage objectives in search-based TCP [60], but with APTEC, aiming to cover more code elements that are more likely to expose performance changes. While Laaber et al. [56] showed that SMBP that solely relies on code coverage is only slightly more effective than a random strategy and code coverage has a low correlation with the performance change size, coverage is still one factor that can be used as a proxy for performance changes. Coverage maximizes the total number of code elements covered by a benchmark b as returned by $elements(b)$.

Coverage Overlap (CO, minimize): Due to Coverage greedily selecting benchmarks based on static coverage information (i.e., already covered code elements remain, which is akin to *Total*), the SBSMBP techniques are prone to rank benchmarks covering the same code elements similarly. To achieve coverage diversity among benchmarks, Coverage Overlap minimizes the cumulative coverage overlap among them. For Coverage Overlap, $elements(b)$ returns the average overlap between a benchmark b and all other benchmarks in the suite.

Historical Performance Change (CH, maximize): This objective maximizes the historically-detected performance change size of early-ranked benchmarks. The idea is that benchmarks that have previously detected large performance changes are more likely to detect performance changes in the future and, hence, should be ranked earlier. This is similar to search-based TCP, which introduces "Fault History Coverage" as an objective [27]. For this objective, $elements(b)$ returns the average performance change size of benchmark b across all previous versions.

Note, we do not employ delta coverage and benchmark execution cost as search objectives, as our preliminary experiments showed that delta coverage does not improve effectiveness and benchmark execution cost is approximately the same for all benchmarks when using a unified execution configuration (number of repetitions).

Our study investigates different combinations of SAs and objectives across all RQs. Since each objective can be considered a proxy metric for finding effective SMBP rankings, it is necessary to select one or more objectives to form different search problems and compare their effectiveness. Particularly, with single-objective SAs, we solve four problems: (1) three one-objective problems, each formed with each of the three objectives; and (2) one three-objective problem, formed by aggregating the three objectives $o_i \in O$ into a single one o' with the classical weighted-sum approach, defined in Eq. (2), using equal weights $w_i = \frac{1}{3}$, similar to Yoo and Harman [82].

$$o' = \sum_{i=1}^{|O|}(w_i * o_i), \sum_{i=1}^{|O|} w_i = 1 \tag{2}$$

We study the effectiveness and efficiency of the single-objective SAs solving the above four problems and the MOEAs solving the three-objective problem (with individual objectives). We refer to the SBSMBP techniques by its SA and employed objectives, e.g., *GA C-CO-CH*.

4) Change-Awareness: This independent variable studies whether "change-awareness" of the SMBP technique improves its effectiveness. That is, does the technique perform better if it considers the code that has changed since the last version in a dedicated way? This is inspired by Mostafa et al. [67], who studied change-aware approaches for SMBP, and research on multi-objective search-based TCP considering "delta coverage" as objective [27]. We consider three approaches: (1) Non-Change-Aware (*nca*), which uses the full coverage information of the current version to encode the objectives *C* and *CO*; (2) Change-Aware Coverage (*cac*), which relies only on coverage information that has changed since the last version, i.e., retains only changed methods, for the objectives *C* and *CO*; (3) Change-Aware Ranking (*car*), which uses the full coverage information (as for *nca*) to rank benchmarks and afterwards groups benchmarks covering a changed method before benchmarks not covering any changed method, while retaining the initial order.

D. Dependent Variables

The dependent variables of our study involve a set of effectiveness and efficiency metrics: one effectiveness variable, i.e., *APFD-P*, to answer RQ1 and its sub-questions; and two efficiency variables, i.e., the prioritization and analysis (total) runtime overhead of the technique, to answer RQ2. They all have been used in previous research on SMBP [56], [67].

1) Effectiveness: The effectiveness measures rely on the performance changes that the benchmarks of a suite detect between two adjacent versions. We rely on the measurements and computed changes (see Section IV-B2) from our previous work [56], where a change is defined based on the bootstrapped confidence interval of the mean difference and not just the mean difference. Note that we do not distinguish the two possible directions of a change, i.e., performance regression (slowdown) or improvement (speedup), but only consider these as a change of a certain size. This aligns with both previous works on SMBP [56], [67]. Assuming that function $change$ returns the change of a benchmark between two versions as a percentage, $change$ is defined as $change : B \mapsto \mathbb{Z}^{0+}$, where B is the set of benchmarks in a suite.

APFD-P [67] adapts *APFD* [73] from unit testing. *APFD* itself is inapplicable for benchmarks, as benchmarks have continuous outputs (i.e., different fault severities; see $change$) as opposed to the discrete ones (pass or fail) for unit tests. *APFD-P*, as an area under the curve (AUC) metric, assesses the fault-detection capabilities of a SMBP technique, ranging from 0 to 1, with 0 and 1 denoting the worst and optimal rankings of a benchmark suite, respectively. A technique that ranks benchmarks with larger changes higher than those with smaller changes is considered better and has a larger *APFD-P* value, as defined in Eq. (3).

$$APFD\text{-}P = \frac{\sum_{x=1}^{n} \frac{detected(x)}{c}}{n} \tag{3}$$

where n is the benchmark suite size; c is the sum of the changes of all the benchmarks; $detected(x)$ returns the cumulative change of the first x benchmarks, see Eq. (4).

$$detected(x) = \sum_{i=1}^{x} change(i) \tag{4}$$

where $change(i)$ is the ith benchmark's change in a ranking.

Previous SMBP works [56], [67] also study normalized discounted cumulative gain (nDCG) and *Top-3*, which we do not consider due to similar results to *APFD-P* in our study.

2) Efficiency: Our experiment evaluates the technique's efficiency with two dependent variables in RQ2. Both concern the technique's runtime overhead imposed on the overall benchmarking time: (1) **Prioritization time** is the time required to run a SMBP technique, i.e., computing the SMBP ranking with all necessary inputs already available; and (2) **Analysis time** is the total time necessary to prioritize a benchmark suite, including extracting all the required information (coverage information and historically-detected performance changes) and the prioritization time. Both times are studied as overhead (in percentage) of the benchmark suite runtimes (see Table I), which is required for comparability across the projects. The overhead of *Additional* and *Total* (this study's baselines) is dominated by the time to extract the coverage information [56]. Note that we neither study the coverage extraction time, as all the coverage-based techniques rely on the same coverage type, nor the time to extract the historically detected changes, as these are available from previous versions and are not computed online at the time of prioritization.

E. Experiment Setup

To deal with the stochastic nature of the SBSMBP techniques, we follow recommended practice [6] and previous research on search-based TCP [22], [27], [61], [65], and repeatedly execute each prioritization 30 times, i.e., for each project, version, and SA, which are called "repetitions" in the rest of the paper. The effectiveness calculations and results, unless otherwise specified, rely on all the repetitions for analyses.

Regarding the variance of a Pareto front (for each project, version, MOEA, and repetition), we select the solution with the median effectiveness for analysis as similarly done by previous research on multi-objective TCP [22], [27], [65]. Other selection strategies, e.g, choosing the solution at a knee point [9], are arguably less applicable in our context where the objectives are just proxies for the context-dependent effectiveness (i.e., *APFD-P*) of SMBP. The efficiency analysis, however, retains the measurements of all the solutions because more performance measurements raise the confidence that the results are representative [30], [53], [66].

Rigorously executing performance measurements is crucial to the validity of the results [30], [66]. There are two basic principles to ensure reliable measurements: (1) sufficient numbers of repetitions and (2) a controlled execution environment. This paper assesses the overhead added to the full benchmark suite, which is in the order of minutes and hours (see Table I). Hence, minor measurement inaccuracies are permissible, as they will not change the overall results. Best practice suggests 30 repeated measurements for every distinct measurement scenario (combinations of projects, versions, and SAs). We reuse the repetitions for tackling the SAs' stochasticity are as performance measurements, i.e., one measurement per repetition.

To ensure reliable measurements, a controlled execution environment is desirable. However, due to the dimension of the runtimes, it is acceptable to use a slightly less controlled environment, as the results will not be impacted much. Hence, we refrain from employing a tightly controlled bare-metal environment and use a high-performance computing (HPC) cluster (Experimental Infrastructure for Exploration of Exascale Computing (eX^3) cluster[2]) instead, as it allows for parallelization to keep the experiment runtime lower. eX^3 is hosted at the first author's institution, which uses Slurm `21.08.8-2` as its cluster management software. The experiment executes the measurements on nodes of the same type (using the *defq* partition of eX^3), with 30 central processing unit (CPU) cores assigned to and a single CPU exclusively reserved for the measurement process. The nodes have AMD EPYC 7601 32-core processors, run Ubuntu `22.04.3`, and have 2 TB total memory shared for all CPUs. The experiments were conducted in the second half of 2023.

F. Statistical Analyses

We employ hypothesis testing in combination with effect size measures to compare different observations. An observation in our context can be, e.g., a single *APFD-P* value of a project, in a version, using one MOEA, for a single repetition. Depending on the analysis and the RQ, the compared set of observations changes, however, the tests remain the same. We follow the best practice for evaluating search algorithms [6].

We use the non-parametric Kruskal–Wallis H [49] test to compare multiple sets of observations. Note that Arcuri and Briand [6] suggest using the Mann-Whitney U test, which, however, only compares two sets of observations. The Kruskal-Wallis H test is considered an extension of the Mann-Whitney U test and is, therefore, compatible with the best practice by Arcuri and Briand [6]. The null hypothesis H_0 states that the different distributions' medians are the same, and the alternative hypothesis H_1 is that they are different. If H_0 can be rejected, we apply Dunn's post-hoc test [23] to identify which pairs of observations are statistically different.

Note that we consciously decided against using the Friedman test, which is used by the SBFT[3] tool competition [21] because our experiment setup *does not* fit into a complete block design as required for the Friedman test. To use the Friedman test (e.g., where the projects are the subjects and the SAs are the treatments), we would need to aggregate the dependent variables of the different versions and experiment repetitions into a single value (e.g., taking the median of the median), which would completely disregard their variability. Doing so is unacceptable and, hence, we use the Kruskal-Wallis H test instead, though it does not distinguish between the different subjects (projects). To alleviate it, we add additional analysis on a per-project level.

We also report the effect size with Vargha-Delaney $\hat{A}_{12}$ [80] to characterize the magnitude of the differences. Two groups of observations are the same if $\hat{A}_{12} = 0.5$. If $\hat{A}_{12} > 0.5$, the first group is larger than the second, otherwise if $\hat{A}_{12} < 0.5$. The magnitude values are divided into four nominal categories, which rely on the scaled $\hat{A}_{12}$ defined as $\hat{A}_{12}^{scaled} = (\hat{A}_{12} - 0.5) * 2$ [38]: "negligible" ($|\hat{A}_{12}^{scaled}| < 0.147$), "small" ($0.147 \leq |\hat{A}_{12}^{scaled}| < 0.33$), "medium" ($0.33 \leq |\hat{A}_{12}^{scaled}| < 0.474$), and "large" ($|\hat{A}_{12}^{scaled}| \geq 0.474$).

All results are considered statistically significant at significance level $\alpha = 0.01$ and with a non-negligible effect size. We control the false discovery rate with the Benjamini-Yekutieli procedure [8] where multiple comparisons are performed. It is considered a more powerful procedure than, e.g., the Bonferroni or Holm corrections for family-wise Type I errors.

G. Threats to Validity

1) Construct Validity: We rely on *APFD-P* as the metric for SMBP effectiveness, which builds on the performance change size as a measure for benchmark importance. Different metrics and measures are likely to impact the results and conclusions of this study. Nevertheless, recent studies in performance regression testing have relied on the performance change size and *APFD-P* for SMBP [12], [17], [56], [67]. The procedure to compute the performance change size is from our previous work [56]. Consequently, this paper suffers from the same validity threats in this regard.

2) Internal Validity: Regarding the threat introduced by the SAs' stochasticity, we repeatedly executed the algorithms and chose the relevant statistical tests and the effect size measure by following the best practice reported by Arcuri and Briand [6]. Regarding the measurement inaccuracies of the runtime overhead, we tried to mitigate this by executing on dedicated HPC nodes and repeated runtime measurements, following the performance engineering best practices [30]. Regarding setting the SAs hyperparameters, different hyperparameters might

[2]https://www.ex3.simula.no/

[3]International Workshop on Search-Based and Fuzz Testing

TABLE II
APFD-P EFFECTIVENESS FOR THE SBSMBP TECHNIQUES (SAS AND OBJECTIVES) AND PROJECTS ACROSS ALL THE VERSIONS AND REPETITIONS. THE VALUES ARE THE MEDIAN ± THE MEDIAN ABSOLUTE DEVIATION (MAD)

SA	Objectives	Overall	Projects									
			Byte Buddy	*Eclipse Collections*	*JCTools*	*Jenetics*	*Log4j 2*	*Netty*	*Okio*	*RxJava*	*Xodus*	*Zipkin*
HC	*C*	0.51±0.10	0.52±0.16	0.50±0.02	0.50±0.07	0.53±0.20	0.50±0.07	0.51±0.08	0.51±0.13	0.50±0.04	0.54±0.10	0.51±0.17
	CO	0.50±0.10	0.51±0.16	0.50±0.03	0.50±0.08	0.49±0.17	0.49±0.06	0.51±0.08	0.56±0.16	0.50±0.04	0.51±0.08	0.52±0.15
	CH	0.51±0.10	0.54±0.17	0.50±0.02	0.51±0.07	0.53±0.18	0.50±0.06	0.51±0.08	0.51±0.13	0.50±0.04	0.53±0.11	0.51±0.14
	C-CO-CH	0.51±0.10	0.54±0.16	0.50±0.03	0.51±0.08	0.52±0.19	0.51±0.07	0.51±0.07	0.50±0.14	0.50±0.04	0.50±0.10	0.53±0.15
GA	*C*	0.56±0.14	0.49±0.26	0.53±0.02	0.62±0.10	0.63±0.11	0.60±0.08	0.52±0.09	0.67±0.11	0.53±0.06	0.70±0.05	0.47±0.17
	CO	0.48±0.13	0.4±0.14	0.52±0.04	0.49±0.08	0.38±0.18	0.44±0.07	0.51±0.08	0.74±0.16	0.51±0.05	0.40±0.06	0.54±0.13
	CH	0.58±0.13	0.68±0.14	0.54±0.03	0.59±0.09	0.57±0.28	0.58±0.07	0.54±0.08	0.59±0.18	0.55±0.07	0.59±0.15	0.59±0.14
	C-CO-CH	0.60±0.13	0.68±0.13	0.55±0.03	0.65±0.07	0.59±0.21	0.61±0.09	0.54±0.09	0.68±0.14	0.56±0.08	0.71±0.05	0.55±0.16
IBEA	*C-CO-CH*	0.57±0.12	0.61±0.15	0.54±0.03	0.59±0.06	0.58±0.23	0.56±0.06	0.51±0.05	0.69±0.15	0.55±0.06	0.67±0.08	0.54±0.14
MOCell	*C-CO-CH*	0.58±0.12	0.59±0.13	0.56±0.03	0.60±0.07	0.57±0.20	0.56±0.06	0.53±0.07	0.76±0.13	0.58±0.07	0.64±0.08	0.52±0.17
NSGAII	*C-CO-CH*	0.56±0.11	0.60±0.11	0.54±0.02	0.59±0.06	0.55±0.20	0.55±0.05	0.52±0.05	0.77±0.14	0.55±0.06	0.63±0.07	0.51±0.17
NSGAIII	*C-CO-CH*	0.56±0.12	0.59±0.14	0.54±0.03	0.59±0.07	0.57±0.20	0.55±0.06	0.53±0.07	0.76±0.14	0.55±0.06	0.63±0.08	0.52±0.17
PAES	*C-CO-CH*	0.52±0.10	0.54±0.17	0.50±0.03	0.56±0.08	0.55±0.18	0.51±0.07	0.51±0.07	0.58±0.13	0.50±0.04	0.57±0.09	0.53±0.15
SPEA2	*C-CO-CH*	0.56±0.11	0.59±0.11	0.54±0.02	0.59±0.06	0.54±0.19	0.55±0.05	0.52±0.05	0.76±0.16	0.55±0.05	0.63±0.07	0.50±0.18

lead to different conclusions. We followed previous research in search-based TCP [22] and used a unified configuration for all the eight SAs.

3) External Validity: The generalizability of our results is concerned with the number of datasets used, and consequently with the projects, versions, and benchmarks. We rely on a single dataset from our previous work [56], which is, to the best of our knowledge, the only one that fits the study. All the projects are written in Java and use *JMH* benchmarks. Our results do not generalize to projects written in other languages and with other benchmarking frameworks. The same limitation applies to other types of performance tests, such as application-level load and stress tests [43], and macro- and application-benchmarks [14], [31]. Finally, the results depend on performance changes executed in controlled, bare-metal environments; hence, they are not generalizable to SMBP effectiveness assessed based on changes observed in less-controlled environments, e.g., the cloud.

V. RESULTS AND ANALYSES

This section describes the results to answer the study's RQs.

A. RQ1: Effectiveness

This section studies the effectiveness of eight different SAs in Section V-A1; compares the best SAs to the greedy baselines, i.e., *Additional*, *Total*, and the *Greedy* strategy in Section V-A2; and assesses the impact of change-awareness in Section V-A3.

1) RQ1a: Effectiveness of SBSMBP Techniques: This section compares the *APFD-P* effectiveness of the eight SAs across all the projects and then per project. In total, we study 14 SBSMBP techniques, based on the SA and search objective combinations (see Section IV-C3). Table II shows the effectiveness results overall and per project.

Overall. From Table II's "Overall" column, we observe that the median *APFD-P* ranges from 0.48 to 0.60. The SBSMBP technique with the highest median *APFD-P* is *GA C-CO-CH* with 0.60, followed by *GA CH* with 0.58 and *MOCell* with 0.58. Interestingly, *GA CH* performs competitively by only relying on the historical performance change size and without relying on coverage. We further notice that *HC* (all objectives) and *GA CO* exhibit the worst effectiveness.

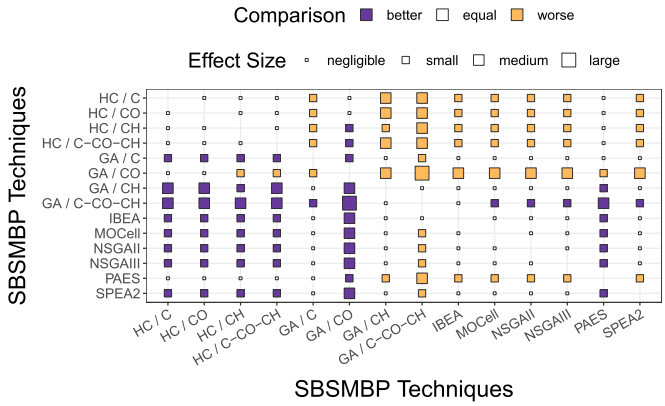

Fig. 3. Overall *APFD-P* effectiveness compared pair-wise for the SBSMBP techniques across all the versions and repetitions.

In addition, we conduct pair-wise comparisons among all the SBSMBP techniques with the statistical tests in Fig. 3. The results confirm that *HC* and *GA CO* are statistically worse than all the other SAs except *PAES*. *HC* is the only local search algorithm and the most primitive in our study; consequently, it is unsurprising that it performs the worst. *GA CO* performs worse than the other *GA*s and the MOEAs because it only optimizes coverage overlap.

The single-objective *GA C-CO-CH* also performs the best statistically, better than all the other SAs except *GA CH* and *IBEA* with at least a small effect size. This is interesting because a simple (single-objective) *GA* using weighted-sum performs equally or better than the MOEAs, which shows that in our context, using a MOEA does not lead to higher effectiveness. In addition, using *GA* instead of any MOEA has the benefit that the output is a single benchmark ranking and not a Pareto front from which a solution has to be selected. The remaining *GA* and MOEA strategies are statistically equivalent though they show a minor difference in median effectiveness.

Per Project. Table II shows the median *APFD-P* per project, and Fig. 4 depicts the pair-wise statistical test results. We observe that the effectiveness depends on the project the SMBP

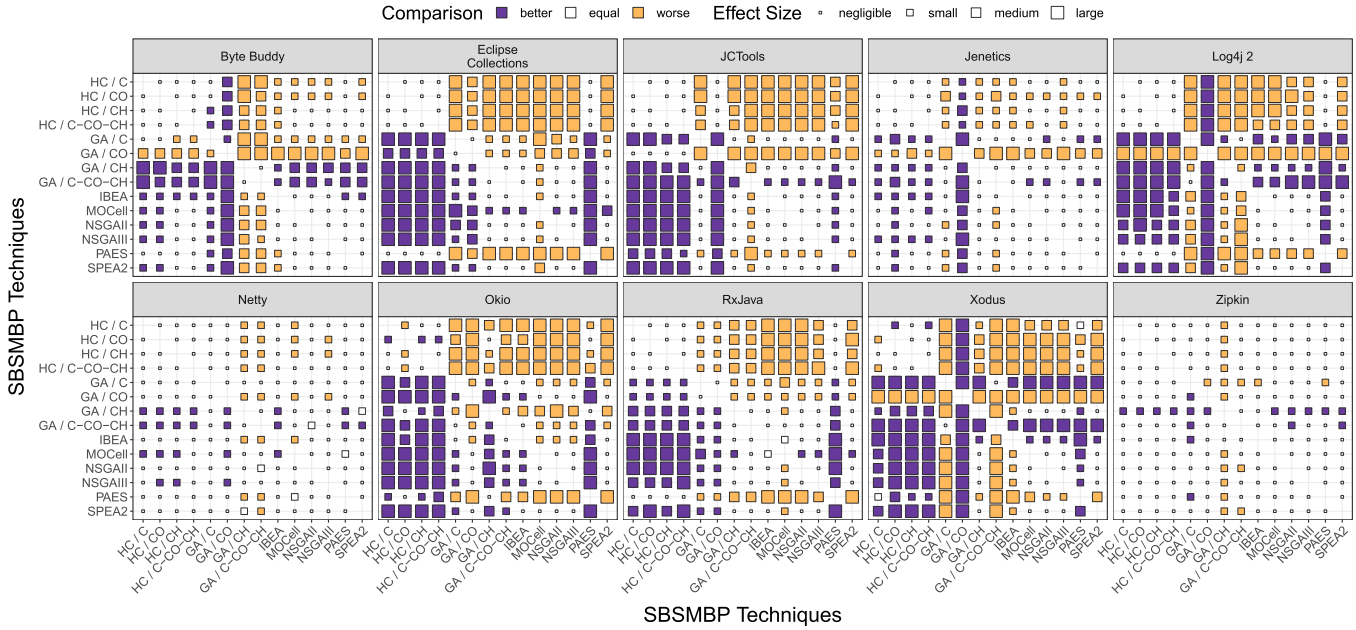

Fig. 4. Per project *APFD-P* effectiveness compared pair-wise for the SBSMBP techniques all the versions and repetitions.

technique is applied to, e.g., for *GA C-CO-CH*, *APFD-P* ranges from 0.54 (for *Netty*) to 0.71 (for *Xodus*).

We mostly notice a similar pattern to the overall results, i.e., *HC* and *GA CO* perform the worst. The only exception is *Okio* where *GA CO* performs well (*NSGAII* performs even better on median, while the other MOEAs except *PAES* are also effective). The reason for this is that *Okio* is the project with the most disjoint coverage sets among the benchmarks and, consequently, a SBSMBP technique with only the *CO* objective can perform well. Moreover, we observe that for *Netty* and *Zipkin*, the SA choice largely does not matter.

The most effective strategies for each project are: (1) *GA C-CO-CH* for six, i.e., *Byte Buddy*, *JCTools*, *Jenetics*, *Log4j 2*, *Netty*, and *Xodus*; (2) *GA CH* for two, i.e., *Byte Buddy* and *Zipkin*; (3) *MOCell* for two, i.e., *Eclipse Collections* and *RxJava*; and (4) *NSGAII* and *GA CO* for one, i.e., *Okio*.

The reasons why a SA is more effective depends on the characteristics of the studied projects and the problem definition. As we are evaluating the SAs on real-world projects, it is impossible to say which characteristics are responsible for the observed differences. To provide such reasons, we would need to run a controlled experiment where we identify certain project characteristics, create projects and benchmark suites with these, and investigate their impact. Such a study is out-of-scope of this paper.

RQ1a Summary: *GA C-CO-CH* is the most effective SBSMBP technique overall, as it performs the best for six of the ten projects. The other SBSMBP techniques worth considering are *GA CH*, particularly because it does not require coverage information, and *MOCell*, which is the most effective MOEA.

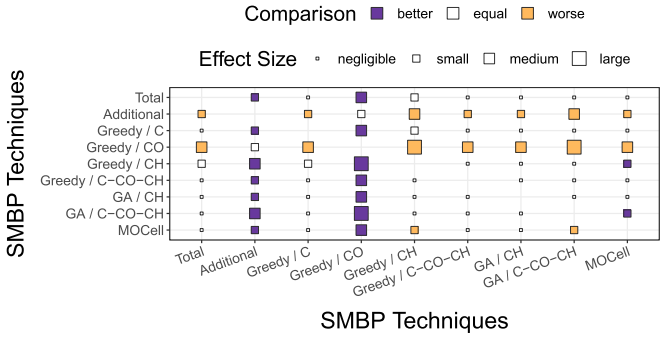

Fig. 5. Overall *APFD-P* effectiveness pairwise comparison among the three best SBSMBP techniques from RQ1a, the greedy baselines, and the *Greedy* techniques across all the versions and repetitions.

2) RQ1b: Comparison to Greedy Techniques: This section compares the SBSMBP techniques to the two greedy baselines (i.e., *Additional* and *Total*) and four *Greedy* techniques, one for each objective *C*, *CO*, and *CH* and the combination *C-CO-CH* (see Section IV-C3). For brevity, we only investigate the three best SBSMBP techniques from RQ1a, i.e., *GA C-CO-CH*, *GA CH*, and *MOCell*. Table III shows the effectiveness results overall and per project.

Overall. From Table III's column "Overall", we observe that the median *APFD-P* ranges from 0.44 to 0.65, where the two greedy baselines achieve 0.60 (for *Total*) and 0.52 (for *Additional*). We already found that *Total* is more effective than *Additional* for SMBP [56], which might come as a surprise to TCP-savvy readers (e.g., see Luo et al. [63]).

A surprising result is that the SBSMBP techniques exhibit either only an equal (*GA C-CO-CH*) or a lower (*GA CH* and *MOCell*) median *APFD-P* compared to *Total*. However, our

TABLE III
APFD-P EFFECTIVENESS FOR THE GREEDY BASELINES, FOUR *GREEDY* TECHNIQUES, AND THREE BEST SBSMBP TECHNIQUES FROM RQ1A; OVERALL AND PER PROJECTS ACROSS ALL THE VERSIONS AND REPETITIONS. THE VALUES ARE THE MEDIAN ± THE MAD

SMBP Algorithm	Objectives	Overall	Projects								
			Byte Buddy	*Eclipse Collections*	*JCTools*	*Jenetics*	*Log4j 2*	*Netty*	*Okio*	*RxJava*	*Xodus*	*Zipkin*
Total	–	0.60±0.13	0.49±0.25	0.60±0.03	0.63±0.13	0.65±0.10	0.65±0.07	0.59±0.09	0.70±0.13	0.58±0.10	0.69±0.04	0.45±0.14
Additional	–	0.52±0.17	0.38±0.16	0.60±0.03	0.50±0.05	0.44±0.26	0.50±0.07	0.51±0.05	0.74±0.13	0.45±0.14	0.61±0.12	0.58±0.20
Greedy	*C*	0.60±0.13	0.49±0.25	0.60±0.04	0.64±0.13	0.65±0.10	0.65±0.07	0.59±0.09	0.71±0.12	0.57±0.10	0.69±0.04	0.45±0.14
	CO	0.44±0.17	0.40±0.15	0.59±0.04	0.39±0.09	0.38±0.16	0.39±0.08	0.42±0.20	0.80±0.14	0.41±0.14	0.38±0.06	0.56±0.14
	CH	0.65±0.15	0.67±0.13	0.67±0.05	0.51±0.09	0.62±0.32	0.65±0.07	0.48±0.24	0.68±0.17	0.64±0.14	0.58±0.16	0.63±0.16
	C-CO-CH	0.61±0.14	0.50±0.28	0.64±0.03	0.65±0.10	0.62±0.14	0.61±0.09	0.60±0.09	0.72±0.15	0.58±0.11	0.72±0.06	0.51±0.18
GA	*CH*	0.58±0.13	0.68±0.14	0.54±0.03	0.59±0.09	0.57±0.28	0.58±0.07	0.54±0.08	0.59±0.18	0.55±0.07	0.59±0.15	0.59±0.14
	C-CO-CH	0.60±0.13	0.68±0.13	0.55±0.03	0.65±0.07	0.59±0.21	0.61±0.09	0.54±0.09	0.68±0.14	0.56±0.08	0.71±0.05	0.55±0.16
MOCell	*C-CO-CH*	0.58±0.12	0.59±0.13	0.56±0.03	0.60±0.07	0.57±0.20	0.56±0.06	0.53±0.07	0.76±0.13	0.58±0.07	0.64±0.08	0.52±0.17

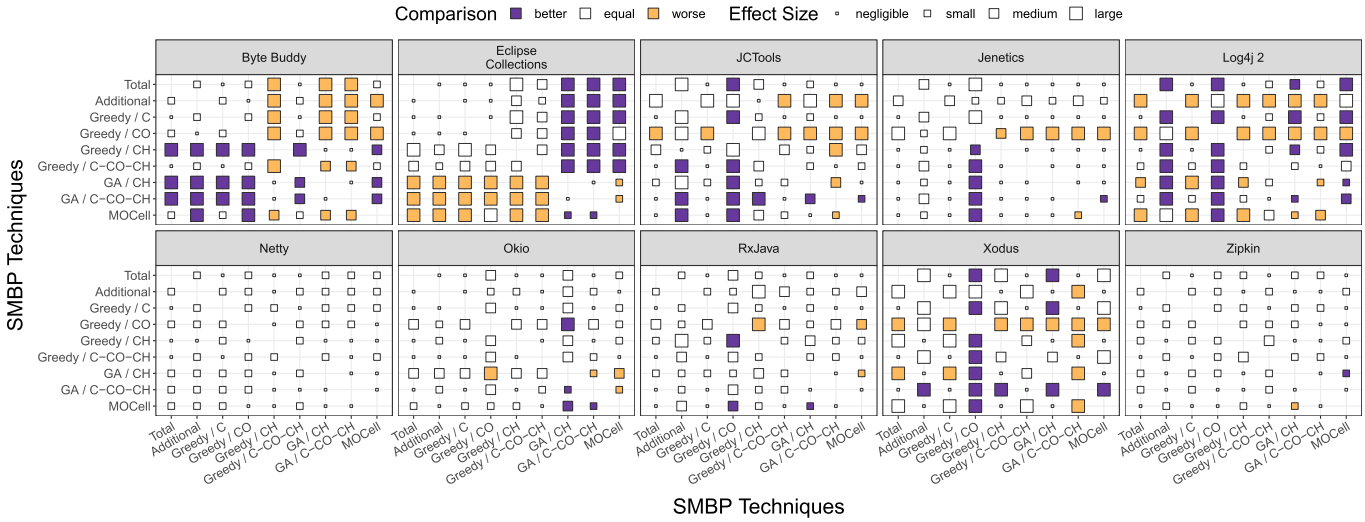

Fig. 6. Per project *APFD-P* effectiveness pairwise comparison among the three best SBSMBP technqiues from RQ1a, the greedy baselines, and the *Greedy* techniques across all the versions and repetitions.

observation aligns with TCP research [63], where the search-based technique is inferior to the greedy baseline (in their case *Additional*). The statistical tests, depicted in Fig. 5, confirm that the SBSMBP techniques are statistically equal to but never worse than *Total* and better than *Additional* (with small and medium effect sizes).

The *Greedy* techniques show a similar pattern as the *GA* techniques in RQ1a: *Greedy CO* is the technique with the lowest median *APFD-P* of 0.44, *Greedy C* is equivalent to *Total* at 0.60, and both *Greedy C-CO-CH* with 0.61 and *Greedy CH* with 0.65 exhibit higher median *APFD-Ps* than *Total*. However, the improvement of the latter two *Greedy* techniques over *Total* is not statistically significant (see Fig. 5). Nevertheless, using *Greedy CH* and achieving a better median effectiveness even though being statistically equivalent is a surprising, yet great result, as it does not require coverage information. RQ2 in Section V-B investigates the implications of this on the technique efficiency.

Per Project. While overall the baseline *Total* is not statistically outperformed by any of the SBSMBP and *Greedy* techniques, the per-project results offer a nuanced view in Table III and Fig. 6.

The best SBSMBP technique, i.e., *GA C-CO-CH* is statistically equivalent to *Total* for eight projects, better for one (*Byte Buddy*), and worse for one (*Eclipse Collections*). The other two SBSMBP techniques, i.e., *GA CH* and *MOCell* perform worse. *GA CH* is statistically better than *Total* for one project (*Byte Buddy*), worse for three, and equivalent for six, whereas *MOCell* is never statistically better, worse for two projects, and equivalent for eight. The median differences are in line: (1) *GA C-CO-CH* is four times better and six times worse than *Total*; (2) *GA CH* is twice better and eight times worse; and (3) *MOCell* is three times better, six times worse, and once equal.

The two best *Greedy* techniques perform favorably compared to *Total*: (1) *Greedy CH* is better for one project (*Byte Buddy*) and equal for nine, while the medians are better for four projects, worse for five, and equal for one; (2) *Greedy C-CO-CH* is equal for all the ten projects, while the medians are better for seven, worse for two, and equal for one.

From a project perspective, the SBSMBP techniques and *Greedy CH* perform better on *Byte Buddy*, while just the SBSMBP techniques perform worse on *Eclipse Collections*. *Byte Buddy* is the project with the smallest benchmark suite, and *Eclipse Collections* the one with the largest. However, this suite size trend is not noticeable for the remaining projects. For *Jenetics*, *Netty*, *Okio*, *RxJava*, and *Zipkin*, it largely does not matter which technique one chooses as long as it is not *Greedy CO* and *GA CH*. For *JCTools*, *Log4j 2*, and *Xodus* any technique

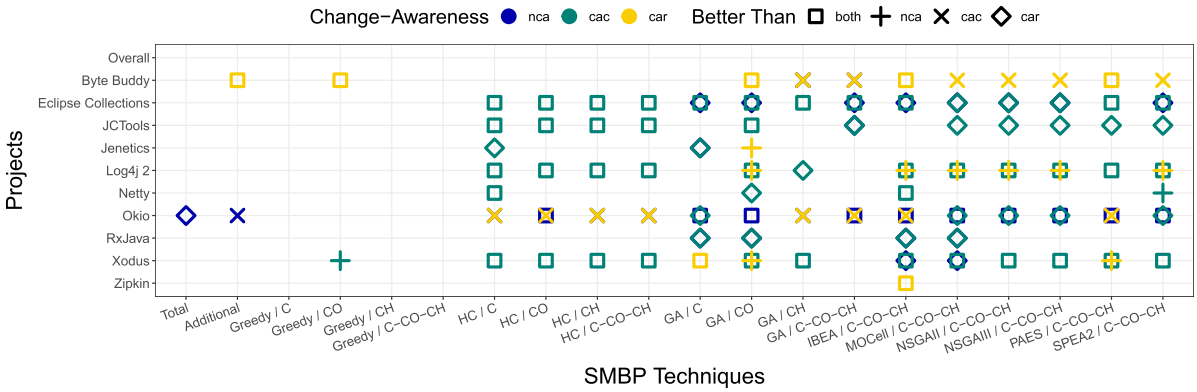

Fig. 7. Per project change-awareness pair-wise comparison for each SMBP technique and project across all the versions and repetitions.

except *Additional*, *Greedy CO*, *GA CH*, and *MOCell* is equally effective.

The strategies with the highest median *APFD-P* for each project are: (1) *Greedy CH* for four, i.e., *Eclipse Collections*, *Log4j 2*, *RxJava*, and *Zipkin*; (2) *Greedy C-CO-CH* for three, i.e., *JCTools*, *Netty*, and *Xodus*; (3) *Total* for two, i.e., *Jenetics* and *Log4j 2*; (4) *GA C-CO-CH* for two, i.e., *Byte Buddy* and *JCTools*; and (5) *GA CH* for one, i.e., *Byte Buddy*.

RQ1b Summary: Only one SBSMBP technique, i.e., *GA C-CO-CH*, is competitive with the best greedy techniques. The baseline *Total* performs surprisingly well, only improved upon by *Greedy CH* for one project and when considering the median *APFD-P*. Considering that *Greedy CH* does not require coverage information makes it a great candidate to generally recommend.

3) RQ1c: Impact of Change-Awareness: This section studies if the SMBP techniques from RQ1a and RQ1b perform differently when incorporating change-awareness. Specifically, we compare *nca* with the two change-aware approaches: *cac* and *car* (see Section IV-C4). Fig. 7 depicts the results of the pair-wise statistical tests displayed per project and SMBP technique. For each project and technique, the figure shows which change-awareness approaches are statistically better than which other approaches.

Overall. Contrary to intuition, i.e., a change-aware approach performs better, change-awareness *does not* improve SMBP effectiveness (see the top row of Fig. 7). To keep the SMBP techniques simple, i.e., not having to manage code change information, this result suggests that *nca* is the best choice overall.

Per Project. From the overall results, we observe differences among the change-aware approaches, although with no best approach across SMBP techniques and projects. Specifically, greedy techniques are less sensitive to change-awareness, i.e., change-awareness has hardly an impact on the effectiveness. Whereas the SBSMBP techniques are considerably impacted by the change-awareness choice, each technique is not equally affected by the same approach. We notice that change-awareness is connected to the project that the SMBP is applied to, e.g., using *cac* for *Byte Buddy*, *car* for *JCTools*, and *nca* for (most SMBP techniques for) *Okio* is most effective. This shows that there often is a benefit for practitioners choosing the "right" change-awareness approach depending on the project and SMBP technique.

RQ1c Summary: Change-awareness does not impact SMBP effectiveness overall. However, depending on the concrete project and SMBP technique, one of the three change-awareness approaches can be more effective, in particular when using a SBSMBP technique.

B. RQ2: Efficiency

While the previous section investigated the effectiveness of the SBSMBP and *Greedy* techniques, a holistic evaluation requires an efficiency evaluation to understand whether using a technique is feasible in practice. This section compares the greedy heuristics *Additional* and *Total*, *Greedy* techniques, and SBSMBP techniques with respect to the overhead they impose on the overall benchmark suite execution time. Table IV shows the median and MAD efficiency overheads overall and per project. Overheads below 1% are depicted as "$< 1\%$". We refrain from reporting statistical tests, as they would arguably neither add valuable insights nor change the conclusions.

Overall. Both baselines add less than 1% for running the algorithm, i.e., prioritization time, and combined with the coverage extraction time have a total runtime overhead, i.e., analysis time, of 17%. This shows that the analysis time is dominated by the coverage extraction and the SMBP algorithm plays only a minor role. These numbers are from our previous work [56]. The same applies to the coverage-based *Greedy* techniques, i.e., *C*, *CO*, and *C-CO-CH*.

The single-objective SBSMBP techniques (i.e., *HC* and *GA*) have around 1% prioritization time. This brings the analysis time to between 17% and 19%, similar to the greedy baselines. *Greedy CH*, *HC CH*, and *GA CH*, however, have a similar analysis and prioritization time. This is because the *CH* objective does not rely on coverage extraction, which takes the majority of the analysis time for coverage-based techniques. In particular, *Greedy CH* becomes (even more) appealing, as it is

TABLE IV
RUNTIME OVERHEAD (PRIORITIZATION AND ANALYSIS TIME) FOR ALL THE PROJECTS AND OVERALL FOR THE SMBP TECHNIQUES

SMBP Algorithm	Objectives	Time	Overall	Projects									
				Byte Buddy	*Eclipse Collections*	*JCTools*	*Jenetics*	*Log4j 2*	*Netty*	*Okio*	*RxJava*	*Xodus*	*Zipkin*
Total	–	Prioritization	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1
		Analysis	17%±<1	13%±<1	104%±<1	19%±<1	8%±<1	22%±<1	49%±<1	14%±<1	23%±<1	16%±<1	20%±<1
Additional	–	Prioritization	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	1%±1	<1%±<1	<1%±<1	<1%±<1	<1%±<1
		Analysis	17%±<1	13%±<1	105%±<1	19%±<1	8%±<1	22%±<1	50%±1	14%±<1	23%±<1	16%±<1	20%±<1
Greedy	*C*	Prioritization	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1
		Analysis	17%±<1	13%±<1	104%±<1	19%±<1	8%±<1	22%±<1	49%±<1	14%±<1	23%±<1	16%±<1	20%±<1
	CO	Prioritization	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1
		Analysis	17%±<1	13%±<1	104%±<1	19%±<1	8%±<1	22%±<1	49%±<1	14%±<1	23%±<1	16%±<1	20%±<1
	CH	Prioritization	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1
		Analysis	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1
	C-CO-CH	Prioritization	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1
		Analysis	17%±<1	13%±<1	104%±<1	19%±<1	8%±<1	22%±<1	49%±<1	14%±<1	23%±<1	16%±<1	20%±<1
HC	*C*	Prioritization	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1
		Analysis	17%±<1	13%±<1	104%±<1	19%±<1	8%±<1	22%±<1	49%±<1	14%±<1	23%±<1	16%±<1	20%±<1
	CO	Prioritization	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1
		Analysis	17%±<1	13%±<1	104%±<1	19%±<1	8%±<1	22%±<1	49%±<1	14%±<1	23%±<1	16%±<1	20%±<1
	CH	Prioritization	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1
		Analysis	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1
	C-CO-CH	Prioritization	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1
		Analysis	17%±<1	13%±<1	104%±<1	19%±<1	8%±<1	22%±<1	49%±<1	14%±<1	23%±<1	16%±<1	20%±<1
GA	*C*	Prioritization	1%±1	2%±1	<1%±<1	<1%±<1	1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	1%±<1
		Analysis	18%±1	15%±1	104%±<1	19%±<1	9%±<1	23%±<1	49%±<1	14%±<1	23%±<1	16%±<1	22%±<1
	CO	Prioritization	1%±1	2%±1	<1%±<1	<1%±<1	1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	1%±<1
		Analysis	19%±1	15%±1	104%±<1	19%±<1	9%±<1	23%±<1	49%±<1	14%±<1	23%±<1	16%±<1	22%±<1
	CH	Prioritization	1%±1	2%±1	<1%±<1	<1%±<1	1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	1%±<1
		Analysis	1%±1	2%±1	<1%±<1	<1%±<1	1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	1%±<1
	C-CO-CH	Prioritization	1%±1	2%±1	<1%±<1	1%±<1	2%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	1%±<1
		Analysis	19%±1	15%±1	104%±<1	20%±<1	9%±<1	23%±<1	49%±<1	14%±<1	23%±<1	16%±<1	23%±<1
IBEA	*C-CO-CH*	Prioritization	1%±2	5%±2	<1%±<1	1%±<1	3%±<1	1%±<1	<1%±<1	1%±<1	<1%±<1	1%±<1	3%±1
		Analysis	20%±2	18%±2	104%±<1	20%±<1	11%±<1	23%±<1	49%±<1	15%±<1	23%±<1	17%±<1	25%±1
MOCell	*C-CO-CH*	Prioritization	<1%±<1	3%±1	<1%±<1	<1%±<1	1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	1%±<1
		Analysis	18%±<1	15%±1	105%±<1	19%±<1	9%±<1	22%±<1	49%±<1	14%±<1	23%±<1	16%±<1	23%±<1
NSGAII	*C-CO-CH*	Prioritization	1%±1	2%±1	<1%±<1	<1%±<1	1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	1%±<1
		Analysis	19%±1	15%±1	104%±<1	20%±<1	9%±<1	23%±<1	49%±<1	14%±<1	23%±<1	16%±<1	22%±<1
NSGAIII	*C-CO-CH*	Prioritization	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1
		Analysis	17%±<1	13%±<1	104%±<1	19%±<1	8%±<1	22%±<1	49%±<1	14%±<1	23%±<1	16%±<1	21%±<1
PAES	*C-CO-CH*	Prioritization	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1	<1%±<1
		Analysis	17%±<1	13%±<1	104%±<1	19%±<1	8%±<1	22%±<1	49%±<1	14%±<1	23%±<1	16%±<1	21%±<1
SPEA2	*C-CO-CH*	Prioritization	2%±2	9%±3	<1%±<1	1%±<1	5%±1	1%±<1	<1%±<1	1%±<1	<1%±<1	1%±<1	3%±1
		Analysis	22%±2	22%±3	105%±<1	20%±<1	13%±1	23%±<1	49%±<1	15%±<1	23%±<1	17%±<1	26%±1

not only among the most effective techniques (with, e.g., *Total*) but also the most efficient one.

Regarding the multi-objective SBSMBP technique, we observe that the overall overheads do not change for the majority of the MOEAs, i.e., *MOCell* (the most effective MOEA from RQ1, see Section V-A), *NSGAII*, *NSGAIII*, and *PAES*. Only *IBEA* and *SPEA2* encounter a slightly larger prioritization time, which increases the analysis time to 19% and 21%, respectively. Given the large suite runtimes (see Table I), this slight increase over the greedy techniques can be considered acceptable.

Per Project. The results per project show a diverse situation for the SMBP techniques that rely on coverage objectives: the analysis times range from between 8% and 13% for *Jenetics* to between 104% and 105% for *Eclipse Collections*. For eight of the projects, the analysis time is 23% or less. For *Eclipse Collections* and *Netty*, where the overhead is 105% and 49% respectively, applying SMBP is not worthwhile.

These project-dependent overheads suggest that the techniques with non-coverage-based objectives are critical to be universally applicable across projects, especially when run as part of CI. Hence, the overhead results for the *CH* techniques are especially promising: all *CH* techniques have an analysis time overhead of 2% or less across all the projects. When considering the most effective technique *Greedy CH*, the overhead even drops below 1% for all the projects. Consequently, the analysis time of the techniques with non-coverage-based objectives is solely dependent on the prioritization time (i.e., the time taken to run the SMBP algorithm), which will likely always be considerably smaller than the extensive benchmark suite runtimes; therefore, it is generally beneficial to apply these SMBP techniques.

RQ2 Summary: The SBSMBP techniques impose only a minor additional overhead compared to the greedy baselines. Employing a SMBP technique that only relies on the *CH* objective gives the lowest and most reliable overhead across all the projects. In particular, *Greedy CH* consistently has less than 1% overhead.

VI. Discussion

The results show that the best SBSMBP technique (*GA C-CO-CH*) is only competitive with the best greedy baseline (*Total*). When using a simple *Greedy* technique that only considers the historical performance change size (*CH*), one can retrieve equally and sometimes more effective SMBP rankings with significantly lower overhead than when relying on coverage objectives (*C* and *CO*), which is the case for both the greedy baselines and the SBSMBP techniques. This section discusses our empirical results from several different perspectives.

The Underperforming Search-Based Techniques. The success of the *Greedy CH* technique in terms of effectiveness begs the question, why are the SBSMBP techniques underperforming in our study? We see the following two reasons.

First, coverage is not "good enough" as objectives for SMBP. This becomes evident from this and our previous study [56]. Furthermore, Chen et al. [11] observed that in the context of benchmark selection, functional bug prediction metrics are more important than code-level performance metrics. This suggests that future SMBP techniques should explore dedicated metrics related to size, diffusion, or history as well as specific code-level performance metrics, such as loops or synchronization (similar to Laaber et al. [55]). However, given the high coverage extraction overhead, non-code metrics should be more favoured over code metrics.

Second, optimal hyperparameters of the SAs could improve SBSMBP effectiveness. For this, we explored the SAs' effectiveness with more generations, investigated the convergence of the objectives, and found that the objectives converge quickly. This suggests that the search space is relatively trivial (for the given objectives) and SAs do not offer an advantage over simple greedy techniques, which our results empirically show. Moreover, the employed search objectives, in particular the coverage-based objectives, are ineffective in finding larger performance changes sooner, and other objectives should be explored in the future (such as the ones mentioned above).

Is it Worth Applying SMBP? While coverage-based SMBP is effective, there is still a considerable overhead. This overhead is mostly due to the time required to extract coverage information [56] and only a small fraction is attributed to the prioritization algorithm (see Section V-B). While this time is arguably lower than for unit testing because benchmarks are repeatedly executed and run much longer [54], whether it is worth employing coverage-based SMBP highly depends on the individual benchmark suite and the project's performance testing objectives. If it is critical to detect performance changes as fast as possible, e.g., the release of a new software version would be otherwise halted, running SMBP can drastically reduce the time to detect these. However, for projects with extensive overheads, such as *Eclipse Collections* and *Netty* in our study, coverage-based SMBP is not worthwhile. A better alternative to coverage-based techniques are SMBP techniques that solely rely on the *CH* objective, which is (sometimes) more effective and substantially more efficient. This suggests that non-coverage-based SMBP techniques are highly suggested to be employed.

Is it Practical to Apply SMBP? Beyond the temporal cost of applying a SMBP technique, there is a cost of extracting and maintaining the information required as input. On the one hand, the techniques require storing historical information about benchmark executions, especially their changes between previous versions, for computing the *CH* objective. On the other hand, there is neither a need to store historical coverage information nor compute the source code difference on every new commit, as the results of RQ1c show that *nca*, the non-change-aware approach, performs equally to the change-aware approaches (i.e., *cac* and *car*) overall. This is probably best done as part of the CI pipeline because it provides the necessary infrastructure to run benchmarks, store the information about the build (i.e., performance change history), and has the code changes readily available.

Relying on Measured Performance Changes. Research on functional TCP often uses datasets with seeded, artificial faults (mutations) to evaluate TCP effectiveness (e.g., [22], [64]). Conversely, performance regression testing research mostly relies on measured performance changes between adjacent versions [56], [67]. This is due to performance mutation testing having only recently received attention [20], [42] and large datasets, such as Defects4J [44], do not exist. Because of the enormous experiment runtimes required for rigorously executing benchmarks (this study's dataset took 89 days to create), performing performance mutation testing, where for each mutant the whole benchmark suite has to be executed, becomes unrealistic. However, this does not invalidate the findings of this study or any other experimental performance regression study relying on measured performance changes as long as the measurements have been conducted rigorously.

Moreover, measuring performance captures changes stemming from "outside" the software under test (SUT), e.g., its dependencies or runtime environment. While it is harder for techniques to also detect these changes, it makes the evaluation more realistic, as the SUT's developers should be aware of any observable performance changes, irrespective of their origin, to take adequate action.

What is an Important Performance Change? Following previous research on SMBP [56], [67], this paper also defines the importance of a benchmark as the performance change size it detects. A benchmark that detects a large performance change is considered more important than a benchmark that detects a small one. Accordingly, we conclude whether a technique is more or less effective. It is, however, unclear whether this definition of importance is "accurate." Alternative formulations of importance could be based on: (1) the impact of the code called by the benchmark on the performance of an application, (2) the project developer's perception and context-dependent knowledge, or (3) the usage frequency of the code from application programming interface (API) clients. Moreover, developers might consider any performance change as important, irrespective of the size. All these different definitions of impor-

tance would likely change our results and require a study of its own.

VII. Related Work

This section discusses the related work on (1) search-based regression testing and (2) software performance testing.

A. Search-Based Regression Testing

Regression testing on unit tests has been a well-established research field for the last 30 years [81]. The three main techniques in regression testing are test suite minimization, RTS, and TCP. Traditionally, techniques relied on greedy heuristics, such as *Additional* and *Total* coverage, to solve the optimization problems of selecting the best subset or prioritizing the most fault-revealing tests [24], [72], [73].

Based on the early success of search techniques in software engineering [36], regression testing techniques were quickly adapted to use search for the optimization. Yoo and Harman [82] formulate RTS as a multi-objective optimization problem and generate Pareto optimal solutions [28]. Li et al. [60] are the first to apply search to TCP by introducing coverage-based objectives based on block, decision, and statement granularity. Our work is inspired by theirs and adapts the objectives for SMBP.

Li et al. [61] take the idea a step forward and introduce multi-objective, search-based TCP with *NSGAII* and two objectives: coverage and execution time. Islam et al. [41] and Marchetto et al. [65] consider three objectives and apply different weights to these. Epitropakis et al. [27] address the $\mathcal{O}(mn)$ complexity of the fitness function by devising a coverage compaction algorithm and consider historical faults as an objective. Finally, Di Nucci et al. [22] introduce a new search algorithm based on the hypervolume [7]. Our work builds on all these in terms of inspiration for the search objectives and the experimental setup (see Sectio IV-C) and ports them to the problem of prioritizing microbenchmarks.

B. Software Performance Testing

Performance testing with microbenchmarks is a relatively new area of research. To this end, earlier studies investigate unit-level performance testing of OSS Java projects [59], [76], which conclude that it is still a niche technique, especially compared to unit testing. Explanations could be that more effort is required to write microbenchmarks, the extra cost of their execution, and the higher complexity required to assess the results with statistical analyses. Samoaa and Leitner [75] investigate parameterization of benchmarks in detail. Grambow et al. [31], [32] utilize information from application benchmarks to guide microbenchmark execution and assess the ability of microbenchmarks to detect application performance changes, with limited success. He et al. [37] and Laaber et al. [54] take an orthogonal approach to optimizing performance testing: they dynamically stop benchmarks when their results are of a desired statistical quality instead of optimizing the execution order of the benchmark suite. Traini et al. [79] further assess whether dynamically stopping is beneficially for measuring in steady state.

There exist works on performance regression testing, i.e., greedy heuristics [17], genetic algorithms [3], [4], and machine learning [11], which have been used to select the optimal benchmarks for a given software version in terms of their ability to find performance changes. These works focus on RTS whereas this paper targets SMBP. In addition, some works exist for performance regression testing of concurrent classes (e.g., [70], [83]). They focus on and generate tests for concurrent software, whereas this paper optimizes the execution of existing benchmark suites. Finally, Huang et al. [40] proposed a prediction method to assess the need for performance testing of a new software version, which targets the version level compared to ours on the benchmark level.

We consider two works on performance regression testing closely related to ours [56], [67]. For a given version, Mostafa et al. [67] prioritize test cases with complex performance impact analyses that require measuring individual components of not only the SUT but also the Java Development Kit (JDK). Their technique focuses on collection-intensive software and is partially only evaluated on unit tests used as performance tests, whereas our technique is generally applicable to any kind of software and is evaluated only on benchmarks. Laaber et al. [56] studied coverage-based, greedy SMBP techniques and found that they are only marginally more effective than a random ordering. We build on their work and use SAs and novel *Greedy* techniques instead.

VIII. Conclusion and Future Work

This paper defines SBSMBP and *Greedy* SMBP techniques relying on three objectives: coverage, coverage overlap among benchmarks, and performance change size obtained from historical data. With an extensive experimental evaluation on 10 Java projects with *JMH* suites, we study the effectiveness and efficiency of a total of 18 new SMBP techniques and compare these to two coverage-based, greedy SMBP baselines. The results reveal that the best SBSMBP technique is competitive with the best greedy baselines but does not improve on its effectiveness. Surprisingly, the *Greedy* technique relying only on the historical performance change size is sometimes more but at least equally effective as the best coverage-based, greedy baselines and SBSMBP techniques while being significantly more efficient.

This paper provides evidence that SMBP is a difficult problem to solve and more work is required going forward. Hence, we envision future research to (1) assess SMBP on performance mutations and developer-reported performance bugs, (2) investigate novel algorithms (potentially search-based) specifically targeting SMBP, and (3) devise process- and performance-focused, ideally non-coverage-based, objectives that serve as better proxies for performance changes than the ones studied here.

REFERENCES

[1] P. Achimugu, A. Selamat, R. Ibrahim, and M. N. Mahrin, "A systematic literature review of software requirements prioritization research," *Inf. Softw. Technol.*, vol. 56, no. 6, pp. 568–585, 2014, doi: 10.1016/j.infsof.2014.02.001.

[2] N. Alshahwan, M. Harman, and A. Marginean, "Software testing research challenges: An industrial perspective," in *Proc. 16th IEEE Int. Conf. Softw. Testing, Verification Validation (ICST)*, Piscataway, NJ, USA: IEEE Press, 2023, pp. 1–10, doi: 10.1109/ICST57152.2023.00008.

[3] D. Alshoaibi, K. Hannigan, H. Gupta, and M. W. Mkaouer, "PRICE: Detection of performance regression introducing code changes using static and dynamic metrics," in *Proc. 11th Int. Symp. Search Based Softw. Eng. (SSBSE)*, New York, NY, USA: Springer-Verlag, 2019, pp. 75–88, doi: 10.1007/978-3-030-27455-9_6.

[4] D. Alshoaibi, M. W. Mkaouer, A. Ouni, A. Wahaishi, T. Desell, and M. Soui, "Search-based detection of code changes introducing performance regression," *Swarm Evol. Comput.*, vol. 73, p. 101101, Aug. 2022, doi: 10.1016/j.swevo.2022.101101.

[5] A. Arcuri, "RESTful API automated test case generation with EvoMaster," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 1, pp. 1–37, 2019, doi: 10.1145/3293455.

[6] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Proc. 33rd Int. Conf. Softw. Eng. (ICSE)*, New York, NY, USA: ACM, 2011, doi: 10.1145/1985793.1985795.

[7] A. Auger, J. Bader, D. Brockhoff, and E. Zitzler, "Theory of the hypervolume indicator: Optimal μ-distributions and the choice of the reference point," in *Proc. 10th ACM SIGEVO Workshop Found. Genetic Algorithms (FOGA)*, vol. 4, New York, NY, USA: ACM, 2009, pp. 1165–1188, doi: 10.1145/1527125.1527138.

[8] Y. Benjamini and D. Yekutieli, "The control of the false discovery rate in multiple testing under dependency," *Ann. Statist.*, vol. 29, no. 4, 2001, doi: 10.1214/aos/1013699998.

[9] J. Branke, K. Deb, H. Dierolf, and M. Osswald, "Finding knees in multi-objective optimization," in *Proc. 8th Int. Conf. Parallel Problem Solving Nature (PPSN)*, New York, NY, USA: Springer-Verlag, 2004, pp. 722–731, doi: 10.1007/978-3-540-30217-9_73.

[10] J. Chen and W. Shang, "An exploratory study of performance regression introducing code changes," in *Proc. 33rd IEEE Int. Conf. Softw. Maintenance Evolution (ICSME),* Piscataway, NJ, USA: IEEE Press, 2017, pp. 341–352, doi: 10.1109/icsme.2017.13.

[11] J. Chen, W. Shang, and E. Shihab, "PerfJIT: Test-level just-in-time prediction for performance regression introducing commits," *IEEE Trans. Softw. Eng.*, vol. 48, no. 5, pp. 1529–1544, May 2022, doi: 10.1109/TSE.2020.3023955.

[12] K. Chen, Y. Li, Y. Chen, C. Fan, Z. Hu, and W. Yang, "GLIB: Towards automated test oracle for graphically-rich applications," in *Proc. 29th ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*, New York, NY, USA: ACM, 2021, pp. 1093–1104, doi: 10.1145/3468264.3468586.

[13] D. Daly, "Creating a virtuous cycle in performance testing at MongoDB," in *Proc. 12th ACM/SPEC Int. Conf. Perform. Eng. (ICPE)*, New York, NY, USA: ACM, 2021, doi: 10.1145/3427921.3450234.

[14] D. Daly, W. Brown, H. Ingo, J. O'Leary, and D. Bradford, "The use of change point detection to identify software performance regressions in a continuous integration system," in *Proc. 11th ACM/SPEC Int. Conf. Perform. Eng. (ICPE)*, New York, NY, USA: ACM, 2020, doi: 10.1145/3358960.3375791.

[15] D. E. Damasceno Costa, C.-P. Bezemer, P. Leitner, and A. Andrzejak, "What's wrong with my benchmark results? Studying bad practices in JMH benchmarks," *IEEE Trans. Softw. Eng.*, vol. 47, no. 7, pp. 1452–1467, Jul. 2021, doi: 10.1109/TSE.2019.2925345.

[16] A. C. Davison and D. Hinkley, "Bootstrap methods and their application," vol. 94, Cambridge, MA, USA: Cambridge University Press, 1997, doi: 10.1017/CBO9780511802843.

[17] A. B. de Oliveira, S. Fischmeister, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Perphecy: Performance regression test selection made simple but effective," in *Proc. 10th IEEE Int. Conf. Softw. Testing, Verification Validation (ICST)*, 2017, pp. 103–113, doi: 10.1109/ICST.2017.17.

[18] K. Deb and H. Jain, "An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part I: Solving problems with box constraints," *IEEE Trans. Evol. Computation*, vol. 18, no. 4, pp. 577–601, Aug. 2014, doi: 10.1109/TEVC.2013.2281535.

[19] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Trans. Evol. Comput.*, vol. 6, no. 2, pp. 182–197, Apr. 2002, doi: 10.1109/4235.996017.

[20] P. Delgado-Pérez, A. B. Sánchez, S. Segura, and I. Medina-Bulo, "Performance mutation testing," *Softw. Testing, Verification Rel.*, vol. 31, no. 5, e1728, 2020, doi: 10.1002/stvr.1728.

[21] X. Devroey, A. Gambi, J. P. Galeotti, R. Just, F. M. Kifetew, A. Panichella, and S. Panichella, "JUGE: An infrastructure for benchmarking Java unit test generators," *Softw. Testing, Verification Rel.*, vol. 33, no. 3, e1838, 2023, doi: 10.1002/stvr.1838.

[22] D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia, "A test case prioritization genetic algorithm guided by the hypervolume indicator," *IEEE Trans. Softw. Eng.*, vol. 46, no. 6, pp. 674–696, Jun. 2020, doi: 10.1109/tse.2018.2868082.

[23] O. J. Dunn, "Multiple comparisons using rank sums," *Technometrics*, vol. 6, no. 3, pp. 241–252, 1964, doi: 10.1080/00401706.1964.10490181.

[24] S. Elbaum, A. Malishevsky, and G. Rothermel, "Incorporating varying test costs and fault severities into test case prioritization," in *Proc. 23rd Int. Conf. Softw. Eng. (ICSE)*, Piscataway, NJ, USA: IEEE Press, 2001, pp. 329–338, doi: 10.1109/icse.2001.919106.

[25] S. Elbaum, G. Rothermel, and J. Penix, "Techniques for improving regression testing in continuous integration development environments," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng. (FSE)*, New York, NY, USA: ACM, 2014, pp. 235–245, doi: 10.1145/2635868.2635910.

[26] D. Elsner, F. Hauer, A. Pretschner, and S. Reimer, "Empirically evaluating readily available information for regression test optimization in continuous integration," in *Proc. 30th ACM SIGSOFT Int. Symp. Softw. Testing Anal. (ISSTA)*, New York, NY, USA: ACM, 2021, pp. 491–504, doi: 10.1145/3460319.3464834.

[27] M. G. Epitropakis, S. Yoo, M. Harman, and E. K. Burke, "Empirical evaluation of pareto efficient multi-objective regression test case prioritisation," in *Proc. Int. Symp. Softw. Testing Anal. (ISSTA)*, New York, NY, USA: ACM, 2015, pp. 234–245, doi: 10.1145/2771783.2771788.

[28] C. M. Fonseca and P. J. Fleming, "An overview of evolutionary algorithms in multiobjective optimization," *Evol. Comput.*, vol. 3, no. 1, pp. 1–16, 1995, doi: 10.1162/evco.1995.3.1.1.

[29] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," in *Proc. 19th Int. Symp. Softw. Testing Anal. (ISSTA)*, New York, NY, USA: ACM, 2010, pp. 147–158, doi: 10.1145/1831708.1831728.

[30] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous Java performance evaluation," in *Proc. 22nd ACM SIGPLAN Conf. Object-Oriented Program., Syst., Appl. (OOPSLA)*, New York, NY, USA: ACM, 2007, pp. 57–76, doi: 10.1145/1297027.1297033.

[31] M. Grambow, C. Laaber, P. Leitner, and D. Bermbach, "Using application benchmark call graphs to quantify and improve the practical relevance of microbenchmark suites," *PeerJ Comput. Sci.*, vol. 7, pp. 1–32, 2021, doi: 10.7717/peerj-cs.548.

[32] M. Grambow, D. Kovalev, C. Laaber, P. Leitner, and D. Bermbach, "Using microbenchmark suites to detect application performance changes," *IEEE Trans. Cloud Comput.*, vol. 11, no. 3, pp. 2575–2590, Jul./Sep. 2023, doi: 10.1109/TCC.2022.3217947.

[33] A. Haghighatkhah, M. Mäntylä, M. Oivo, and P. Kuvaja, "Test prioritization in continuous integration environments," *J. Syst. Softw.*, vol. 146, pp. 80–98, Dec. 2018, doi: 10.1016/j.jss.2018.08.061.

[34] D. Hao, L. Zhang, L. Zhang, G. Rothermel, and H. Mei, "A unified test case prioritization approach," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 2, pp. 1–31, 2014, doi: 10.1145/2685614.

[35] M. Harman, "The current state and future of search based software engineering," in *Proc. Future Softw. Eng. (FOSE)*, Piscataway, NJ, USA: IEEE Press, 2007, pp. 342–357, doi: 10.1109/fose.2007.29.

[36] M. Harman and B. F. Jones, "Search-based software engineering," *Inf. Softw. Technol.*, vol. 43, no. 14, pp. 833–839, 2001, doi: 10.1016/S0950-5849(01)00189-6.

[37] S. He, G. Manns, J. Saunders, W. Wang, L. Pollock, and M. L. Soffa, "A statistics-based performance testing methodology for cloud applications," in *Proc. 27th ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*, New York, NY, USA: ACM, 2019, pp. 188–199, doi: 10.1145/3338906.3338912.

[38] M. R. Hess and J. D. Kromrey, "Robust confidence intervals for effect sizes: A comparative study of Cohen's d and Cliff's delta under non-normality and heterogeneous variances," in *Proc. Annu. Meeting Amer. Educ. Res. Assoc.*, 2004.

[39] T. C. Hesterberg, "What teachers should know about the bootstrap: Resampling in the undergraduate statistics curriculum," *Amer. Statist.*, vol. 69, no. 4, pp. 371–386, 2015, doi: 10.1080/00031305.2015.1089789.

[40] P. Huang, X. Ma, D. Shen, and Y. Zhou, "Performance regression testing target prioritization via performance risk analysis," in *Proc. 36th IEEE/ACM Int. Conf. Softw. Eng. (ICSE)*, New York, NY, USA: ACM, 2014, pp. 60–71, doi: 10.1145/2568225.2568232.

[41] M. M. Islam, A. Marchetto, A. Susi, and G. Scanniello, "A multi-objective technique to prioritize test cases based on latent semantic indexing," in *Proc. 16th Eur. Conf. Softw. Maintenance Reeng. (CSMR)*, Piscataway, NJ, USA: IEEE Press, 2012, pp. 21–30, doi: 10.1109/csmr.2012.13.

[42] M. Jangali, Y. Tang, N. Alexandersson, P. Leitner, J. Yang, and W. Shang, "Automated generation and evaluation of JMH microbenchmark suites from unit tests," *IEEE Trans. Softw. Eng.*, vol. 49, no. 4, pp. 1704–1725, Apr. 2023, doi: 10.1109/TSE.2022.3188005.

[43] Z. M. Jiang and A. E. Hassan, "A survey on load testing of large-scale software systems," *IEEE Trans. Softw. Eng.*, vol. 41, no. 11, pp. 1091–1118, Nov. 2015, doi: 10.1109/tse.2015.2445340.

[44] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in *Proc. Int. Symp. Softw. Testing Anal. (ISSTA)*, New York, NY, USA: ACM, 2014, pp. 437–440, doi: 10.1145/2610384.2628055.

[45] T. Kalibera and R. Jones, "Quantifying performance changes with effect size confidence intervals," Univ. Kent, Kent, U.K., Tech. Rep. 4–12, 2012. Accessed: Nov. 10, 2022. [Online]. Available: http://www.cs.kent.ac.uk/pubs/2012/3233

[46] T. Kalibera and R. Jones, "Rigorous benchmarking in reasonable time," in *Proc. ACM SIGPLAN Int. Symp. Memory Manage. (ISMM)*, New York, NY, USA: ACM, 2013, doi: 10.1145/2464157.2464160.

[47] M. Kim, T. Hiroyasu, M. Miki, and S. Watanabe, "SPEA2+: Improving the performance of the strength pareto evolutionary algorithm 2," in *Proc. 8th Int. Conf. Parallel Problem Solving Nature (PPSN)*, vol. 3242, New York, NY, USA: Springer-Verlag, 2004, doi: 10.1007/978-3-540-30217-9_75.

[48] J. D. Knowles and D. Corne, "The Pareto archived evolution strategy: A new baseline algorithm for Pareto Multiobjective optimisation," in *Proc. Congr. Evol. Comput. (CEC)*, Piscataway, NJ, USA: IEEE Press, 1999, doi: 10.1109/cec.1999.781913.

[49] W. H. Kruskal and W. A. Wallis, "Use of ranks in one-criterion variance analysis," *J. Amer. Statist. Assoc.*, vol. 47, no. 260, pp. 583–621, 1952, doi: 10.1080/01621459.1952.10483441.

[50] C. Laaber, "chrstphlbr/pa: v0.1.0," Nov. 2022, doi: 10.5281/zenodo.7308066.

[51] C. Laaber and P. Leitner, "An evaluation of open-source software microbenchmark suites for continuous performance assessment," in *Proc. 15th Int. Conf. Mining Softw. Repositories (MSR)*, New York, NY, USA: ACM, 2018, pp. 119–130, doi: 10.1145/3196398.3196407.

[52] C. Laaber and S. Würsten, "chrstphlbr/bencher: Release v0.4.0," Jan. 2024, doi: 10.5281/zenodo.10527360.

[53] C. Laaber, J. Scheuner, and P. Leitner, "Software microbenchmarking in the cloud. How bad is it really?" *Empirical Softw. Eng.*, vol. 24, pp. 2469–2508, Aug. 2019, doi: 10.1007/s10664-019-09681-1.

[54] C. Laaber, S. Würsten, H. C. Gall, and P. Leitner, "Dynamically reconfiguring software microbenchmarks: Reducing execution time without sacrificing result quality," in *Proc. 28th ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*, New York, NY, USA: ACM, 2020, pp. 989–1001, doi: 10.1145/3368089.3409683.

[55] C. Laaber, M. Basmaci, and P. Salza, "Predicting unstable software benchmarks using static source code features," *Empirical Softw. Eng.*, vol. 26, no. 6, pp. 1–53, 2021, doi: 10.1007/s10664-021-09996-y.

[56] C. Laaber, H. C. Gall, and P. Leitner, "Applying test case prioritization to software microbenchmarks," *Empirical Softw. Eng.*, vol. 26, no. 6, pp. 1–48, 2021, doi: 10.1007/s10664-021-10037-x.

[57] C. Laaber, H. C. Gall, and P. Leitner, "Replication package "Applying test case prioritization to software microbenchmarks"," 2021, doi: 10.5281/zenodo.5206117.

[58] C. Laaber, T. Yue, and S. Ali, "Replication package "Evaluating search-based software microbenchmark prioritization"," 2024, doi: 10.5281/zenodo.10527125.

[59] P. Leitner and C.-P. Bezemer, "An exploratory study of the state of practice of performance testing in Java-based open source projects," in *Proc. 8th ACM/SPEC Int. Conf. Perform. Eng. (ICPE)*, New York, NY, USA: ACM, 2017, pp. 373–384, doi: 10.1145/3030207.3030213.

[60] Z. Li, M. Harman, and R. M. Hierons, "Search algorithms for regression test case prioritization," *IEEE Trans. Softw. Eng.*, vol. 33, no. 4, pp. 225–237, Apr. 2007, doi: 10.1109/TSE.2007.38.

[61] Z. Li, Y. Bian, R. Zhao, and J. Cheng, "A fine-grained parallel multi-objective test case prioritization on GPU," in *Proc. 5th Symp. Search Based Softw. Eng. (SSBSE)*, New York, NY, USA: Springer-Verlag, 2013, pp. 111–125, doi: 10.1007/978-3-642-39742-4_10.

[62] J. Liang, S. Elbaum, and G. Rothermel, "Redefining prioritization: Continuous prioritization for continuous integration," in *Proc. 40th IEEE/ACM Int. Conf. Softw. Eng. (ICSE)*, New York, NY, USA: ACM, 2018, pp. 688–698, doi: 10.1145/3180155.3180213.

[63] Q. Luo, K. Moran, and D. Poshyvanyk, "A large-scale empirical comparison of static and dynamic test case prioritization techniques," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng. (FSE)*, New York, NY, USA: ACM, 2016, pp. 559–570, doi: 10.1145/2950290.2950344.

[64] Q. Luo, K. Moran, L. Zhang, and D. Poshyvanyk, "How do static and dynamic test case prioritization techniques perform on modern software systems? An extensive study on GitHub projects," *IEEE Trans. Softw. Eng.*, vol. 45, no. 11, pp. 1054–1080, Nov. 2019, doi: 10.1109/tse.2018.2822270.

[65] A. Marchetto, M. M. Islam, W. Asghar, A. Susi, and G. Scanniello, "A multi-objective technique to prioritize test cases," *IEEE Trans. Softw. Eng.*, vol. 42, no. 10, pp. 918–940, Oct. 2016, doi: 10.1109/tse.2015.2510633.

[66] A. Maricq, D. Duplyakin, I. Jimenez, C. Maltzahn, R. Stutsman, and R. Ricci, "Taming performance variability," in *Proc. 13th USENIX Conf. Operating Syst. Des. Implementation (OSDI)*, Carlsbad, CA, USA: USENIX Association, 2018, pp. 409–425. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/maricq

[67] S. Mostafa, X. Wang, and T. Xie, "PerfRanker: Prioritization of performance regression tests for collection-intensive software," in *Proc. 26th ACM SIGSOFT Int. Symp. Softw. Testing Anal. (ISSTA)*, New York, NY, USA: ACM, 2017, pp. 23–34, doi: 10.1145/3092703.3092725.

[68] A. J. Nebro, J. J. Durillo, F. Luna, B. Dorronsoro, and E. Alba, "MOCell: A cellular genetic algorithm for multiobjective optimization," *Int. J. Intell. Syst.*, vol. 24, no. 7, pp. 726–746, 2009, doi: 10.1002/int.20358.

[69] A. J. Nebro, J. Pérez-Abad, J. F. Aldana-Martin, and J. García-Nieto, *Evolving a Multi-Objective Optimization Framework*. New York, NY, USA: Springer-Verlag, 2021, pp. 175–198, doi: 10.1007/978-981-16-0662-5_9.

[70] M. Pradel, M. Huggler, and T. R. Gross, "Performance regression testing of concurrent classes," in *Proc. Int. Symp. Softw. Testing Anal. (ISSTA)*, New York, NY, USA: ACM, 2014, pp. 13–25, doi: 10.1145/2610384.2610393.

[71] S. Ren, H. Lai, W. Tong, M. Aminzadeh, X. Hou, and S. Lai, "Non-parametric bootstrapping for hierarchical data," *J. Appl. Statist.*, vol. 37, no. 9, 2010, pp. 1487–1498, doi: 10.1080/02664760903046102.

[72] G. Rothermel and M. J. Harrold, "Empirical studies of a safe regression test selection technique," *IEEE Trans. Softw. Eng.*, vol. 24, no. 6, pp. 401–419, Jun. 1998, doi: 10.1109/32.689399.

[73] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test case prioritization: An empirical study," in *Proc. IEEE Int. Conf. Softw. Maintenance (ICSM)*, Piscataway, NJ, USA: IEEE Press, 1999, pp. 179–188, doi: 10.1109/icsm.1999.792604.

[74] G. Rothermel, R. J. Untch, and C. Chu, "Prioritizing test cases for regression testing," *IEEE Trans. Softw. Eng.*, vol. 27, no. 10, pp. 929–948, Oct. 2001, doi: 10.1109/32.962562.

[75] H. Samoaa and P. Leitner, "An exploratory study of the impact of parameterization on JMH measurement results in open-source projects," in *Proc. 12th ACM/SPEC Int. Conf. Perform. Eng. (ICPE)*, New York, NY, USA: ACM, 2021, pp. 213–224, doi: 10.1145/3427921.3450243.

[76] P. Stefan, V. Horký, L. Bulej, and P. Tuma, "Unit testing performance in Java projects: Are we there yet?" in *Proc. 8th ACM/SPEC Int. Conf. Perform. Eng. (ICPE)*, New York, NY, USA: ACM, 2017, pp. 401–412, doi: 10.1145/3030207.3030226.

[77] K.-J. Stol and B. Fitzgerald, "The ABC of software engineering research," *ACM Trans. Softw. Eng. Methodol.*, vol. 27, no. 3, pp. 1–51, 2018, doi: 10.1145/3241743.

[78] L. Traini et al., "How software refactoring impacts execution time," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 2, pp. 1–23, 2022, doi: 10.1145/3485136.

[79] L. Traini, V. Cortellessa, D. Di Pompeo, and M. Tucci, "Towards effective assessment of steady state performance in Java software: Are we there yet?" *Empirical Softw. Eng.*, vol. 28, no. 13, pp. 1–57, 2023, doi: 10.1007/s10664-022-10247-x.

[80] A. Vargha and H. D. Delaney, "A critique and improvement of the "CL" common language effect size statistics of McGraw and Wong," *J. Educ. Behav. Statist.*, vol. 25, no. 2, pp. 101–132, 2000, doi: 10.2307/1165329.

[81] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Software: Testing*, *Verification Rel.*, vol. 22, no. 2, pp. 67–120, 2012, doi: 10.1002/stvr.430.

[82] S. Yoo and M. Harman, "Pareto efficient multi-objective test case selection," in *Proc. Int. Symp. Softw. Testing Anal. (ISSTA)*, New York, NY, USA: ACM, 2007, pp. 140–150, doi: 10.1145/1273463.1273483.

[83] T. Yu and M. Pradel, "Pinpointing and repairing performance bottlenecks in concurrent programs," *Empirical Softw. Eng.*, vol. 23, no. 5, pp. 3034–3071, 2018, doi: 10.1007/s10664-017-9578-1.

[84] H. Zhang, M. Zhang, T. Yue, S. Ali, and Y. Li, "Uncertainty-wise requirements prioritization with search," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 1, pp. 1–54, 2021, doi: 10.1145/3408301.

[85] E. Zitzler and S. Künzli, "Indicator-based selection in multiobjective search," in *Proc. 8th Int. Conf. Parallel Problem Solving Nature (PPSN)*, vol. 3242, New York, NY, USA: Springer-Verlag, 2004, pp. 832–842, doi: 10.1007/978-3-540-30217-9_84.

[86] E. Zitzler, M. Laumanns, and L. Thiele, "SPEA2: Improving the strength pareto evolutionary algorithm," ETH Zurich, Comput. Eng. Netw. Lab., Zürich, Switzerland, TIK Rep. 103, 2001, doi: 10.3929/ETHZ-A-004284029.