Impact of JVM Configurations on Test Runtime

Abdelrahman Baz
The University of Texas at Austin
Austin, TX, USA
ambaz@utexas.edu

Milos Gligoric
The University of Texas at Austin
Austin, TX, USA
gligoric@utexas.edu

August Shi
The University of Texas at Austin
Austin, TX, USA
august@utexas.edu

***Abstract*—JVM provides dozens of configuration flags, with many flags intended for tuning application performance. We empirically study the impact of JVM configuration flags on software testing runtime. We focus on an extensive study that shows not only the great impact of JVM configurations on test runtime (up to 43.89% reduction in runtime when using certain configurations) but also shows that those configurations that reduce runtime are rare and thus hard to find. Modern techniques based on machine learning or combinatorial testing that search through combinations of configuration flags are still not as effective at finding the best configurations for test runtime. Finally, we show that JVM configurations that provide good speedup retain this power over a number of commits. We believe that this paper provides strong motivation for further work on finding the best JVM configurations to optimize test runtime.**

***Index Terms*—JVM configurations, regression testing, performance.**

I. Introduction

Regression testing is an important part of the software development process, but can be very costly. In regression testing, developers run an existing test suite every time they make changes to the code as to check that their changes do not break any existing functionality [1]. However, test suites can take a long time to run. Further, developers are frequently making changes, triggering running the test suite for all those changes, compounding the cost of testing [2]. This high cost of testing remains a problem for developers in industry, both in the cost of developers waiting for test results and machine cost for running these tests [3]–[7].

We explore a novel direction of reducing the cost of regression testing. In particular, in Java projects, tests are all run in a Java Virtual Machine (JVM), that both interprets and compiles/optimizes an execution dynamically, including test executions. This JVM is highly configurable with flags that control memory limits, garbage collection, optimization levels, and more [8]. *Our insight is that the JVM configuration can have a high impact on test runtime*. As such, there may be a specific JVM configuration that can be used to greatly speed up the runtime for a specific test suite on a specific project. Furthermore, developers are running the same tests after every change, usually without any changes to the tests. When developers do make changes to the code under test, prior work found that these changes tend to be rather small [2], so the same tests are run on similar code, resulting in similar execution traces. Therefore, running a test suite in a JVM specifically tuned to reduce its runtime can result in these tests running faster on all future versions of code as well, thereby reducing the runtime of regression testing. Also, tuning configurations for a specific test suite run would likely lead to greater reduction in runtime than finding the best configuration in which to run the program for all general inputs, like for compiler autotuning [9]–[13].

We conduct an empirical study to evaluate the impact of configuring a JVM on test runtime and motivate future work in this new space. We evaluate on 20 popular open-source Java projects that were used in past work on software testing [2], [14]–[16]. For each project, we take a commit within that project's version history and search for flags to use from the *flag space*. We consider a combination of these flags to form a *JVM configuration*. We then measure the difference in test runtime within this configured JVM versus the runtime within the default configured JVM. Developers can use the configuration that results in the greatest reduction runtime for future runs.

We evaluate different strategies that can search through this configuration flag space. First, we evaluate *random generation*, where we generate JVM configurations by randomly combining flags. Second, we evaluate using a state-of-the-art *machine learning-based compiler autotuning* technique, BOCA [13]. BOCA was previously evaluated on GCC, searching for an optimal combination of compiler flags to use for compiling and optimizing a specific program. BOCA's goal is to produce a compiled binary that can be efficient in most scenarios, whereas our goal is to optimize for when running a specific test suite. However, BOCA's approach in searching for an optimal combination of flags matches this problem domain, and so we use it for comparison. Third, we evaluate using *combinatorial testing* [17]–[19] to generate JVM configurations. Combinatorial testing techniques aim to generate a minimal combination of flags such that all t-tuples of flags are included within those combinations. The intuition is to test the interactions between flags, which we use here to see whether those interactions lead to better test runtime.

We find that random generation can generate configurations that reduces test runtime on average by 11.90% across all projects (up to 42.13% for one project, and reduce test runtime for 17 out of 20 projects). In comparison, BOCA was only able to generate configurations that reduce test runtime by on average 2.58% (and only reduce runtime for five out of 20 projects). Using combinatorial testing, we could generate configurations that reduce test runtime by 7.84% on average (and only reduce runtime for 14 out of 20 projects). Further, when we evaluate using the best configuration across many commits in the project's history, we still observe a reduction in test runtime, with an average of 7.43% at each commit for each project. These results show that a configuration can still be relevant on commits beyond the commit for which they were generated, so developers can reuse them.

We also experiment with using the same configuration when running tests across commits while applying a regression test selection (RTS) technique [1], [2], [14], [15], [20]–[25], which runs a subset of tests after every change, to see whether the configuration still provides a reduction in runtime even if a different set of tests are run after the code changes. We see that the average reduction in runtime, when using the best configuration commits where RTS selects to run some tests, is 8.28% compared against running those same, selected tests in the default configuration. As such, the use of a new configuration for the JVM can result in runtime reduction even if developers are not running all tests.

The main contributions of our work are:

- **Idea.** We propose to modify the JVM runtime environment specifically for a given test suite by exploring JVM configurations as to reduce the test runtime. This is the first work on optimizing test runtime via JVM configurations. Our proposal stands in sharp contrast to prior work that usually run fewer tests to reduce test runtime.
- **Study.** We designed a study to evaluate the impact of JVM configurations on test runtime through random generation, an existing compiler autotuning approach, and combinatorial testing, as possible means to generate the configurations.
- **Findings.** Our results, on 20 open-source projects, show that randomly generating configurations can reduce the runtime by 11.90% on average, but finding good configurations is non-trivial. This reduction is much greater than that found using the prior compiler autotuning approach or combinatorial testing. Further, we find the reduction persist even across changes, with on average 7.43% reduction on each commit. The reduction also happens when running a subset of tests using RTS, at on average 8.28% compared against running the subset of tests with the default JVM configuration.
- **Artifact.** We share our scripts for running experiments, the configurations explored by each approach for all projects, and the experiment results [26].

Overall, our findings demonstrate the benefit in searching for an optimal configuration of JVM for testing, motivating future work in ways to find even better configuration(s) or more efficient ways to search for a good configuration.

II. Methodology

We conduct an empirical study to evaluate whether different JVM configurations can have significant impact on test runtime. A *JVM configuration* consists of a combination of JVM flags. For example, the following JVM configuration changes (a) the default JIT compilation, (b) garbage collection, and (c) the JVM runtime behaviors:

```
java -XX:-DoEscapeAnalysis
-XX:+UseSerialGC -XX:-UsePerfData
-Xshare:off
```

where the flags, in order, disable escape analysis, use the serial garbage collector, disable collecting performance data, and disables class data sharing during the JVM execution.

We run *tests* within a configured JVM to see whether its runtime changes significantly compared against when run in the default configuration. Note that, by *default configuration*, we mean whatever is the configuration the project sets for running tests. For the most part, the default configuration is simply running the JVM without additional flags. A developer would be most interested to see whether they can use a custom configuration specific for a test suite such that running those tests within a JVM using that configuration would speed up their test runtime. (Note that we are not concerned with trying to find some configuration that provides the same benefits for generally any project's test suite.)

To generate the configurations, we evaluate three strategies: (1) a random strategy, (2) an ML-based compiler autotuning approach, and (3) combinatorial testing strategies. We first describe the initial flag space from which these strategies will sample from and then describe how each strategy works.

A. Flag Space

Figure 1 shows our process to define the *flag space*, i.e., valid flags and dependencies among those flags.

Initial set of flags. First, we construct an initial set of JVM flags from which to sample (1 in Figure 1). We start with 178 documented JVM flags available in Java 8 (the version that we use) [8]. However, many of these flags would not have any (positive) effect on JVM performance, e.g., `+PrintCompilation` only prints out details concerning internal JVM compiler optimizations. As such, we manually perform an initial filtering of these flags.

Filtering irrelevant flags. We manually check the description of the available JVM flags in the official Java documentation [8] and choose a subset based on our understanding of how they might affect the runtime (2). This subset of flags can be divided into four categories. We include (1) four *Non-Standard Flags*, which are general purpose flags, e.g., `Xmx=s` configures the max heap memory to be `s`; (2) four *Advanced Runtime Flags*, which control the runtime behavior of the JVM, e.g., `UseCompressedOops` enables the use of compressed pointers; (3) 13 *Advanced JIT Compiler Flags*, which control the just-in-time (JIT) compilation and optimizations, e.g., `TieredStopAtLevel=2` stops the JIT compilation at optimization level 2; and (4) 19 *Advanced Garbage Collection Flags*, which control JVM garbage collection (GC), e.g., `UseParallelGC` forces the JVM to use the parallel garbage collector.

Values for flags. For flags involving binary choices, we choose the configuration that does the opposite of default, e.g., we use

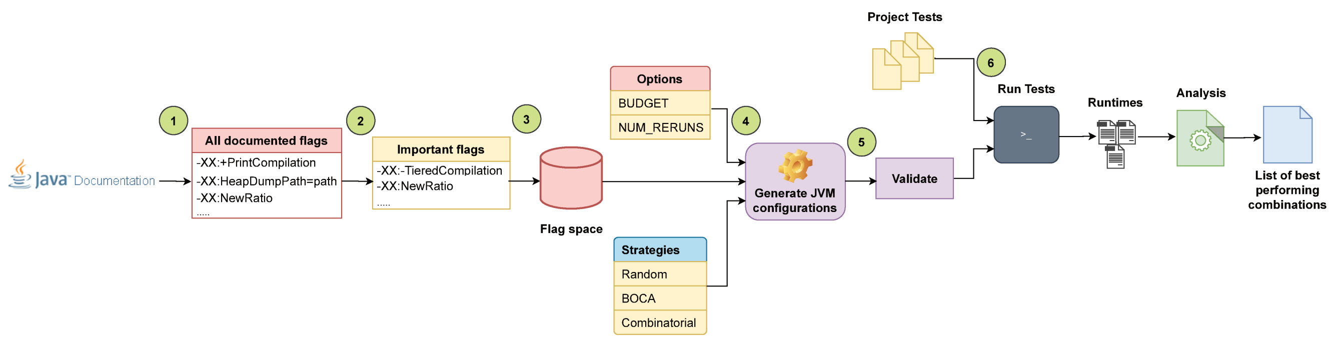

Fig. 1: Methodology to create the flag space.

the flag `-C1ProfileCalls` that disables profiling calls for the sake of JIT compilation, which is normally "on" by default. We do not explicitly include the default configuration flags in the space, because not including the flag when sampling from the space would be the same as explicitly including it. For flags that take a value from a pre-determined set, the flag space includes all those values, e.g., `AllocatePrefetchInstr` has possible values of 0 (default), 1, 2, and 3, so we consider using `AllocatePrefetchInstr` with values 1, 2, and 3 in the flag space. For flags that take a value within a continuous range, we choose two values: half and double the default value, e.g., `TargetSurvivorRatio=n` requires a value `n` ranging from 0 to 100, representing the garbage collector's survivor ratio, which by default is 50 in the JVM we use, so we choose values 25 and 100 as other choices. Each choice constitutes a new flag in the initial flag space. Ultimately, we have 72 JVM flags in the flag space from which to sample from.

Extracting constraints. In addition, we manually inspect the documentation to determine whether there are any dependencies between the flags (3 in Figure 1). For example, the flag `ParallelGCThreads=n` controls the number of threads available for the parallel garbage collector, but it only works when `UseParallelGC` is enabled. As such, if a configuration includes `ParallelGCThreads=n`, it must also include `UseParallelGC`. On the other hand, there are flags that should not be in the same configuration. For example, flags `UseParallelGC` and `UseSerialGC` both configure which garbage collector to use, but they cannot be used together. Similarly, we have explicit separate flags when they require arguments, e.g., the flag space includes `ParallelGCThreads=2` and `ParallelGCThreads=4`, but we cannot have both in the same configuration. We track all dependencies and conflicts between flags to ensure that the final combination of flags is valid.

In the end, our flag space contains 72 JVM flags, 6 dependencies, and 59 conflicts in total.

B. Generation and Running

We generate configurations using one of three strategies (4). Each strategy produces configurations to apply to the JVM when running tests. For each strategy, after it generates a configuration, we always validate the configuration w.r.t. the dependencies and constraints we determined between flags (5). If the configuration has a flag that is missing a dependency, we include the additional flag. If we identify a conflict between two flags, we determine the dependents of each of these two flags in the current configuration and ultimately remove the flag that has fewer dependents along with its dependents, in order to avoid losing too many flags at once in a configuration. We keep removing flags until resolving all conflicts. We also check that we are always generating a new configuration and not repeating seen ones.

We run all tests within a JVM using the generated configurations and check whether all tests have the same outcomes compared against running the tests using the default JVM configuration (6), i.e., we need to see the same passing tests, failing tests, those with errors, or skipped tests as in the default run. If tests do not preserve the same outcomes under a generated configuration, we discard that configuration, aiming to generate another one. We also set a timeout of twice the default time to run tests and discard any configurations that surpass this timeout as it is possible that tests under such configurations run infinitely.

C. Generation Strategies

1) Random Generation: We randomly generate configurations by sampling flags from the flag space. Figure 2 shows pseudocode representing the random generation process. First, we choose a random number `comb_size` between 2 and $len(flag\_space)$ (line 9). The reason we set this `comb_size` first is to allow for a roughly even distribution of the number of flags within the eventual combination. We then randomly sample `comb_size` flags to form a configuration (line 12). After validating the configuration (line 15) and checking it is new (lines 19-20), we add it to the overall list of configurations, which get returned at the end. We configure random generation to produce 60 configurations (`budget = 60`).

2) BOCA: BOCA is a compiler autotuning approach based off of Bayesian optimization [13]. BOCA searches for an optimal set of optimization flags for C/C++ compilers. BOCA's goal is to optimize the compilation of a C/C++ program such that the final compiled binary has good general performance, better than if compiled using the default GCC compilation flags. While BOCA has a different goal from ours (optimizing

```
def get_random_configurations(flag_space, budget):
  # A list of configurations to configure the JVM
  configurations = []
  # The maximum number of of flags per combination
  max_comb_size = len(flag_space)

  while len(configurations) < budget:
    # Randomly choose the size of the combination
    comb_size = random.randint(2, max_comb_size)

    # Randomly select flags without replacement to
        form a combination
    combination = random.sample(flag_space,
        comb_size)


    # Validate and adjust the combination
    combination = validate_combination(combination)


    configuration = ' '.join(combination)
    # Check if it's a new combination
    if is_new(configuration, configurations):
      configurations.append(configuration)
  return configurations
```

Fig. 2: Algorithm for generating random configurations.

```
1  def boca_run(flag_space, initial_size, budget):
2    # A set of all configurations to configure the JVM
3    train = set()
4    result = 1e8 # Keep track of best runtime
5    for i in range(initial_size):
6      x = random.randint(0, 2 ** len(FLAGS))
7      x = generate_conf(x)
8      ...
9      # Run combination and train the model
10     ...
11     train.add((x, runtime(x)))
12
13   steps = 0
14   while initial_size + steps < budget:
15     ...
16     # Use train to generate candidates and predict
           best solution based on a RF model
17     ...
18     train.add((best_solution, runtime(best_solution)
           ))

19     if best_result < result:
20       result = best_result
21   return train, result
```

Fig. 3: Algorithmic view of main BOCA steps.

the JVM for a specific test suite), we can still use the approach to generate configurations, since they all consist of flags.

Figure 3 shows pseudocode representing a summary of a BOCA run. Initially, BOCA constructs a surrogate model using Random Forest [27] based on an initial training set (lines 5-11) that maps configuration to the runtime, so the model can predict runtime per configuration. After training the model, BOCA identifies a new set of candidate configurations, considering both exploitation and exploration of flags, and it uses the model to predict the runtime for each one. BOCA takes the generated configuration with the best expected improvement and runs the tests five times in a JVM using that configuration to get the actual test runtime. If tests fail to run under a configuration, BOCA re-generates candidates again and tries to run the best one until a configuration is successful. This new pair of configuration and runtime is added to the training set as to retrain the surrogate model for another iteration of prediction and subsequent retraining (line 18). We configure BOCA to run until observing runtime for 60 configurations, the same as random generation. Finally, we have BOCA return all configurations it ran, along with their runtimes. The overall intuition behind BOCA is that the surrogate model can quickly guide the search process towards the configurations that provide the best runtime without actually needing to run anything, and the multiple iterations allow for the model to continuously be updated and improved.

While we evaluate BOCA using its default experimental parameters [13], we decide to modify some of these parameters to improve the process. We find that the initial training set is rather small (just two data points [13]), and training initially on more data can make a better initial surrogate model for predictions, so we change the initial training set size to 10. Furthermore, we feel the amount of reruns that BOCA uses to establish the actual runtime for a configuration to be too small (only 5), so we update the number of reruns to be the same used for establishing statistical significance (Section III). We refer to this modified version of BOCA as BOCA$_m$.

3) Combinatorial Testing: Combinatorial testing aims to efficiently test interactions between configuration flags in a system [17]–[19]. The intuition is that bugs in a configurable system are due to the interactions between just a few of those configuration flags, so the goal is to generate a minimal number of combinations of flags that encompass all t-wise tuples of those flags, where t is some small number like 2 or 3. We can similarly apply combinatorial testing techniques to our problem domain, with the intuition that interactions between the flags impact the test runtime. Combinatorial testing then outputs configurations that cover all t-wise tuples of flags.

We specifically use a state-of-the-art combinatorial testing technique SamplingCA [19] that can efficiently generate combinations that cover all pair-wise (t=2) tuples of flags. We ensure that the constraints between flags are preserved in each combination and that all pairs of flags are covered by the combinations. However, finding an optimal configuration for the JVM to run tests may require more complex interactions between flags, beyond just pairs. As such, we also evaluate three-wise (t=3) combinatorial testing with our own code: 1) we generate all possible three-wise tuples of all flags in the flag space and consider them uncovered, 2) while we still have uncovered tuples, we create a new empty combination of flags, 3) we iterate through the uncovered tuples and add each one to the combination if adding it keeps the combination valid, 4) we remove all tuples covered by the new combination from the uncovered set, and 5) we confirm that the combinations together cover all three-wise combinations. Unlike for random or BOCA, we only run as many configurations that these pair-wise or three-wise strategies would generate, namely 15 and 45 configurations, respectively.

TABLE I: Evaluation projects. 'Runtime' is in seconds.

ID	Project	SHA	# Tests	Runtime
P1	google/compile-testing	b6e19e9	231	5.15
P2	*apache/commons-email	00cc321	191	5.59
P3	logfellow/logstash-logback-encoder	7044d87	462	6.69
P4	*jhy/jsoup	afc38d8	1181	6.96
P5	apache/commons-csv	547c5a2	829	8.60
P6	apache/commons-codec	e2cecc7	1339	12.35
P7	apache/commons-collections	dc6d9f8	6386	19.41
P8	*apache/commons-jexl	f8725b2	964	24.29
P9	JSQLParser/JSqlParser	0ec2600	1608	24.94
P10	*apache/commons-configuration	ed32b41	2873	27.81
P11	*apache/commons-beanutils	41f7b90	1353	35.49
P12	asterisk-java/asterisk-java	5c16184	358	36.08
P13	*apache/commons-imaging	c021adf	991	41.72
P14	fasterxml/jackson-core	90eaabb	1291	56.39
P15	apache/commons-bcel	c819e54	375	59.55
P16	apache/commons-compress	74256e9	2400	65.00
P17	apache/commons-net	26fbd9e	415	65.91
P18	*tabulapdf/tabula-java	8bfa3ad	210	73.83
P19	*apache/commons-dbcp	6f7ec82	1501	104.22
P20	addthis/stream-lib	af045cb	137	142.78

III. Experiment Setup

We address the following research questions:

- **RQ1**: How much reduction in test runtime can be achieved by applying different JVM configurations?
- **RQ2**: How much can the best configuration be minimized and which flags are the most common between projects?
- **RQ3**: How much reduction in test runtime can the best configuration maintain across commits?
- **RQ4**: How much reduction in test runtime can the best configuration maintain even when running different tests using regression test selection (RTS)?

We address RQ1 to check whether running tests using a JVM under different configurations can significantly reduce test runtime. We address RQ2 to see whether a configuration can be minimized, removing redundant/unnecessary flags while preserving runtime reduction. We are interested to see whether there are flags that are useful across different projects. We address RQ3 to see whether the proposed configurations can maintain a reduction in runtime even as developers make changes to their project. Finally, we address RQ4 to see whether the reduction in runtime persists even if developers are not running all the tests. Since developers would commonly use regression test selection (RTS) [2], [4]–[6] as part of their development process, we want to see whether changing the JVM configuration can help reduce runtime even further on top of RTS.

Subjects. We perform our study on 20 open-source Java projects from GitHub. We collect these projects based on prior work on RTS [2], [14]–[16], where researchers evaluated their test runtimes across many commits. We select single module Maven projects that build using Java 8 and can finish the end to end experiment within 24 hours. For each project, we take the latest commit at the time of our experiments, and we generate different configurations at this commit. Table I lists for each projects an ID for use in later tables, the commit we use, the number of tests, and test runtime (in seconds) in the default configuration. If the project uses some JVM flags for their testing, we mark that project with * in the table.

Experimental steps. To measure statistical significance in runtime differences between configurations, we rerun all configurations 10 times to collect a distribution of test runtimes. We compare this distribution of runtimes against the test runtimes collected from running the tests in the default configuration the same number of times, and we use the Student's t-test [28] to compare the two distributions.

Configuration minimization. To minimize the flags in a configuration, we iteratively run tests under a JVM configured using subsets of flags from the initial configuration. For instance, if we have a set of flags represented as {`f1, f2, f3, f4`}, the process begins by checking the impact of `f1` on test runtime. Subsequently, if this individual flag does not yield the expected reduction in runtime over running in the default configured JVM, we try the subset combination {`f2, f3, f4`}. If this subset combination of flags provides a similar or better improvement as the the original combination {`f1, f2, f3, f4`} that we started with, then the subset {`f2, f3, f4`} becomes the new subset we want to minimize. This process continues, systematically evaluating different combinations of flags, until either an individual flag is found to be sufficient or all flags have been checked individually. In the end, we sort all the configurations we tried during the process and pick the best configuration that provides a similar or better improvement as the initial starting one. We acknowledge that different minimization techniques are possible, but we leave exploring those for future work.

Running across commits. We run the tests under the best configuration found for the project on the latest commit across 50 commits going backwards in history from that latest commit for the project. We also run using the best configuration across commits while applying RTS. We use STARTS, a static, class-level RTS technique [14], [15], [29]. If STARTS determines no tests should be run due to some changes, we ignore these commits. For the other commits where tests are run, we measure runtime when running those tests with the best configuration and in the default JVM.

Hardware configurations. We run all our experiments in a Docker container built from an Ubuntu 20.04 Docker image. We use JDK 8 and Maven 3.8.3, and we use Python 3 for data collection and analysis. We run each project in its own container and we limit the container to use 2 CPUs and 8GB of RAM, similar to resources available in continuous integration services [30], [31].

IV. Evaluation

A. *RQ1: Reduction in Test Runtime*

Figure 4 shows for each project a boxplot representing the spread of runtime reduction achieved using the randomly generated 60 configurations, compared against using the default JVM configuration. The dashed red line across the boxplots represents 0%, namely having the same runtime as when

TABLE II: Number of significant differences in runtimes across randomly generated configurations per project.

Project	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	P16	P17	P18	P19	P20
# Signf. pos.	12	6	19	1	6	5	13	1	1	8	7	6	8	7	19	8	0	0	0	26
# Signf. neg.	33	33	26	35	19	34	9	18	46	46	26	38	46	43	19	50	46	58	11	23

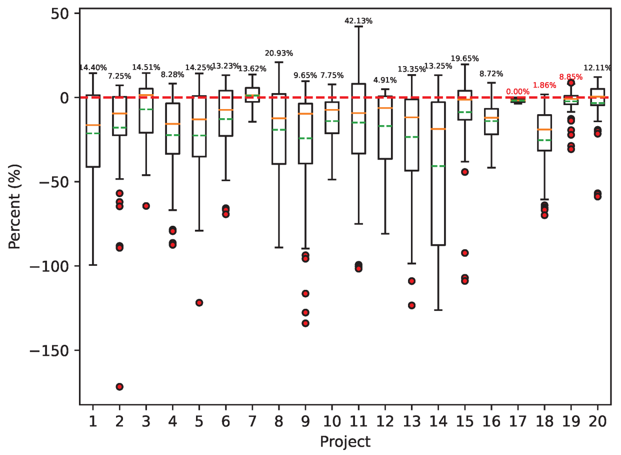

Fig. 4: Boxplots of median runtime reduction (higher is better) per project for randomly generated configurations.

using the default JVM configuration. We see from the figure that the test runtime is indeed impacted by the change in JVM configuration. While most JVM configurations result in a negative reduction in runtime (i.e., test runtime is higher than that when run in the default JVM configuration), we see that there exists configurations that result in positive, statistically significant reduction in runtime (data above the red line).

Table II shows, for each project, the number of randomly generated configurations with a positive reduction in runtime (top row) and the number with a negative significant reduction (bottom row). (Runs that were not statistically significant are not counted.) Only three projects, P17, P18, and P19, did not have a configuration that performs better than the default.

Figure 5 further illustrates the comparison between the test runtime in the default JVM configuration versus that achieved in the configuration resulting in the biggest reduction in runtime. Each plot shows two boxes, where the left box shows the 10 runtimes from running the tests in the default JVM configuration while the right box illustrates the 10 runtimes from running the tests in the best configuration generated for that project. The dashed red line is the median runtime found for the default runtimes. For all but one projects, we see that the median runtime from the best configuration runtimes is lower than that from the default JVM configuration. In fact, for most projects, we observe that *all* runtimes collected for the best configuration are lower than *all* runtimes collected for the default JVM configuration. There are a few projects where there is substantial overlap in runtimes between the two, suggesting no statistically significant difference.

Random generation. Table III tabulates the median reduction in runtime achieved by the best configuration found from randomly generating configurations, per project (column "Rnd. (%)"). We measure reduction as the difference between the median runtime in the default JVM and the median runtime using the best configuration, divided by runtime in the default JVM (the bigger the better). We only show the reduction in the table if there is a positive, statistically significant reduction ($p < 0.05$), and we show 0 otherwise (a developer would use the default JVM configuration). For the projects where there is a configuration that reduces runtime, the reduction ranges from 4.91% to 42.13% compared against default.

BOCA. Table III also shows per project the median reduction in runtime achieved by the best configuration found by using BOCA (column "BOCA (%)") and BOCA$_m$ (column "BOCA$_m$ (%)"). We see that BOCA and BOCA$_m$ could generate a configuration that significantly reduces runtime for only five and seven projects, respectively. For every other project, we present the reduction as 0 to indicate no significant runtime reduction. Interestingly, for four of these projects where BOCA achieves a reduction in runtime, that reduction is higher than the best configuration found using random generation, with even one project where BOCA finds a configuration that reduces runtime whereas the random generation does not. For BOCA$_m$, we observe three projects where BOCA$_m$ generates a configuration better than the best one generated randomly. However, overall, the average reduction in runtime across all projects is only 2.58% (including the 0 reduction for many of the projects) and 4.26% for BOCA and BOCA$_m$, respectively, indicating that the BOCA strategy is not as effective. These results suggest that existing machine learning-based approaches are lacking in this new domain of configuring the JVM to optimize test runtime.

Combinatorial testing. Table III also shows per project the median reduction in runtime achieved by the best configuration found by using pair-wise testing (column "2-wise (%)") and three-wise testing (column "3-wise (%)"). For pair-wise testing, the average reduction in runtime across all projects is only 4.79%. Only five projects could use one of the pair-wise testing configurations to achieve a reduction in runtime. For three-wise testing, the average reduction in runtime across all projects is 7.84%, which is higher than pair-wise. Further, 14 projects could use one of the three-wise testing configurations to achieve a reduction in runtime. These results suggest that higher reduction in runtime comes from exploring interactions between more flags, i.e., the interactions between pairs of flags is insufficient to substantially reduce runtime. However, all these configurations still do not have as much reduction as configurations generated randomly. These results suggest that substantial runtime reduction requires more complex interactions between flags in this domain. It is possible that higher t-wise testing would find better configurations for all projects at the expense of needing to try many more configurations.

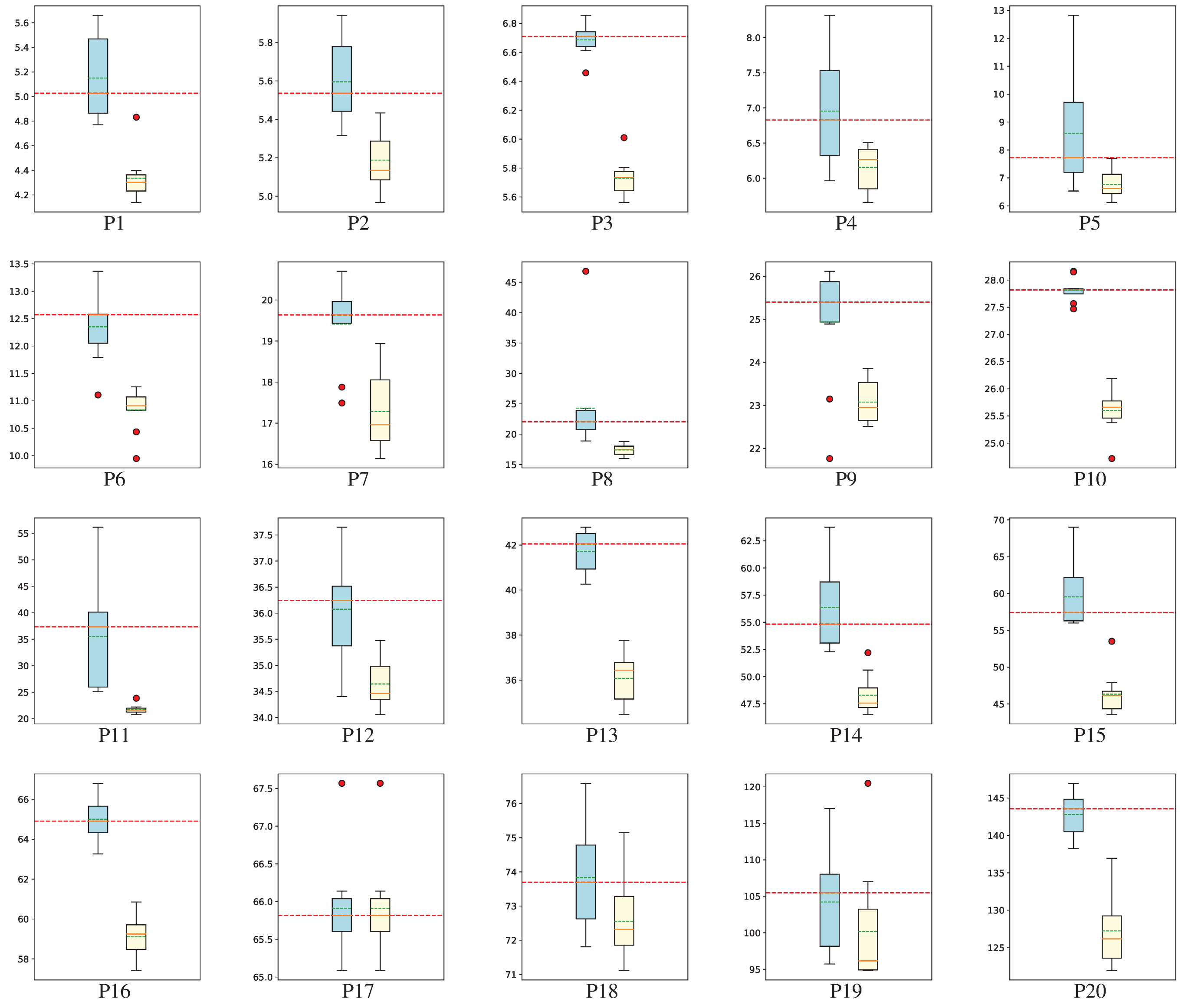

Fig. 5: Default (box on the left) vs. best (box on the right) configuration runtimes for each project, measured in seconds. Dashed red line is median of default runtimes. Solid yellow is the median of best configuration runtimes. Dashed green line is the mean of each box runtimes.

RQ1: *Different JVM configurations can significantly impact test runtime. Randomly generating these configurations can result in configurations that reduce the runtime by 11.90%, on average. BOCA and combinatorial testing can generate configurations that reduce runtime on average by 4.26% and 7.84%, respectively, which is less than that achievable by simply randomly generating configurations.*

For all remaining research questions, we consider only the best configurations generated randomly, given that those configurations generally provide greater reductions in runtime among all strategies.

B. *RQ2: Minimal Common Flags*

Table IV shows the results of minimizing the set of flags from the best configuration generated. We show results only for projects where there was actually a configuration that significantly reduces runtime (so we do not show results for projects P17, P18, and P19). We show for each project the number of flags remaining after minimization compared against the number of flags originally in the best randomly generated configuration, before minimization. When there is only one flag remaining after minimization, we also show what the flag is in the table (column “Single flag”).

TABLE III: Reduction in runtime.

Project	Rnd (%)	BOCA (%)	BOCA_m (%)	2-wise (%)	3-wise (%)
P1	14.40	0.00	12.13	14.38	13.22
P2	7.25	12.46	5.40	0.00	11.75
P3	14.51	16.33	14.79	9.99	14.78
P4	8.28	0.00	0.00	0.00	10.09
P5	14.25	0.00	0.00	11.34	14.48
P6	13.23	11.12	0.00	0.00	3.86
P7	13.62	0.00	0.00	0.00	8.51
P8	20.93	0.00	24.41	0.00	0.00
P9	9.65	0.00	9.58	0.00	0.00
P10	7.75	10.12	11.17	16.21	14.56
P11	42.13	0.00	7.62	43.89	43.14
P12	4.91	0.00	0.00	0.00	3.11
P13	13.35	0.00	0.00	0.00	3.13
P14	13.25	0.00	0.00	0.00	0.00
P15	19.65	0.00	0.00	0.00	8.68
P16	8.72	0.00	0.00	0.00	3.36
P17	0.00	1.52	0.00	0.00	0.00
P18	0.00	0.00	0.00	0.00	0.00
P19	0.00	0.00	0.00	0.00	0.00
P20	12.11	0.00	0.00	0.00	4.03
Average	**11.90**	**2.58**	**4.26**	**4.79**	**7.84**

TABLE IV: Minimized configurations.

Project	Min. len.	Orig. len.	Red. inc.	Single flag
P1	5	29	0.00	-
P2	1	7	2.91	XX:C1UpdateMethodData
P3	1	26	1.00	XX:TieredStopAtLevel=2
P4	1	32	0.85	XX:SurvivorRatio=16
P5	27	27	0.00	-
P6	1	11	17.05	XX:C1ProfileCalls
P7	28	34	0.00	-
P8	17	17	0.00	-
P9	5	21	0.53	-
P10	1	8	9.34	XX:TieredStopAtLevel=2
P11	7	11	1.43	-
P12	10	10	0.00	-
P13	1	4	1.34	Xnoclassgc
P14	1	11	10.94	XX:DoEscapeAnalysis
P15	11	11	0.00	-
P16	1	24	6.27	XX:NewRatio=1
P20	1	5	8.99	XX:C1OptimizeVirtualCallProfiling

TABLE V: Most common flags after minimization.

Flag	# Projects
-XX:-C1ProfileCalls	5
-XX:TieredStopAtLevel=2	5
-XX:-UsePerfData	4
-Xshare:off	4
-XX:+UseCondCardMark	4

We see that the number of flags after minimization drops substantially, with 10 projects having just one flag. Only four projects have the same number of flags remaining after minimization (projects P5, P8, P12, and P15), suggesting that their runtime reductions require the interactions between all flags in the configuration. On average, the number of flags in each best configuration drops from 16.94 to 7.00 after minimization. Interestingly, we observe that for cases where we could remove flags, there is an even greater reduction in runtime. We show the additional percentage-point increase in reduction in the table per project; the average increase is 3.57 percentage-points.

Table V shows the top five flags found among the minimized best configurations across all projects, ranked by their frequency among projects (column “# Projects”). The most common flags are `-C1ProfileCalls` and `TieredStopAtLevel=2`, which are flags related to JVM JIT optimizations. `-C1ProfileCalls` stops the JVM from profiling method calls for information to make optimization decisions; removing profiling can reduce runtime overhead. `TieredStopAtLevel=2` prevents the JIT compiler to optimize at the highest level, which requires more profiling that adds overhead. The prevalence of these flags suggest that projects' tests are fast-running enough that it is not worth it for the overhead needed to determine when to optimize at the highest level. However, these flags only occur among five projects, so most projects do not share any common flags, suggesting that there is no “easy” solution of using just a few flags that would generally reduce test runtime of any project.

RQ2: *We can reduce the number of flags in the best configurations while still preserving or even improving the reduction in runtime. We observe some flags that are in common among the best configurations after minimization, but not among the majority of projects.*

TABLE VI: Reduction in runtime across commits.

Project	# Commits	Red. (%)	# Commits > 0%
P1	50	12.61	50
P2	50	5.19	50
P3	50	12.16	50
P4	50	9.99	50
P5	50	9.52	50
P6	50	9.20	50
P7	50	3.58	42
P8	50	15.48	50
P9	40	-0.58	14
P10	50	10.39	50
P11	49	43.46	49
P12	50	0.66	35
P13	50	-0.23	21
P14	50	-6.26	2
P15	50	3.08	50
P16	50	-1.38	16
P20	50	-0.48	21
Average	-	**7.43**	-

C. RQ3: Reduction in Runtime across Commits

Table VI show the results of running tests across multiple commits with a JVM configured using the best randomly generated configuration, found on the starting commit. We can only evaluate on commits for which we can build the code on that commit; the total number of commits we evaluate per project is shown under the "# Commits" column. Also, we exclude the projects where we do not find a configuration that statistically significantly reduces the runtime from the default (Section IV-A), representing when a developer would simply use the default JVM configuration.

On average, the best configuration maintains a reduction of 7.43% in runtime across commits. We also show the number of commits for which the configuration achieves a positive, significant reduction in runtime over the runtime from running in the default JVM, under column "# Commits > 0%". We see that 9 projects maintain a positive reduction across all 50 commits.

Figure 6 illustrates how much the reduction in runtime varies from commit to commit for each project when running under the generated configuration. The figure shows a box per project (ID on x-axis) where the y-axis is the percentage reduction relative to runtime from running in the default JVM on the corresponding commit. We see that the possible reduction can range widely for different projects and their commits, with some commits even having increased runtime. However, overall, most of the boxplot values remain above 0 (the dashed red line), indicating a general reduction in runtime across commits.

RQ3: *Reusing the best configuration generated on one version of the project can still reduce the test runtime on other commits, with an average reduction in runtime per commit of 7.43%.*

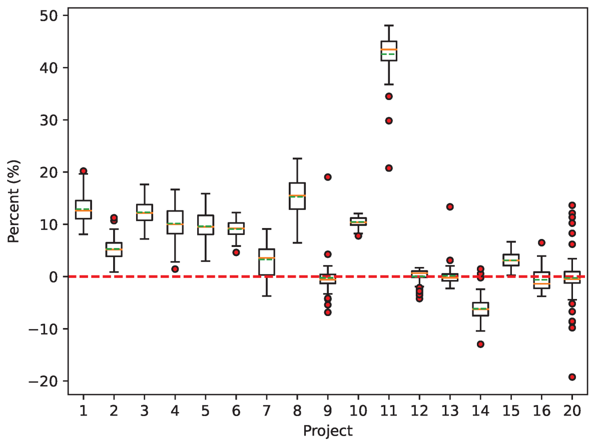

Fig. 6: Distribution of reduction in runtime across commits per project.

TABLE VII: Reduction in runtime across commits with RTS.

Project	# Commits	Red. (%)
P1	49	19.21
P2	9	3.91
P3	19	16.74
P4	30	12.35
P5	12	12.02
P6	15	10.50
P7	21	9.52
P8	21	13.84
P9	26	-6.42
P10	24	12.31
P11	13	34.44
P12	16	-2.11
P13	21	0.69
P14	34	-2.61
P15	8	2.16
P16	26	2.91
P20	8	1.23
Average	-	**8.28**

D. RQ4: Reduction in Runtime with RTS

Table VII show the results of running tests via RTS across multiple commits with a JVM configured with the best randomly generated configuration. We report in the tables the number of commits on which we evaluate (column "# Commits"), given that the RTS tool STARTS may select to run no tests based on the changes (Section III). We show the runtime reduction compared against running the same tests selected using RTS across commits but using the default JVM configuration. On average, we see that the best configuration maintains an average reduction in runtime of 8.28%.

Figure 7 shows boxplots of the range of runtime reductions across commits on which we run STARTS. Compared against the results when not using RTS, we observe more projects with higher variance in the reduction in runtime across commits, possibly due to generally running fewer tests using RTS.

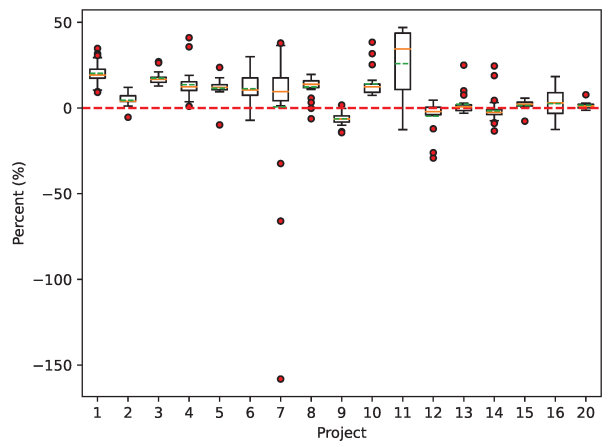

Fig. 7: Distribution of reduction in runtime across commits with RTS per project.

RQ4: *The best configuration can still provide a reduction in runtime across commits even when run on top of using RTS techniques that run fewer tests, with an average percentage reduction in runtime of 8.28%. Running tests under custom JVM configurations can help reduce the cost of regression testing even further when combined with RTS.*

E. Discussion

Reduction in runtime. To better understand the reduction in runtime, we examine further the tests from project P11, which had the highest reduction in runtime (Table III). We observe one test class, `MemoryLeakTestCase` that takes about 82% of the entire test suite runtime. The tests within `MemoryLeakTestCase` try to force the garbage collector to run by filling up memory. As a result, the tests are highly affected by the garbage collection policy. We find that the minimized configuration that preserves the runtime reduction for this project indeed includes flags that modify the garbage collection policy.

Concerning cases where there is no reduction in runtime, we inspected the tests of these projects further. For projects P18 and P19, none of the strategies for generating configurations could generate one that reduces runtime. From Figure 4, we see that project P18 has most of its randomly generated configurations resulting in negative reduction in runtime, suggesting that while the tests in this project are impacted by changes in configurations, it can be difficult to find one that actually reduces the runtime. For project P17, we see from Figure 4 that the distribution of runtimes obtained from the randomly generated configurations is very tight, with most of them very close to 0 reduction. When we investigated the tests in P17, we find one test class, `TelnetClientTest` that takes about 70% of the test suite runtime. The tests within `TelnetClientTest` call `Thread.sleep()` with a large wait time, which forces each test to run at least as long as that time. As such, the test runtimes are more affected by these long wait times than any changes due to different configurations.

Configurations that lead to different test results. Recall from Section II that it is possible that tests do not exhibit the same results when run under some configurations, and we have to discard these configurations. For some projects, we observe a high number of configurations that lead to different test results, with as much as 237 for one project. We looked into the reasons for why tests have different results under certain configurations, and we find that for the most part the tests are actually being skipped. We see that these tests use `Assumptions`, an API built into JUnit that allows a test to be skipped if certain dynamic conditions are not met, such as having enough available memory. Given that many of the flags affect conditions such as memory, several configurations that use such flags can result in these tests being skipped. We also observe tests that error and fail when they used to be passing due to similar reasons involving configurations. Future work that aims to efficiently generate valid configurations for a specific test suite would need to take into account these relations between tests and flags as to avoid using such flags as part of the search.

Running cost. While random generation seems to be the most effective at generating configurations that can reduce test runtime, our random generation experiments on average take around 15 hours per project, mainly due to the 10 reruns per configuration we use to collect enough runtimes as to perform statistical analysis. If a developer were to directly use this approach to search for a configuration that reduces the test runtime the most, they would encounter this search cost. Potentially, a developer could run this search on a weekly basis (over the weekends), and the continuous integration test runs over the subsequent week can use the best configuration found, allowing faster testing that entire week. However, we note that, in this work, we are focused on evaluating the impact of changing JVM configurations on test runtime, and the presented techniques we use may not be the most efficient still. Future work can explore less costly approaches to achieve similar reductions, e.g., not rerunning tests 10 times per configuration during search.

V. Threats to Validity

Our results may not generalize to all projects. We create our evaluation dataset by consulting prior work with the similar goal of reducing test runtime, i.e., work on RTS, and filter out those whose test runtime is too short. We run the tests from each project in a Docker container configured to be similar to those provided by common continuous integration platforms [30], [31]. The test runtimes we observe are representative of what developers would see when they run in their continuous integration.

To scale our experiments, we ran each project in it own Docker container. We configure each container to use the same resources and run multiple containers simultaneously on the same machine (Section III). We believe we have taken the proper steps to mitigate the noise from running our experiments at different times.

Machine performance could be impacted by various factors, e.g., background processes running on a machine, so runtime might differ for the same experiment across several days. To mitigate this problem, we compare any non-default run with the default run that we ran closest in time to it.

VI. Related Work

Regression test selection (RTS) aims to run only the tests impacted by the changes as code evolves via analyzing the relationship between code changes and tests [2], [15], [20]–[23], [25], [32] or even through machine learning to predict which tests' outcomes differ after the change [6], [33]–[35]. However, RTS techniques can be unsafe, i.e., they miss to select truly impacted tests due to limitations in the selection approach [36], [37]. Test-suite reduction (TSR) analyzes the tests on a single code version, removing tests that are redundant w.r.t. some metric, such as code coverage [38]–[45]. While the reduced test suites run faster on future commits, they may still miss to detect future faults because the relevant tests are not in the reduced test suite anymore [46]. Finally, test-case prioritization (TCP), in contrast with RTS and TSR, does run all tests, but runs them in a different order as to prioritize the tests likely to detect faults, based on metrics like code coverage or code diversity [47]–[56]. While TCP does not miss to run relevant tests, the runtime overall remains the same. In contrast with all these techniques, we study the impact of using different JVM configurations to speed up testing while running all tests, without risk of missing to run key tests. We also evaluate how this idea of using a custom JVM configuration for tests can be combined with RTS to have additional reductions in runtime.

Prior work on configuration testing focused on exploring the space of configurations to expose faults in highly configurable software, both in how to efficiently explore that space while avoiding the combinatorial explosion problem and to better test configurations as software evolves [57]–[63]. While we also evaluate searching through a configuration space, specifically for the JVM, our focus is not on detecting faults that occur due to these configurations, neither in the JVM itself or code that runs on the JVM. We aim to evaluate the impact that differences in configurations has on test runtime, leading to configurations that reduce that runtime for future testing.

Compiler autotuning generates the best combination of compiler optimization flags for compiling a program [9]–[13]. These techniques use machine learning approaches to search the compiler optimization flag space and find the best combination of flags to create a compiled binary that is generally performant, no matter the input. Opentuner [64] is a general autotuning framework that relies on user-specified search space. Canales et al. proposed to represent JVM flags as features in a feature model and use this model as an input for a genetic algorithm [65]. Our work evaluates the effectiveness of tuning JVM flags specifically for executing specific tests, not all possible inputs. Despite the difference in goals, the two directions both similarly have to generate combination of flags, so we use recent compiler autotuning technique BOCA [13] in our evaluation, finding that BOCA on its own is not effective in this specific problem domain.

There have been other proposed work for speeding up testing. Dong et al. proposed saving and reusing common state generated by tests across commits, with the intuition that the time to generate this state normally is more time-consuming than loading the state [66]. Saff et al. proposed reducing test runtime by converting expensive parts of test execution to instead use mocks [67]. Stratis and Rajan proposed a way to reorder tests as to minimize cache misses, thereby reducing test runtime for future versions while still running all tests [68]. Our work has a similar intuition in optimizing based on a current version of code and reusing the optimizations for future versions, except we focus on configuring the JVM.

VII. Conclusions

We evaluate the effects of different JVM configurations on test runtime. The intuition is that, given developers are rerunning the same tests during regression testing, if there are configurations that reduce test runtime on one commit, reusing these configurations can similarly reduce test runtime for future commits. Our evaluation involves generating JVM configurations, which are combinations of JVM configuration flags, with which to apply to the JVM where tests are run. We use different strategies for generating configurations: random generation, machine learning-based compiler autotuning, or combinatorial testing. Our evaluation on 20 open-source Java projects shows that different JVM configurations can lead to reductions in test runtime by 11.90% on average (and up to 43.89%). We find that random generation tends to produce the best configurations that reduce runtime the most, suggesting existing work in compiler autotuning or combinatorial testing are still lacking in this domain. We further evaluate the best configurations across commits, finding a similar reduction in runtime. We also find that using the same configurations even when not running the same tests every time, i.e., running on top of RTS that runs a subset of tests, can still provide significant reduction in runtime. Future work should develop techniques to generate configurations that significantly reduce runtime more efficiently than random generation. These future techniques should also be more effective, generating configurations that can result in even greater reductions in runtime.

Acknowledgment

We thank the anonymous reviewers for their comments and feedback. This work is partially supported by the US National Science Foundation under Grant Nos. CCF-2107291, CCF-2217696, and CCF-2313027.

References

[1] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Journal of Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[2] M. Gligoric, L. Eloussi, and D. Marinov, "Practical regression test selection with dynamic file dependencies," in *International Symposium on Software Testing and Analysis*, 2015, pp. 211–222.

[3] K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy, “The art of testing less without sacrificing quality,” in *International Conference on Software Engineering*, 2015, pp. 483–493.

[4] A. Shi, S. Thummalapenta, S. K. Lahiri, N. Bjørner, and J. Czerwonka, “Optimizing test placement for module-level regression testing,” in *International Conference on Software Engineering*, 2017, pp. 689–699.

[5] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco, “Taming Google-scale continuous testing,” in *International Conference on Software Engineering, Software Engineering in Practice*, 2017, pp. 233–242.

[6] M. Machalica, A. Samylkin, M. Porth, and S. Chandra, “Predictive test selection,” in *International Conference on Software Engineering, Software Engineering in Practice*, 2019, pp. 91–100.

[7] S. Elbaum, G. Rothermel, and J. Penix, “Techniques for improving regression testing in continuous integration development environments,” in *International Symposium on Foundations of Software Engineering*, 2014, pp. 235–245.

[8] “Java 8 documentation,” https://docs.oracle.com/javase/8/docs/technotes/tools/unix/java.html, 2022.

[9] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano, “A survey on compiler autotuning using machine learning,” *ACM Computing Surveys (CSUR)*, vol. 51, no. 5, pp. 1–42, 2018.

[10] A. H. Ashouri, G. Mariani, G. Palermo, and C. Silvano, “A bayesian network approach for compiler auto-tuning for embedded processors,” in *2014 IEEE 12th Symposium on Embedded Systems for Real-time Multimedia (ESTIMedia)*. IEEE, 2014, pp. 90–97.

[11] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. O’Boyle, J. Thomson, M. Toussaint, and C. K. Williams, “Using machine learning to focus iterative optimization,” in *International Symposium on Code Generation and Optimization (CGO’06)*. IEEE, 2006, pp. 11–pp.

[12] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. O’Boyle, and O. Temam, “Rapidly selecting good compiler optimizations using performance counters,” in *International Symposium on Code Generation and Optimization (CGO’07)*. IEEE, 2007, pp. 185–197.

[13] J. Chen, N. Xu, P. Chen, and H. Zhang, “Efficient compiler autotuning via bayesian optimization,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1198–1209.

[14] O. Legunsen, A. Shi, and D. Marinov, “STARTS: STAtic Regression Test Selection,” in *International Conference on Automated Software Engineering (Tool Demonstrations Track)*, 2017, pp. 949–954.

[15] O. Legunsen, F. Hariri, A. Shi, Y. Lu, L. Zhang, and D. Marinov, “An extensive study of static regression test selection in modern software evolution,” in *International Symposium on Foundations of Software Engineering*, 2016, pp. 583–594.

[16] M. Gligoric, L. Eloussi, and D. Marinov, “Ekstazi: Lightweight test selection,” in *International Conference on Software Engineering (Tool Demonstrations Track)*, 2015, pp. 713–716.

[17] R. Kuhn, R. Kacker, and Y. Lei, *Introduction to Combinatorial Testing*. Chapman and Hall, 2013.

[18] C. Nie and H. Leung, “A survey of combinatorial testing,” *ACM Computing Surveys*, vol. 43, no. 2, pp. 1–29, 2011.

[19] C. Luo, Q. Zhao, S. Cai, H. Zhang, and C. Hu, “SamplingCA: effective and efficient sampling-based pairwise testing for highly configurable software systems,” in *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 1185–1197.

[20] G. Rothermel and M. J. Harrold, “A safe, efficient regression test selection technique,” *ACM Transactions on Software Engineering Methodology*, vol. 6, no. 2, pp. 173–210, 1997.

[21] L. Zhang, “Hybrid regression test selection,” in *International Conference on Software Engineering*, 2018, pp. 199–209.

[22] A. Orso, N. Shi, and M. J. Harrold, “Scaling regression testing to large software systems,” in *International Symposium on Foundations of Software Engineering*, 2004, pp. 241–251.

[23] A. Shi, P. Zhao, and D. Marinov, “Understanding and improving regression test selection in continuous integration,” in *International Symposium on Software Reliability Engineering*, 2019, pp. 228–238.

[24] A. Celik, M. Vasic, A. Milicevic, and M. Gligoric, “Regression test selection across JVM boundaries,” in *Symposium on the Foundations of Software Engineering*, 2017, pp. 809–820.

[25] A. Gyori, O. Legunsen, F. Hariri, and D. Marinov, “Evaluating regression test selection opportunities in a very large open-source ecosystem,” in *International Symposium on Software Reliability Engineering*, 2018, pp. 112–122.

[26] “Impact of JVM configurations on test runtime dataset,” https://sites.google.com/view/jvm-impact-on-test-runtime, 2024.

[27] L. Breiman, “Random forests,” *Machine learning*, vol. 45, pp. 5–32, 2001.

[28] Student, “The probable error of a mean,” *Biometrika*, pp. 1–25, 1908.

[29] “STARTS,” https://github.com/TestingResearchIllinois/starts, 2024.

[30] “GitHub Actions,” https://github.com/features/actions, 2024.

[31] “Travis-CI,” https://travis-ci.org, 2024.

[32] E. D. Ekelund and E. Engström, “Efficient regression testing based on test history: An industrial evaluation,” in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 449–457.

[33] J. Zhang, Y. Liu, M. Gligoric, O. Legunsen, and A. Shi, “Comparing and combining analysis-based and learning-based regression test selection,” in *International Conference on Automation of Software Test*, 2022, pp. 17–28.

[34] D. Elsner, F. Hauer, A. Pretschner, and S. Reimer, “Empirically evaluating readily available information for regression test optimization in continuous integration,” in *International Symposium on Software Testing and Analysis*, 2021, pp. 491–504.

[35] Y. Wu, Y. Chen, X. Xie, B. Yu, C. Fan, and L. Ma, “Regression testing of massively multiplayer online role-playing games,” in *2020 IEEE international conference on software maintenance and evolution (ICSME)*, 2020, pp. 692–696.

[36] C. Zhu, O. Legunsen, A. Shi, and M. Gligoric, “A framework for checking regression test selection tools,” in *International Conference on Software Engineering*, 2019, pp. 430–441.

[37] A. Shi, M. Hadzi-Tanovic, L. Zhang, D. Marinov, and O. Legunsen, “Reflection-aware static regression test selection,” *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 187:1–187:29, 2019.

[38] T. Y. Chen and M. F. Lau, “A new heuristic for test suite reduction,” *Journal of Information and Software Technology*, vol. 40, no. 5-6, pp. 347–354, 1998.

[39] ——, “A simulation study on some heuristics for test suite reduction,” *Journal of Information and Software Technology*, vol. 40, no. 13, pp. 777–787, 1998.

[40] G. Rothermel, M. J. Harrold, J. von Ronne, and C. Hong, “Empirical studies of test-suite reduction,” *Journal of Software Testing, Verification and Reliability*, vol. 12, no. 4, pp. 219–249, 2002.

[41] H. Zhong, L. Zhang, and H. Mei, “An experimental study of four typical test suite reduction techniques,” *Journal of Information and Software Technology*, vol. 50, no. 6, pp. 534–546, 2008.

[42] J. Black, E. Melachrinoudis, and D. Kaeli, “Bi-criteria models for all-uses test suite reduction,” in *International Conference on Software Engineering*, 2004, pp. 106–115.

[43] D. Hao, L. Zhang, X. Wu, H. Mei, and G. Rothermel, “On-demand test suite reduction,” in *International Conference on Software Engineering*, 2012, pp. 738–748.

[44] J. A. Jones and M. J. Harrold, “Test-suite reduction and prioritization for modified condition/decision coverage,” in *International Conference on Software Maintenance*, 2001, pp. 92–102.

[45] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid, “An empirical study of JUnit test-suite reduction,” in *International Symposium on Software Reliability Engineering*, 2011, pp. 170–179.

[46] A. Shi, A. Gyori, S. Mahmood, P. Zhao, and D. Marinov, “Evaluating test-suite reduction in real software evolution,” in *International Symposium on Software Testing and Analysis*, 2018, pp. 84–94.

[47] J. Chen, Y. Lou, L. Zhang, J. Zhou, X. Wang, D. Hao, and L. Zhang, “Optimizing test prioritization via test distribution analysis,” in *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 656–667.

[48] Z. Li, M. Harman, and R. M. Hierons, “Search algorithms for regression test case prioritization,” *IEEE Transactions on Software Engineering*, vol. 33, no. 4, pp. 225–237, 2007.

[49] T. Mattis and R. Hirschfeld, “Lightweight lexical test prioritization for immediate feedback,” *Programming Journal*, vol. 4, pp. 12:1–12:32, 2020.

[50] B. Jiang, Z. Zhang, W. K. Chan, and T. H. Tse, “Adaptive random test case prioritization,” in *International Conference on Automated Software Engineering*, 2009, pp. 233–244.

[51] Y. Lu, Y. Lou, S. Cheng, L. Zhang, D. Hao, Y. Zhou, and L. Zhang, "How does regression test prioritization perform in real-world software evolution?" in *International Conference on Software Engineering*, 2016, pp. 535–546.

[52] Q. Luo, K. Moran, and D. Poshyvanyk, "A large-scale empirical comparison of static and dynamic test case prioritization techniques," in *International Symposium on Foundations of Software Engineering*, 2016, pp. 559–570.

[53] Q. Peng, A. Shi, and L. Zhang, "Empirically revisiting and enhancing IR-based test-case prioritization," in *International Symposium on Software Testing and Analysis*, 2020, pp. 324–336.

[54] R. K. Saha, L. Zhang, S. Khurshid, and D. E. Perry, "An information retrieval approach for regression test prioritization based on program changes," in *International Conference on Software Engineering*, 2015, pp. 268–279.

[55] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei, "Bridging the gap between the total and additional test-case prioritization strategies," in *International Conference on Software Engineering*, 2013, pp. 192–201.

[56] A. Sharif, D. Marijan, and M. Liaaen, "Deeporder: Deep learning for test case prioritization in continuous integration testing," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2021, pp. 525–534.

[57] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. Le Traon, "Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 650–670, 2014.

[58] C. H. P. Kim, D. Marinov, S. Khurshid, D. Batory, S. Souto, P. Barros, and M. D'Amorim, "SPLat: Lightweight dynamic analysis for reducing combinatorics in testing configurable systems," in *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2013, pp. 257–267.

[59] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel, "A comparison of 10 sampling algorithms for configurable systems," in *International Conference on Software Engineering*, 2016, pp. 643–654.

[60] M. Mukelabai, D. Nešić, S. Maro, T. Berger, and J.-P. Steghöfer, "Tackling combinatorial explosion: A study of industrial needs and practices for analyzing highly configurable systems," in *International Conference on Automated Software Engineering*, 2018, pp. 155–166.

[61] X. Qu, M. B. Cohen, and G. Rothermel, "Configuration-aware regression testing: An empirical study of sampling and prioritization," in *International Symposium on Software Testing and Analysis*, 2008, pp. 75–86.

[62] T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou, and S. Pasupathy, "Do not blame users for misconfigurations," in *Symposium on Operating Systems Principles*, 2013, pp. 244–259.

[63] R. Cheng, L. Zhang, D. Marinov, and T. Xu, "Test-case prioritization for configuration testing," in *International Symposium on Software Testing and Analysis*, 2021, pp. 452–465.

[64] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "Opentuner: An extensible framework for program autotuning," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*, 2014, pp. 303–316.

[65] F. Canales, G. Hecht, and A. Bergel, "Optimization of java virtual machine flags using feature model and genetic algorithm," in *Companion of the ACM/SPEC International Conference on Performance Engineering*, 2021, pp. 183–186.

[66] J. Dong, Y. Lou, and D. Hao, "SRRTA: Regression testing acceleration via state reuse," in *International Conference on Automated Software Engineering*, 2021, pp. 1244–1248.

[67] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst, "Automatic test factoring for Java," in *International Conference on Automated Software Engineering*, 2005, pp. 114–123.

[68] P. Stratis and A. Rajan, "Speeding up test execution with increased cache locality," *Journal of Software Testing, Verification and Reliability*, vol. 28, no. 5, pp. 1–17, 2018.