Test Scheduling Across Heterogeneous Machines While Balancing Running Time, Price, and Flakiness

Hengchen Yuan*[1], Jiefang Lin*[1], Wing Lam[2], August Shi[1]
[1] The University of Texas at Austin, USA
{hcyuan,jiefang,august}@utexas.edu
[2] George Mason University, USA
winglam@gmu.edu

***Abstract*—Scheduling tests to run in parallel across different machines is an effective way to reduce overall test running time. Prior work has focused on scheduling tests across homogeneous machines, namely, machines all of the same configuration. However, using all the same configuration may not be the most cost-effective way to reduce test running time.**

We propose scheduling tests across machines with different configurations, namely heterogeneous machines. Doing so allows us to balance various factors, e.g., price, as tests may have similar running times on different machine configurations but result in drastically different monetary prices. Furthermore, there can be flaky tests that fail more often on different machine configurations, so scheduling them across heterogeneous machines gives better control over their flaky-failure rates. Our approach, GASearch, leverages genetic algorithms and a fitness function to balance running time and price to efficiently generate a heterogeneous machine configuration on which to run tests. We also model flaky-failure rate of tests on different machines within the fitness function as a factor of running time, where a failing flaky test would be rerun until it passes (or to a maximum number of runs) to confirm if it is a flaky failure, so we can balance all factors at once. We evaluate our approach on test suites from 24 modules in open-source Maven projects. Compared against baselines that schedule across homogeneous machines, we find that scheduling across heterogeneous ones can achieve a lower running time and price.

***Index Terms*—Test scheduling, heterogeneous machines, resource-dependent flakiness.**

I. Introduction

Testing is an important part of the software development process, but it can be very costly due to the large number of tests to run [1]–[5]. Test parallelization is an effective way to speed up testing, by scheduling the tests to run in parallel and distributed across different machines [6]–[12]. When a developer determines a set number of machines on which they can schedule tests, they typically consider the same configuration for each machine, i.e., machines with the same number of CPUs, RAM, and disk memory [13]. We consider scheduling tests across machines all with the same configuration as scheduling across *homogeneous machines*. Homogeneous machines may be easier to manage, given that all the machines are of the same configuration, but restricting to use only one configuration can limit test parallelization.

When scheduling tests, a developer needs to consider several factors: how many machines and which configurations to use. On one extreme, a developer may choose to optimize for *running time*, choosing the machines that allow the tests to run the fastest. However, these machines tend to be more expensive, so a developer will pay a high price. On the other hand, a developer may choose to optimize for *price*, opting for the cheapest machines on which to run tests, even though the tests can run longer. In addition, a developer also needs to consider the *flakiness of tests*. A *flaky test* is a test that can pass or fail when run on the same version of code [14, 15]. Developers rerun failing tests to check for flakiness, e.g., Google developers automatically rerun failing tests to see whether they are flakily failing [16], which adds to the running time. Prior work studied resource-affected flaky tests, whose flaky flaky-failure rates change depending on the resources, such as RAM, available on the machine on which tests are run [17]. Resource-affected tests add an extra factor when considering the machine configurations to use.

In this work, we propose to schedule tests across *heterogeneous machines*. Unlike homogeneous machines, we allow machines to have different configurations from each other. Heterogeneous machines allow for more flexibility in how tests can be scheduled so the scheduling algorithm can schedule tests to result in more optimal test running time, machine price, or flaky-failure rate.

We propose GASearch to schedule tests across heterogeneous machines. GASearch uses a genetic algorithm, with a fitness function that combines both test running time and machine price, with a weighting mechanism to control which factor has a bigger effect. Furthermore, we encode the flaky-failure rate of tests on different machine configurations into our fitness function by modeling their effect on test running time: the higher the flaky-failure rate of a test when run on a specific configuration, the more likely it is rerun when it fails, therefore increasing overall running time. Our fitness function models flaky-failure rate effect on running time using

*These authors contributed equally to this work.

the flaky-failure rate of each test and the cost of reruns (up to 10) to get a passing result if such tests were to fail. GASearch first uses a genetic algorithm approach to determine the heterogeneous machines on which to schedule the tests, and then uses a greedy algorithm to produce an allocation scheme of tests across those machines, where the greedy choice is to schedule a test on a machine that minimizes the fitness function. Developers interested in scheduling their tests across heterogeneous machines will need to provide an arbitrary number of test runs on configurations that are available to them along with the weights on how much the developers care about the running time and price. Once the optimal heterogeneous machines to use is obtained, developers can run tests using those machine configurations for future code changes, optimizing for their important factors.

We conduct an empirical study on the effectiveness of scheduling tests across heterogeneous machines. We evaluate GASearch on 24 modules from 22 open-source Maven projects, taken from a public dataset of test outcomes and their running times on different machine configurations [17,18]. We compare GASearch against two baselines that schedule tests on homogeneous machines: (1) the GitHub baseline that uses the same configuration used by the GitHub Actions continuous integration service [19], representing a common usage scenario where developers use the machines provided by GitHub and (2) the smart baseline that finds the optimal homogeneous machines [20]. We also compare against the random baseline that randomly chooses machine configurations before greedily allocating tests, allowing for a simple way of getting heterogeneous machines. We find that GASearch can achieve a lower running time and price over that of the baselines. When optimizing purely for price, it achieves a price that is on average 45% of the GitHub baseline and 81% of the smart baseline. When optimizing purely for running time, it achieves a running time that is on average 91% of the GitHub baseline and 99% of the smart baseline. GASearch also outperforms the random baseline on all fronts, but interestingly, the random baseline does outperform the other baselines when optimizing for price, on average. Note that GASearch can improve one metric at the expense of the other, i.e., improved running time comes at the cost of increased price. When we adjust the fitness function to better balance between the two metrics, we find that GASearch schedule tests such that the tradeoff is more favorable, e.g., having a slightly higher running time but a much larger reduction in price.

This paper makes the following main contributions:

- We propose scheduling tests across heterogeneous machines to provide more optimal test running time and machine price over homogeneous machines.
- We implement GASearch, a genetic algorithm approach to schedule tests across heterogeneous machines, balancing running time, price, and flaky-failure rate.
- We evaluate GASearch against baselines that schedule tests across homogeneous machines and a random baseline for heterogeneous machines. We find that GASearch can improve further upon running time and price over the baselines, and the tradeoff between the two factors is much better when balancing both within the fitness function, with both running time and price reduced against the baselines. Our artifact with experiment scripts and data is publicly available [21].

II. Example

Consider tests from the `javadelight/delight-nashorn-sandbox` project in our dataset: `TestInaccessible.test_file`, `TestIssue34.testIssue34_Scenario2`, and `TestMemoryLimit.test`; for convenience, we refer to them as `t1`, `t2`, and `t3`, respectively. Assume the following: (1) we want to schedule these tests to run across two machines, which can be of `C1` and `C2` configurations, each comprised of a different numbers of CPUs and amount of RAM (Section IV has more details on these configurations), (2) `t1`, `t2` and `t3` take 5.22s, 7.53s, and 5.00s, respectively, to run on `C1`, while the tests take 4.24s, 6.17s, and 4.58s, respectively, to run on `C2`, (3) `t3` is a flaky test that can flakily fail on both `C1` and `C2`, though with a different, non-zero flaky-failure rate on each. Developers tend to rerun tests that fail to check that they are flaky [3]. Test reruns are stopped once the test passes and it is clear that there is a flaky failure (and not a real fault).

Based on these assumptions, we can compute an expected running time for a test based on the expected number of times it should be rerun. To be more specific, suppose our strategy is to rerun a failed test as long as it keeps failing, up to 10 times. If the flaky-failure rate of `t3` on `C1` is 0.56, then the expected number of times it will be rerun is $\left(1 + \sum_{i=1}^{10} 0.56^i\right) = 2.268$[1]. As it takes 5.00s to run on `C1`, we can expect the running time for `t3` to be 11.34s on `C1`. The flaky-failure rate of `t3` on `C2` is 0.64, which leads to the expected number of times to rerun the test to be 2.722. As `t3` takes 4.58s to run on `C2`, the expected running time for `t3` on `C2` then becomes 12.47s.

If we restrict ourselves to using homogeneous machines where all machines are of the same configuration, the best option would be to use two `C2` machines, where we schedule tests `t1` and `t2` to one machine and `t3` to the other. The overall running time of tests scheduled across these machines is the longest running time of the tests on a single machine (as tests are run in parallel across different machines). The overall running time is 12.47s, as `t1` and `t2` take a combined $4.24 + 6.17 = 10.41$s to run on a `C2` machine and `t3` takes 12.47s to run on its own on the other machine.

However, we see that there can be a more optimal way to schedule tests to achieve a faster overall test running time if we use heterogeneous machines, where we can use different configurations for different machines. If we allow one machine to use configuration `C1`, while the other one uses `C2`, we can schedule test `t3` on the `C1` machine and put the remaining tests on the other machine. Scheduling tests this way results in an overall running time of 11.34s, which is the running time

[1]See Section III for detailed calculation.

of `t3` on `C1` (the other two tests continue to take a total of $4.24 + 6.17 = 10.41$s to run on the other machine).

III. Scheduling Tests across Heterogeneous Machines

We propose scheduling tests across heterogeneous machines, i.e., machines that can have different configurations from each other. The number of possible combinations of machines increases exponentially from homogeneous (all machines are of the same configuration), making the search for an optimal combination of machines much more expensive. We implement a genetic algorithm [22] to search for the heterogeneous machines on which to schedule tests, while optimizing for relevant metrics, such as running time, price, and flaky-failure rate. We refer to this approach as *GASearch*.

The input to GASearch is the set of tests, the number of machines available, a set of C machine configurations to select from, the time to run each test on each of the C configurations, and the flaky-failure rate of each test when run on each of the C configurations (computed by the number of times the test fails out of the number of times the test was rerun on a machine with the configuration). The output is an *allocation scheme*, which consists of a set of mappings of which tests to run on which machines, where each machine has a defined configuration and the number of unique machines matches the number specified in the input.

A. Genetic Algorithms

Genetic algorithms are metaheuristic search methods that draw inspiration from natural evolution and genetics to optimize solutions for complex problems [22]. In genetic algorithms, a population of candidate solutions is iteratively evolved towards better regions of the search space through selection, crossover, and mutation. To guide this search, genetic algorithms rely on a *fitness function* that measures the quality of each individual in the population, producing a *fitness value*. The overall goal is to create new individuals that improve upon the fitness value.

For our problem, we treat a combination of machines as an individual in the population, composed of different machine configurations. Further, we restrict the search to consider combinations of machines of length L, where we choose L from $\{1, 2, 4, 6, 8, 10, 12\}$. We essentially search for the optimal combination of machines of each length L and then report the most optimal one among them.

B. Initial Setup

We construct an initial population of N combinations of machines. This initial population contains C combinations of machines, each of length L, where each of these combinations of machines contains L identical configurations, corresponding to each of the possible C available configurations. Essentially, we start with all possible homogeneous combinations of machines of length L. We want to include these combinations of machines initially, because we want to eventually construct a heterogeneous combination of machines that improves upon the possible homogeneous combinations of machines. If we cannot find a heterogeneous combination of machines that improves upon the homogeneous combinations of machines, the best homogeneous combination of machines will be outputted as the most optimal solution. Aside from all the homogeneous combinations of machines, we fill in the remaining initial population with randomly generated combinations of machines of length L taken from the C configurations.

C. Fitness Function

We define a fitness function that evaluates the effectiveness of a given allocation scheme for a combination of machines. Our fitness function models both the overall running time of tests scheduled across the combination of machines and the monetary price involved with running these machines:

$$Fitness(A) = \alpha\beta Time_{para}(A) + (1-\alpha)Price(A) \quad (1)$$

where $Time_{para}$ and $Price$ are functions that model the overall test running time and monetary price, respectively, for a given allocation scheme A representing the mapping from tests to machines of specific configurations. We also define a parameter α to control the weight that running time and price has towards the overall fitness value. Specifically, when α is 0, we optimize for minimum price only, while when α is 1, we optimize for minimum running time only. As our goal is to minimize running time and price, the lower the fitness value, the better. β is a scaling factor defined in Equation (7).

Running time and flaky-failure rate. We first define how to compute the running time of tests on a single machine. Let $testtime(t, m)$ be a function that returns the running time for test t on a specific machine m. Normally, all tests should be run once on the machine. However, if the test fails, developers rerun the test to check whether the failure is a flaky failure [16], rerunning up to some maximum r number of times. If the test passes, then the developer is sure that the failure is a flaky failure and stops rerunning the test. If the flaky-failure rate of the test is high, then the likelihood of needing to rerun goes up, resulting in a higher running time.

To model the effect of flaky-failure rate on running time, we compute an *expected running time* per test based on the scenario that the developer reruns the test if it fails:

$$exptime(t, m) = (1 + \sum_{i=1}^{r-1} fr(t, m)^i) * testtime(t, m) \quad (2)$$

where $fr(t, m)$ is the flaky-failure rate of test t on machine m. We also set r to be 10 in this study. The summation term $\sum_{i=1}^{r-1} fr(t, m)^i$ encapsulates the cumulative probability of flaky failures occurring across test reruns. Each $fr(t, m)^i$ represents the probability of the test failing i times consecutively. We multiply by the test's running time on the machine to compute the expected running time.

The overall running time of all tests on a machine m is therefore the sum of $exptime(t, m)$ for all tests t on m:

$$MachTime(A, m) = \sum_{t \in tests(A,m)} exptime(t, m) \quad (3)$$

where $tests(A, m)$ returns the tests scheduled to machine m in the allocation scheme A.

Overall running time. We define the function $Time_{para}$ to take as input the allocation scheme A and compute the time to run all the tests scheduled across the machines represented in A in parallel (tests scheduled on different machines can be run in parallel):

$$Time_{para}(A) = \max_{m \in machines(A)} MachTime(A, m) \quad (4)$$

where $machines(A)$ represents the machines on which A operates. Essentially, this running time is the longest time needed to run the tests scheduled to a single machine by A.

Another consideration is whether there should be a setup time applied per machine. This setup time corresponds to needing each machine to separately rebuild the project code, which can vary between different machine configurations. Currently, our running time metrics assume a use scenario where the developers do not explicitly build code on each machine (especially redundant given that each machine has the same code on which tests will run), but rather the code is built "offline" and quickly uploaded to each machine, minimizing setup time. If we consider setup time as part of the running time, we can compute the running time per machine as:

$$MachTime_{st}(A, m) = ST(m) + \sum_{t \in tests(A,m)} exptime(t, m) \quad (5)$$

where $ST(m)$ represents that setup time on machine m. We evaluate both with and without setup time in our evaluation.

Price. The price of running tests for a given allocation scheme involves the time needed to run the tests on the machines.

$$Price(A) = \sum_{(t,m) \in A} price(t, m) \quad (6)$$

where $price(t, m)$ is the price of running test t on m.

Scaling factor. We observe that the running time value is consistently much larger than price in absolute terms. As such, weighting the two together in a single fitness function will be biased towards running time. We introduce a scaling factor β to better balance the two. For our evaluation, we compute β as the average price-to-time ratio of all tests on all configurations in our dataset, namely:

$$\beta = \frac{\sum_{(t,m) \in Dataset} price(t, m)}{\sum_{(t,m) \in Dataset} exptime(t, m)} \quad (7)$$

Algorithm 1: Greedy Tests Allocation

```
   Input : α, β, tests, machine_list,
           tests_attribute_mapping
   Output: allocation
1  allocation ← ∅
2  foreach t in sorted(tests) do
3      min_fitness ← inf
4      min_machine ← None
5      foreach m in machine_list do
6          temp_allocation ← allocation.add(t, m)
7          fitness ← fitness(α, β, temp_allocation,
             tests_attribute_mapping)
8          if fitness < min_fitness then
9              min_fitness ← fitness
10             min_machine ← m
11         end
12     end
13     allocation ← allocation.add(t, min_machine)
14 end
15 return allocation
```

D. Allocating Tests

The fitness value is defined w.r.t. some allocation scheme, namely which tests are scheduled on which machines. This allocation scheme problem is similar to the Multiprocessor Scheduling Problem, known to be NP-hard [23,24]. We utilize a LPT (Longest Process Time)-based greedy algorithm [25] to perform the allocation scheme.

Algorithm 1 shows our greedy algorithm. Given a set of tests and the machines on which to schedule those tests, we sort the tests in descending order by their running time on their fastest machine (Line 2). We iterate through the tests in this order and compute the fitness value after trying to schedule the test on each machine (Lines 3-4). We schedule the test on the machine that results in the lowest fitness value (Lines 5-13) before moving to the next test. We output the final allocation scheme after processing all tests.

E. Search Operators

In each iteration of the genetic algorithm, we compute the fitness value for each individual, i.e., combination of machines (and its corresponding allocation scheme), in the population. We first apply a selection operator that selects the top 35% of the individuals based on their fitness values. We then apply a crossover operator on the selected individuals by randomly pairing up the individuals and crossing the machines contained in each one. For each pair, we set a random crossover point to swap machines between the parents, creating two new children individuals from those machines taken from the parents. We continue applying this crossover operator on random pairs of selected individuals until achieving a new population of size N. We set $N = 100$ in our experiments. Next, we apply a mutation operator on each new individual. The mutation operator iterates through each machine within and mutates the

machine to a different machine with probability $1/L$ (the length of the combination of machines).

After applying selection, crossover, and mutation operators to generate a new pool of N individuals, we have completed one iteration of the genetic algorithm. We repeat for I iterations. After the final iteration, we take the individual and corresponding allocation scheme that provides the best fitness value and report them as the final output.

F. Implementation

To implement GASearch, we use the DEAP framework [26], which provides support for quickly implementing genetic algorithms. We use the built-in data structures and APIs from DEAP to encode our problem. We configure the search to run for 50 generations, i.e., setting $I = 50$.

IV. Evaluation Setup

We answer the following research questions:

- **RQ1**: How does scheduling tests on heterogeneous machines compare against homogeneous machines?
- **RQ2**: How does changing the weight factor affect the tradeoffs between running time and price?
- **RQ3**: What are the flaky-failure rates from using GASearch's allocation schemes?
- **RQ4**: How does GASearch's allocation scheme compare against a brute-force search that finds the optimal allocation scheme?
- **RQ5**: How well would GASearch perform if using just a subset of test data?

We address RQ1 to see whether using heterogeneous machines can achieve better test running time and machine price than using homogeneous machines. We address RQ2 to show the effects of the weight factor in obtaining better tradeoffs between running time and price. We address RQ3 to show the expected chance of a build failure with a GASearch allocation scheme. While we can encode the flaky-failure rate of tests as rerunning failing tests, we still want to show the probability of the test suite flakily failing. We address RQ4 to see whether GASearch is more effective than a brute-force approach at finding the most optimal allocation scheme. Finally, we address RQ5 to see whether GASearch performs just as well if it is guided by just a subset of test information, i.e., running time and flaky-failure rate, collected from a fewer number of runs across different configurations. This RQ helps check the practicality of GASearch in the scenario where a developer does not have much data for guiding GASearch.

A. Dataset

We use a publicly available dataset taken from prior work by Silva et al. on evaluating test flakiness when run on different machine configurations [17, 18][2]. Their data includes the running times of each test on 12 different machine configurations. A machine configuration is defined by the number of CPUs and the amount of RAM available to the machine. Table I shows a description of each of these configurations. The dataset also provides the flaky-failure rate of each test on each machine configuration, i.e., the number of times each test fails when rerun up to 300 times on each machine configuration.

TABLE I
Machine configurations from [17]. "# CPU" with non-integer value means a core is shared across multiple tasks [27]. Hourly costs are specified on AWS Fargate [28]

ConfigID	# CPU	Mem (GB)	Price (USD/Hour)
C1	0.1	1	0.002548
C2	0.1	2	0.003881
C3	0.25	2	0.005703
C4	0.5	2	0.008739
C5	0.5	4	0.011406
C6	1	4	0.017478
C7	1	8	0.022812
C8	2	4	0.029622
C9	2	8	0.034956
C10	2	16	0.045624
C11	4	8	0.059244
C12	4	16	0.069912

Starting with the 32 modules[3] from 27 projects in Silva et al.'s dataset, we filter out modules with insufficient data. Silva et al. collected their data by running the tests in each module for 30 "rounds", where in each round they run the tests 10 times on each of the 12 configurations. We observe that for some modules, the dataset has "incomplete" rounds, i.e., rounds without data for all tests or all tests are not run 10 times. We filter out projects with fewer than 10 complete rounds for all tests; we observe the remaining modules have at least 27 complete rounds of data. We notice that the dataset has some tests duplicated across different modules, so we filter out modules that contain a subset of the tests from other (parent) modules. Finally, we obtain 24 modules across 22 projects.

Table II shows a breakdown of the modules. Column "ID" is a module ID we use to refer to the module in future tables and figures, "Project" is the name of the project in the form of a GitHub username/repository identifier, "Module" is the name of the module as marked in the dataset. "# Test" is the number of tests in the module, and "Avg. Running Time (s)" is the average test suite running time across all configurations. "Flaky-Failure Rate Range (%)" is the range of flaky-failure rate. Note that each test may have different flaky-failure rates on different configurations. We say a test is flaky if it has a non-zero flaky-failure rate on at least one configuration. The minimum value of the range refers to the lowest flaky-failure rate of all the flaky tests of this module on all the configurations. The maximum value is similarly the highest flaky-failure rate for a flaky test across all configurations.

B. Baselines

We compare GASearch against three baselines. The first baseline is what we call the *GitHub baseline*, where we use only the machine configuration that matches the standard available machines for the GitHub Actions continuous integration service [19], namely a configuration that uses 2 CPUs and

[2]Obtained June 2023.

[3]A Maven project may contain multiple modules, each with its own tests.

TABLE II
STATISTICS OF THE PROJECTS AND MODULES IN OUR EVALUATION

ID	Project	Module	# Test	Flaky-Failure Rate Range (%)	Avg. Running Time (s)
M1	activiti/activiti	.	2047	0.0 - 91.1	0.44
M2	alibaba/fastjson	.	4459	0.0 - 09.0	0.02
M3	apache/commons-exec	.	55	0.0 - 03.0	0.54
M4	apache/httpcore	.	713	0.0 - 12.8	0.04
M5	apache/incubator-dubbo	dubbo-remoting-netty	14	0.0 - 10.7	3.44
M6	apache/incubator-dubbo	dubbo-rpc-dubbo	66	0.0 - 05.9	2.24
M7	davidmoten/rxjava2-extras	.	390	0.0 - 01.0	0.22
M8	elasticjob/elastic-job-lite	.	560	0.0 - 00.0	0.09
M9	flaxsearch/luwak	luwak	202	0.0 - 04.0	0.16
M10	fluent/fluent-logger-java	.	18	0.0 - 41.0	2.61
M11	javadelight/delight-nashorn-sandbox	.	79	0.0 - 63.7	1.53
M12	jknack/handlebars.java	.	412	0.3 - 03.4	0.03
M13	joel-costigliola/assertj-core	.	6267	0.0 - 80.3	0.00
M14	kagkarlsson/db-scheduler	.	25	0.0 - 29.0	0.60
M15	kevinsawicki/http-request	.	163	0.0 - 01.0	0.02
M16	nationalsecurityagency/timely	server	144	0.3 - 03.7	0.31
M17	ninjaframework/ninja	.	305	1.3 - 90.0	0.17
M18	orbit/orbit	.	83	0.0 - 34.7	0.52
M19	qos-ch/logback	.	863	0.0 - 84.5	0.24
M20	spring-projects/spring-boot	.	1689	0.0 - 00.7	0.21
M21	square/retrofit	retrofit	297	0.0 - 00.3	0.04
M22	square/retrofit	retrofit-adapters.rxjava	80	0.0 - 00.0	0.07
M23	wro4j/wro4j	wro4j-extensions	308	0.0 - 00.7	0.81
M24	zxing/zxing	.	345	2.1 - 05.5	0.61
	Average		816	0.2 - 24.0	0.62

8GB of RAM (effectively C9 from Table I). We choose this baseline to match the scenario where developers of open-source GitHub projects use only those available machines. We use the same number of machines that GASearch proposes for the same module, to ensure fair comparison, because the GitHub baseline does not search for the number of machines.

The second baseline is what we call the *smart baseline*, where we search for the homogeneous machines based on the same fitness function that GASearch uses. Due to using the same configuration for all machines, we can find the optimal homogeneous machines by trying all possibilities and choosing the one that produces the best fitness value.

The third baseline is what we call the *random baseline*, where we randomly choose the heterogeneous machines. We use this baseline as a simple and straightforward way to schedule tests across heterogeneous machines as comparison, to show whether a more sophisticated solution like GASearch is needed and whether just using heterogeneous machines can be better than using homogeneous machines. After randomly choosing the machines, we use the same greedy algorithm as GASearch to map tests to the specific machines.

For RQ4, we define a brute-force search that finds the optimal solution based on the fitness function. The brute-force search explores all possible combinations of machines. With L machines and C configurations, there are L^C possibilities. With this exponential search space, we restrict brute-force search to consider up to six machines. We apply the same restriction to GASearch for comparison. We use the same greedy algorithm to schedule tests across machines for both.

C. Metrics

We compare GASearch against each baseline by measuring the ratio of GASearch's allocation scheme's running time and price over a baseline's allocation scheme's running time and price, respectively. More specifically, we compute the running time ratio as: $\frac{Time_{para}(A_G)}{Time_{para}(A_B)}$. Similarly, the price ratio is: $\frac{Price(A_G)}{Price(A_B)}$. A_G represents the allocation scheme that GASearch produces and A_B represents the allocation scheme that the baseline produces. A ratio of 1 indicates that GASearch produces an allocation scheme that achieves the same running time or price as the baseline, whereas a value higher than 1 indicates that GASearch's result is worse than the baseline and a value lower than 1 indicates that GASearch's result is better than the baseline.

As optimizing for one metric can come at the expense of the other, e.g., optimizing for lower running time can lead to a higher price, we introduce a new metric to capture whether the tradeoff between the two is fair. For an allocation scheme A_G produced by GASearch and another allocation scheme A_B produced by a baseline, we compute the tradeoff as a product of the two running time and price ratios comparing the two:

$$Tradeoff(A_G, A_B) = \frac{Time_{para}(A_G)}{Time_{para}(A_B)} \times \frac{Price(A_G)}{Price(A_B)} \quad (8)$$

For example, if the running time achieved by A_G is twice as fast as that achieved by A_B (i.e., the ratio is 50%) but comes at twice the price, that tradeoff is somewhat fair, and it results in a tradeoff value of 1. Having a value greater than 1 would indicate that the tradeoff is unfair, e.g., the running time is

twice as fast but costs relatively more, while a value less than 1 would indicate that the tradeoff is better, e.g., the running time is twice as fast yet requires less than twice the price.

For RQ2, we evaluate the effects of changing the weight parameter α in the fitness function, going from 0 to 1 in steps of 0.05. At each value of α, we measure the same running time and price ratios, and compare GASearch against the GitHub baseline and smart baseline, where smart baseline uses the same fitness function with the same α.

For RQ3, we measure the overall build failure probability for an allocation scheme based on the flaky-failure rates of each test on the machines on which they are scheduled. Essentially, we are interested in knowing the likelihood of the entire build failing, which occurs when at least one test fails. We compute this likelihood by computing the probability of all tests passing and subtracting that probability from 1. Given an allocation scheme A, the build failure probability $BuildFail(A)$ under this allocation scheme is:

$$BuildFail(A) = 1 - \prod_{(t,m) \in A} (1 - fr(t,m)) \tag{9}$$

GASearch may not consistently compute the same allocation schemes for the same input information, due to the nondeterministic nature of the genetic algorithm. As such, we rerun GASearch five times to obtain different allocation schemes. We compute the metrics comparing each allocation scheme against the baselines and report the average across the allocation schemes for each metric in our evaluation. For the random baseline, we also rerun it five times to obtain multiple allocation schemes, and we compare each GASearch allocation scheme against each one, reporting a final average.

V. Evaluation

A. RQ1: Heterogeneous vs Homogeneous Machines

Figure 1 compares GASearch against the GitHub baseline, smart baseline and random baseline for each module in terms of running time and price, computed considering no setup time per machine (Section III-C). For each module, we show the running time ratio and price ratio compared against the three baselines, with a line representing the average ratios across all modules. The top figure shows the results when optimizing for running time only ($\alpha = 1$) while the bottom figure shows the results when optimizing for price only ($\alpha = 0$).

When optimizing for running time only, GASearch outperforms the GitHub baseline in 23 modules, with the lowest running time ratio of 0.33 and an average running time ratio of 0.91. GASearch outperforms the smart baseline in 10 modules, meaning 14 modules work best with homogeneous machines. The lowest ratio running time ratio is 0.85, and the average running time ratio is 0.99. GASearch outperforms the random baseline in 22 modules, with the range of running time ratio from 0.41 to 1 and an average running time ratio of 0.83.

When optimizing for price only, all price ratios are less than 1 for all baselines. For the GitHub baseline, the range of price ratios is from 0.04 to 0.84, with an average ratio of 0.45. For the smart baseline, the range of price ratios is from 0.54 to 1.00, with an average ratio of 0.81. For the random baseline, the range of price ratios is from 0.61 to 1.00, with an average ratio of 0.88. Note that the random baseline results in better price than the other two baselines, suggesting that simply using heterogeneous machines, even if determined randomly, can result in better price than using homogeneous machines.

In general, GASearch does not improve running time as much over the baselines as compared with price. We find that tests in a module tend to always run faster on expensive configurations. Further, the difference in running times of individual tests does not differ too much across configurations. As we measure the parallel running time, this metric is largely dominated by the tests with the longest running times. For example, in module M8, the running time of test `ZookeeperRegistryCenterInitFailureTest-.assertInitFailure` is about 10s on each configuration, and when using more than 4 machines, the sum of the running time of all tests on the other machines is less than the running time of this one test. In this case, GASearch cannot produce a better heterogeneous solution, because there is no change to the parallel running time when considering different configurations for other machines, as those other tests matter very little. When optimizing for running time, the homogeneous machines tend to remain the best. On the other hand, when optimizing for price, there is larger variance in price between tests across different configurations, allowing for more space to explore for reducing price.

Figure 2 also compares GASearch against the baselines, but with the scenario of including a setup time per machine (Section III-C). When optimizing for running time only, GASearch achieves improved running time over the GitHub baseline across all modules, but the average running time ratio is 0.92. Compared against the smart baseline, the average running time ratio is 1.00; most modules have a running time ratio of 1. For the random baseline, the average running time ratio is 0.35, much worse than without setup cost.

When optimizing for price only, GASearch performs better on all modules with the average ratio of 0.56 compared to the GitHub baseline. When compared to the smart baseline, the average price ratio is 0.97, where once again most modules have a price ratio of 1. GASearch and the smart baseline commonly propose using just one or two machines to reduce price, so they tend to propose the same machines. Compared to the random baseline, the average price ratio is 0.57.

In projects with tests that run relatively fast compared to machine setup time, any deviations in their running time is overshadowed by the machine setup time. As such, the best homogeneous and heterogeneous machines ultimately end up with the same running time and price, with the setup time being the most important factor. The lower the setup time, the more relevant it becomes to schedule tests across heterogeneous machines, to the limit of having no setup time (effectively copying instead of rebuilding code on machines). If a developer is running tests in this kind of scenario, then they would benefit from using heterogeneous machines. For

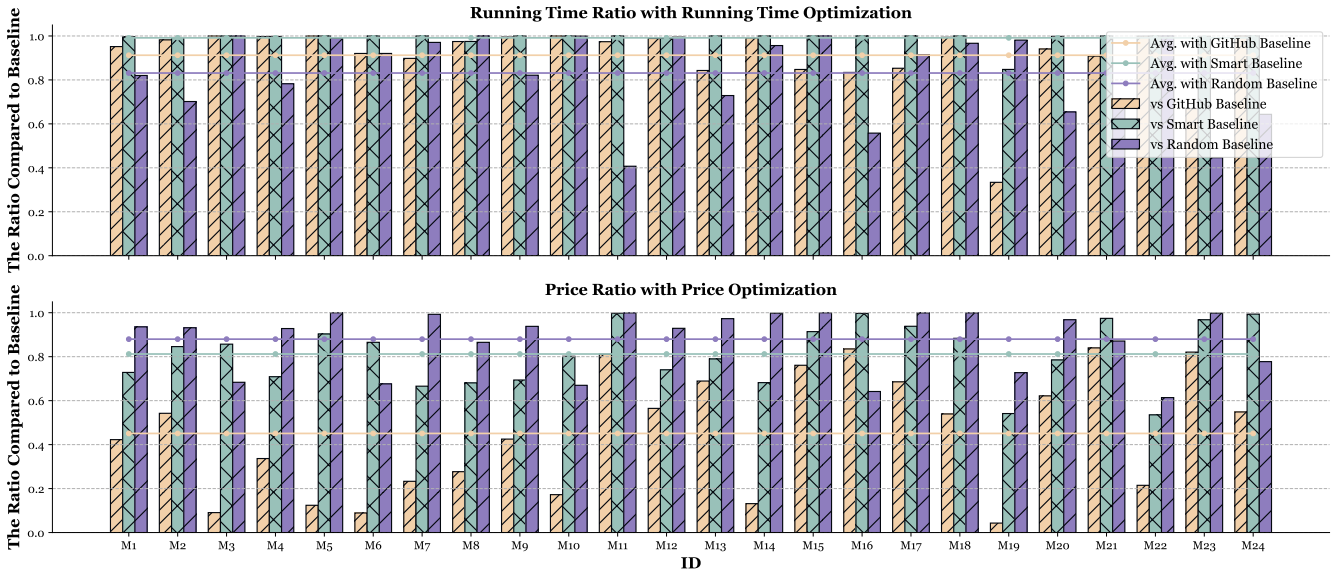

Fig. 1. Comparing GASearch running time and price against baselines, without setup time.

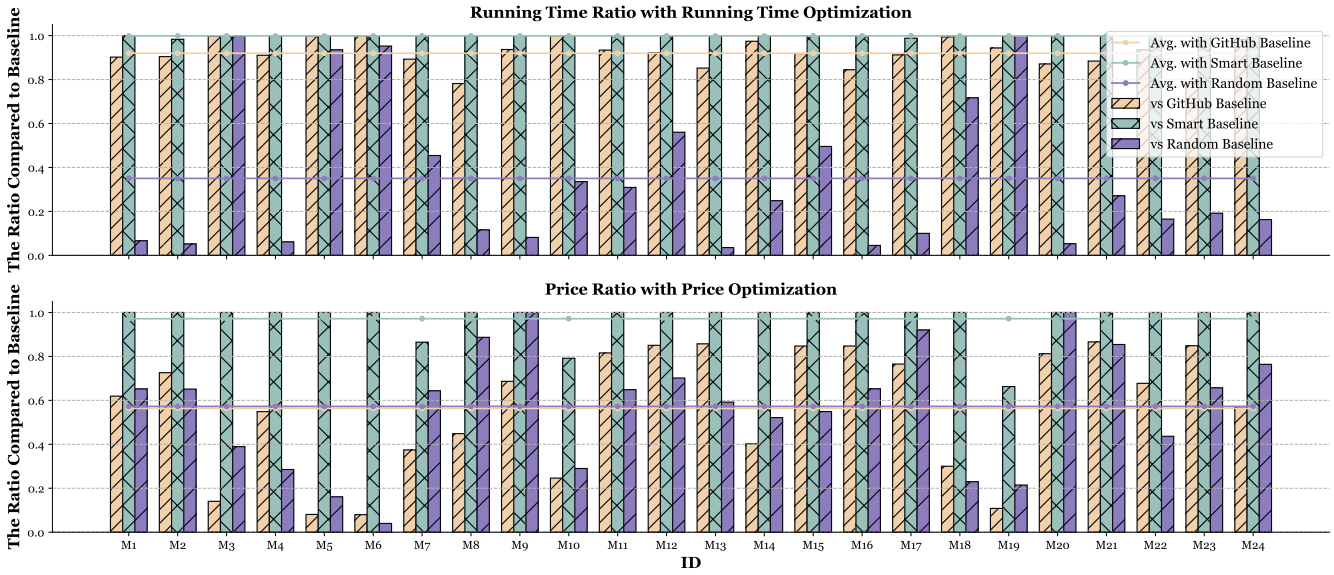

Fig. 2. Comparing GASearch running time and price against baselines, with setup time.

subsequent RQs, we show only the results when setup time is not included, to illustrate further this scenario and how heterogeneous machines may help.

B. RQ2: Effect of Fitness Function Weight

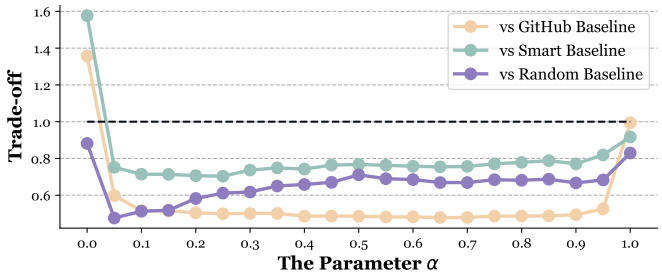

Fig. 3. Tradeoff value as α changes.

A change in the weight parameter α allows GASearch to consider both factors simultaneously, leading to better tradeoffs. Figure 3 shows the change in tradeoff as the weight parameter α changes. At $\alpha = 0$, i.e., optimizing for price only, the tradeoff is very poor; the tradeoff of GASearch's allocation scheme is 1.36 compared to the GitHub baseline, 1.58 compared to the smart baseline, and 0.88 compared to the random baseline. The tradeoff is close to 1 at the other extreme of $\alpha = 1$, matching our RQ1 findings that the running time ratio between GASearch and the baselines is close to 1.

When adjusting for α to not be 0 or 1, i.e., combining both factors, the average tradeoffs across modules become less than 1. Comparing against the GitHub baseline, we observe that the minimum tradeoff is 0.48 when α is 0.65. GASearch achieves a running time ratio of 0.99, while saving half of the cost: the price ratio is only 0.47 of the GitHub baseline.

Comparing against the smart baseline, the minimum tradeoff

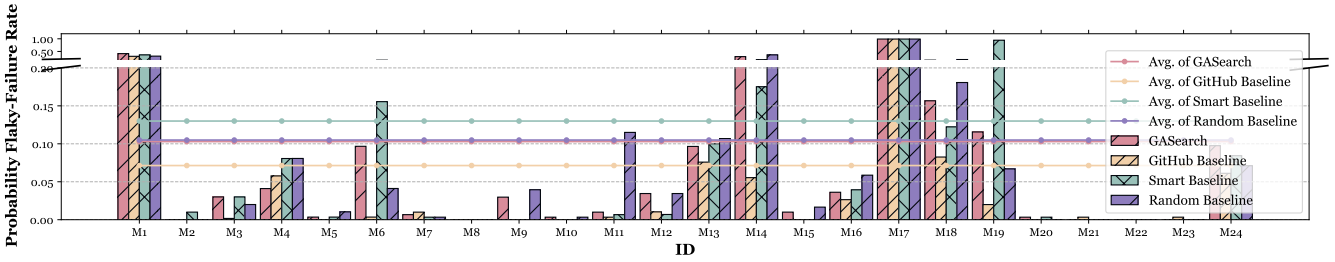

Fig. 4. Comparison of the probability of build failure.

is 0.70 when α is 0.25. GASearch achieves a running time ratio of 0.86, with a price ratio of only 0.81. Both ratios being less than 1 show how GASearch can achieve both faster running time and lower price at the same time, indicating a good tradeoff between these two factors.

Comparing against the random baseline, the tradeoff is at its minimum of 0.48 when the α is set to 0.05. GASearch achieves a running time ratio of 0.53 and a price ratio of just 0.89, an advantageous tradeoff between these two factors.

These results show that, in general, at the extreme values of α (when GASearch optimizes only for one metric), GASearch can achieve a larger improvement for the metric being optimized at the detriment of the other one, and the tradeoff may not be worth it. If a developer cares solely about one metric, then using these extreme values of α can be beneficial for them. However, if they need a good balance and are concerned whether a certain running time improvement is worth the price, they can tune the parameter α to a better value to achieve a better tradeoff. Given that the tradeoff is similar across all α values between 0 and 1, but the best α to use is different among the baselines, we choose to evaluate the subsequent RQs using only $\alpha = 0.5$ for ease of presentation.

C. RQ3: Effect of flaky-failure rate

Figure 4 shows the build failure probability when using the machines proposed by GASearch, the GitHub baseline, the smart baseline, and the random baseline across all modules, with average build failure probabilities of 0.103, 0.071, 0.130 and 0.105, respectively. Most modules have build failure probability less than 0.1. Modules with high build failure probability have individual tests with high flaky-failure rate across all configurations, e.g., in module M17, a test has a flaky-failure rate of 0.9 on every machine, leading to nearly 0.99 build failure probability for any allocation scheme.

Overall, the GitHub baseline proposes machines with the lowest average build failure probability, using relatively more expensive machines that likely have fewer flaky failures. GASearch and the smart baseline encode the flaky-failure rate of each test in the fitness function as a chance to rerun. By balancing these factors together, the fitness function allows an acceptable chance of flaky failures if the final tradeoff is reasonable, leading to higher build failure probabilities. GASearch, on average, achieves a lower build failure probability than the smart baseline. Interestingly, the random baseline achieves a similar build failure probability as GASearch.

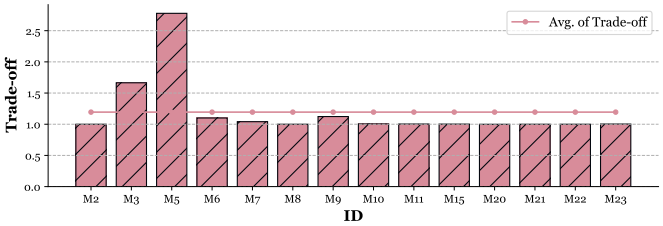

Fig. 5. The tradeoff between normal GASearch and when forcing flaky-failure rate = 0.

Forcing a low build failure probability may not result in good reductions of running time or price, leading to lower tradeoffs as well. We conduct an experiment where we modify GASearch to only schedule tests on machines on which they have a flaky-failure rate of 0%. However, some modules have tests that have a non-zero flaky-failure rate on *every* configuration, meaning it is impossible to have a 0% build failure probability. For this experiment, we use only the 14 modules that can have a 0% build failure probability.

Figure 5 shows the tradeoff between GASearch and 0-flaky-failure rate GASearch. We compute tradeoff using GASearch as the baseline. All modules' tradeoffs are greater than or equal to 1, with an average of tradeoff of 1.19, meaning enforcing a 0% flaky-failure rate per test makes the overall results worse. However, most modules have tradeoffs close to 1, suggesting that GASearch produces allocation schemes that naturally maintain a 0% build failure probability. On the other hand, we observe cases where the tradeoff is much worse, e.g., M5 has a test that maintains 0% flaky-failure rate on just one configuration, restricting the approach to schedule the test on that configuration, but it gives a bad tradeoff.

Only GASearch produced an allocation scheme with a 0% build failure probability for all 14 modules; the smart baseline can do so only for 13. In that one module, there are two tests that can achieve a 0% flaky-failure rate, but only on two separate configurations. As such, it is impossible to choose just one configuration for all tests and maintain an overall 0% flaky-failure rate. on homogeneous machines while maintaining a 0% flaky-failure rate. This example demonstrates another benefit of scheduling tests across heterogeneous machines, giving flexibility in ensuring tests have a 0% flaky-failure rate.

D. RQ4: Comparing GASearch vs. Brute-Force

Table III shows the time to run GASearch and the brute-force search for each module (α= 0.5). The table also shows the ratio of the fitness values of the best allocation schemes

TABLE III
COMPARING GASEARCH AGAINST BRUTE-FORCE SEARCH

ID	*Fitness Value When* $\alpha = 0.5$ Calculation Time (s) GASearch	Brute-Force	Ratio Fitness Value
M1	47.33	650.41	1.00
M2	113.90	864.52	1.00
M3	1.52	478.73	1.00
M4	19.08	528.19	1.00
M5	0.61	479.49	1.00
M6	1.55	470.58	1.00
M7	9.90	500.51	1.00
M8	14.57	519.32	1.00
M9	5.24	490.40	1.00
M10	0.74	472.51	1.00
M11	1.79	476.38	1.00
M12	10.70	504.43	1.00
M13	106.20	1042.12	1.00
M14	1.01	471.17	1.00
M15	3.31	488.55	1.00
M16	3.04	495.45	1.00
M17	7.24	495.40	1.00
M18	2.41	474.58	1.00
M19	23.52	543.88	1.00
M20	37.49	618.51	1.00
M21	7.91	494.49	1.00
M22	2.56	476.08	1.00
M23	5.57	496.19	1.00
M24	6.20	495.79	1.00
Average	18.06	542.82	1.00

produced by GASearch and the brute-force search. As they both optimize for the same fitness function, we want to see how close their solutions' fitness values compare to each other.

We see that GASearch achieves close to the same fitness value as brute-force search (less than 0.01 difference), suggesting that GASearch's solution is optimal for the fitness function. GASearch runs much faster compared against brute-force search, needing only about 3% of the time that brute-force search needs. Also, recall that we could not have brute-force search parallelize on more than six machines (Section IV-B).

E. RQ5: Effect of Guidance Data

Figure 6 illustrates the tradeoff between one-round data guided GASearch and full-data guided GASearch (full-data guided GASearch is the baseline). For 16 modules, we see that the tradeoff is close to 1 (less than 0.1 difference), suggesting that using just one round of data (i.e., 10 reruns) can produce solutions with tradeoffs just as good as when using all data, so a developer may not need to spend as much time collecting data as we did before they can schedule tests.

On average, the tradeoff between one-round data guided GASearch and full-data guided GASearch is 1.53. We observe some modules with very bad tradeoffs. They contain many tests whose running times and flaky-failure rates differ greatly in a single round of data compared against using all.

VI. DISCUSSION

This study delves into the implications of using heterogeneous machines for test scheduling, focusing on running time, price, and flaky-failure rate. This approach still faces similar challenges faced by traditional strategies that use homogeneous machines. As code evolves, if test running times change substantially or flaky-failure rate changes, the allocation scheme may become worse. We assume such factors remain stable, so the same allocation scheme can be reused for a period of time. If there are substantial changes, e.g., newly added tests, or enough accumulated changes over time, developers can re-collect running information for the tests substantially affected by changes. Developers can then use this new running information to recompute the allocation scheme.

Our current evaluation assumes no order-dependent tests, i.e., tests whose outcome depends on other tests [14, 29]. We assume test dependencies will be provided. Similar to Lam et al.'s work on order-dependent-test-aware regression testing techniques [30], we can group tests with their dependencies as one "unit" and schedule them together.

A technique to schedule tests across heterogeneous machines requires collecting test running information across various machine configurations a priori. The smart baseline also needs to collect that information despite using homogeneous machines (it needs to know which configuration to use for all tests), whereas the GitHub baseline only needs to collect this information for tests on the one configuration it considers. Regardless, collecting running information can be expensive, and we imagine developers can run tests across different machine configurations during off-hours when they are not actively developing (e.g., during the weekend). This upfront cost is amortized by future savings in running time and price from using the determined optimal heterogeneous machines. We can estimate at what point this upfront cost is worth it. For example, if we consider just the price, we can compute the price of collecting running information across the 12 machine configurations by running the tests on all configurations 10 times (the fewer reruns, as evaluated in RQ5). The GitHub baseline will require a lower cost, simply rerunning the tests on the same machine configuration. As GASearch proposes using heterogeneous machines that result in a lower price than the GitHub baseline, we can compute after how many test runs in the future will GASearch result in a lower price than the GitHub baseline, i.e., if we let R be the number of test runs when the two have the same price, we solve for R in the equation $R * GitHubPrice + GitHubSetupPrice = R * GASearchPrice + GASearchSetupPrice$, where $GitHubPrice$ is the price from using the GitHub baseline proposed homogeneous machines, $GASearchPrice$ is the price from using the GASearch proposed heterogeneous machines, and the various $*SetupPrice$ are the prices needed to collect running information. If we use the average values for each of these variables we collected from our study, we estimate R to be around 151, meaning a developer should expect to see that using GASearch to produce heterogeneous machines pays off over the GitHub baseline after 151 test runs in the future. Given that developers in companies are frequently making changes that require running tests, e.g., Microsoft reports hundreds of builds per day for some projects [31], developers can very quickly break even and save.

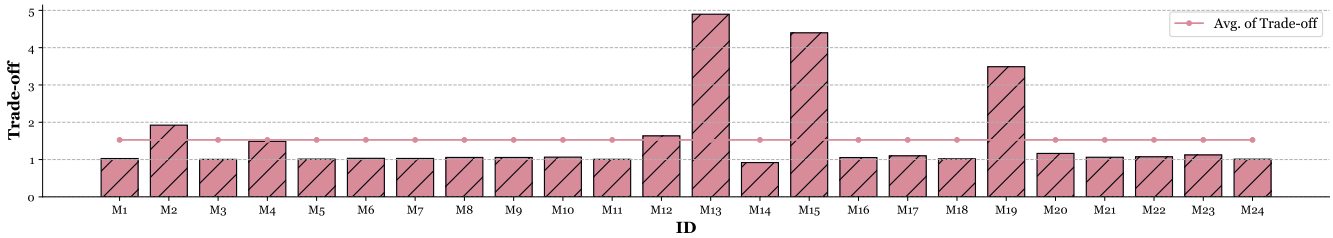

Fig. 6. The tradeoff between one-round data guided GASearch and full-data guided GASearch.

VII. Threats to Validity

The results we report may not generalize beyond the projects and tests on which we evaluate. We use data collected by prior work [17, 18], including test running time and flaky-failure rate data across different machine configurations, and we filter further to obtain subjects on which we can fairly evaluate, namely those with enough complete test data.

We limit GASearch to run for only 50 generations of improvement, which is more than past software engineering tasks [32, 33] explored. Running more generations may lead to a more optimal solution w.r.t. the fitness value. As such, the numbers we report are a lower limit on how effective GASearch can be. We later conducted an experiment running for more generations (100), finding solutions with similar results. We designed our algorithm for a minimalist approach to reduce randomness in the process, e.g., we choose an elitist selection operator where only the best individuals participate in crossover, which has been found effective for certain domains [34, 35]. When we use GASearch five times per module to compute different solutions (Section IV), the average deviation in the fitness value between them was less than 1e-6. When we experimented with using selection operators like Roulette-wheel selection [36] to allow for some chance of using worse individuals for crossover, the difference in fitness value was less than 1e-3. Future work can explore other variations in genetic algorithms to obtain better solutions.

GASearch may not produce as optimal solutions as brute-force search. However, given that brute-force search is already much more expensive than GASearch with six machines (Section V-D) and will become even more expensive with more machines, GASearch is clearly more practical.

VIII. Related Work

Silva et al. recently studied the effects of different computational resources on test flakiness [17]. They ran tests from open-source projects on 12 different machine configurations 300 times to measure the flaky-failure rate of each test on different configurations. They find that the flaky-failure rate per test changes based on the configuration. The amount of resources available on the machine affects the test outcome, where generally tests are more likely to pass when run on machines with more resources. We build upon their work by considering this varying flaky-failure rate depending on machine configuration as a factor in our problem of scheduling tests. We directly use their dataset to evaluate our approach [18].

Developers commonly schedule tests across different machines for faster testing. For example, Microsoft relies on its internal distributed build system CloudBuild to run tests across different machines in their cloud environment [7], scheduling tests the moment they need to be run and machines are available. There has been work in improving test running time when using this style of distributed build system for testing, e.g., Shi et al. [8] and Vakilian et al. [6] proposed to refactor tests or code as to help the build system more precisely schedule tests across machines. Unlike these distributed build systems, we consider scheduling tests "offline", deciding which machines each test should go on ahead of time. Stratis et al. [37] proposed test scheduling in a heterogeneous system, where tests can be scheduled on machines with drastically different hardware, e.g., machines with CPUs and GPUs. While they consider machines of different configurations, they are also scheduling tests as soon as resources are available.

Genetic algorithms have been used in prior work for various software engineering tasks. For example, Le Goues et al. [32] proposed GenProg to automatically repair buggy code via a genetic algorithm, where an individual is a sequence of code edits, with each generation creating different sequences of code edits with the goal to pass all tests. Fraser and Arcuri [33] proposed EvoSuite, a technique for automatically generating Java unit tests via genetic algorithms, where an individual is a method sequence representing a unit test and the fitness function is the branch coverage achieved by the tests.

IX. Conclusions

We propose scheduling tests to run in parallel across heterogeneous machines for more efficient testing, unlike traditional approaches that schedule tests across machines that use the same configuration for all machines. We present GASearch, a genetic algorithm approach to schedule tests across heterogeneous machines to reduce test running time and price. We also consider the risks of resource-constrained flaky tests, whose pass/fail outcomes depend on the machine on which they are run. We find that GASearch can schedule tests better than baselines that schedule tests across homogeneous machines. By adjusting the weights in the fitness function, GASearch can also schedule tests to better balance the tradeoffs between running time and price.

Acknowledgment

This work was partially supported by NSF grants: CCF-2145774 and CCF-2338287. We also thank Sarfraz Khurshid for his comments and Dragon Testing for their support.

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] 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.

[4] 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.

[5] 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.

[6] M. Vakilian, R. Sauciuc, J. D. Morgenthaler, and V. Mirrokni, "Automated decomposition of build targets," in *International Conference on Software Engineering*, 2014, pp. 123–133.

[7] H. Esfahani, J. Fietz, Q. Ke, A. Kolomiets, E. Lan, E. Mavrinac, W. Schulte, N. Sanches, and S. Kandula, "CloudBuild: Microsoft's distributed and caching build service," in *International Conference on Software Engineering Companion*, 2016, pp. 11–20.

[8] 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.

[9] T. Kim, R. Chandra, and N. Zeldovich, "Optimizing unit test execution in large software programs using dependency analysis," in *Asia-Pacific Workshop on Systems*, 2013, pp. 19:1–19:6.

[10] S. Misailovic, A. Milicevic, N. Petrovic, S. Khurshid, and D. Marinov, "Parallel test generation and execution with Korat," in *International Symposium on Foundations of Software Engineering*, 2006, pp. 135–144.

[11] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *International Conference on Software Engineering*, 2002, pp. 467–477.

[12] F. Steimann, M. Frenkel, and R. Abreu, "Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators," in *International Symposium on Software Testing and Analysis*, 2013, pp. 314–324.

[13] "About GitHub-hosted runners," https://docs.github.com/en/actions/using-github-hosted-runners/using-github-hosted-runners/about-github-hosted-runners, 2024.

[14] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *International Symposium on Foundations of Software Engineering*, 2014, pp. 643–653.

[15] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli, "Understanding flaky tests: The developer's perspective," in *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 830–840.

[16] "Flaky tests at Google and how we mitigate them," https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html, 2016.

[17] D. Silva, M. Gruber, S. Gokhale, E. Arteca, A. Turcotte, M. d'Amorim, W. Lam, S. Winter, and J. Bell, "The effects of computational resources on flaky tests," *arXiv preprint arXiv:2310.12132*, 2023.

[18] "The effects of computational resources on flaky tests (artifact)," https://zenodo.org/records/10015435, 2023.

[19] "GitHub Actions," https://github.com/features/actions, 2024.

[20] M. Beller, G. Gousios, and A. Zaidman, "Oops, my tests broke the build: An explorative analysis of Travis CI with GitHub," in *International Conference on Mining Software Repositories*, 2017, pp. 356–367.

[21] "GASearch: Test scheduling across heterogeneous machines while balancing runtime, price, and flakiness," https://sites.google.com/view/gasearchartifact, 2024.

[22] M. Mitchell, *An Introduction to Genetic Algorithms*. The MIT Press, 1998.

[23] E. S. Hou, N. Ansari, and H. Ren, "A genetic algorithm for multiprocessor scheduling," *IEEE Transactions on Parallel and Distributed systems*, vol. 5, no. 2, pp. 113–120, 1994.

[24] J. Carpenter, S. H. Funk, P. Holman, A. Srinivasan, J. H. Anderson, and S. K. Baruah, "A categorization of real-time multiprocessor scheduling problems and algorithms." in *Handbook of Scheduling - Algorithms, Models, and Performance Analysis*, J. Y. Leung, Ed. Chapman and Hall/CRC, 2004.

[25] C.-Y. Lee and J. D. Massey, "Multiprocessor scheduling: combining LPT and MULTIFIT," *Discrete applied mathematics*, vol. 20, no. 3, pp. 233–242, 1988.

[26] "Deap," https://github.com/DEAP/deap, 2024.

[27] "Runtime options with Memory, CPUs, and GPUs," https://docs.docker.com/config/containers/resource_constraints, 2024.

[28] "AWS Fargate," https://aws.amazon.com/fargate, 2024.

[29] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, "iDFlakies: A framework for detecting and partially classifying flaky tests," in *International Conference on Software Testing, Verification, and Validation*, 2019, pp. 312–322.

[30] W. Lam, A. Shi, R. Oei, S. Zhang, M. D. Ernst, and T. Xie, "Dependent-test-aware regression testing techniques," in *International Symposium on Software Testing and Analysis*, 2020, pp. 298–311.

[31] W. Lam, K. Muşlu, H. Sajnani, and S. Thummalapenta, "A study on the lifecycle of flaky tests," in *International Conference on Software Engineering*, 2020, pp. 1471–1482.

[32] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A generic method for automatic software repair," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, 2012.

[33] G. Fraser and A. Arcuri, "Evolutionary generation of whole test suites," in *International Conference on Quality Software*, 2011, pp. 31–40.

[34] S. Baluja and R. Caruana, "Removing the genetics from the standard genetic algorithm," in *International Conference on Machine Learning*, 1995, pp. 38–46.

[35] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2013.

[36] P. McMinn, "Search-based software test data generation: A survey," *Journal of Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[37] P. Stratis and G. Brown, "Assessing the effect of device-based test scheduling on heterogeneous test suite execution," in *International Conference on Evaluation and Assessment in Software Engineering*, 2018, pp. 193–198.