Hierarchy-Aware Regression Test Prioritization

Hao Wang
UC Berkeley
Berkeley, CA, USA
hwang628@berkeley.edu

Pu (Luke) Yi
Stanford University
Stanford, CA, USA
lukeyi@stanford.edu

Jeremias Parladorio
National University of Rio Cuarto
Rio Cuarto, Argentina
jeremiasparladorio@gmail.com

Wing Lam
George Mason University
Fairfax, VA, USA
winglam@gmu.edu

Darko Marinov
University of Illinois
Urbana-Champaign, IL, USA
marinov@illinois.edu

Tao Xie
Peking University
Beijing, China
taoxie@pku.edu.cn

***Abstract*—Regression testing is widely used to check whether software changes lead to test failures. Regression Test Prioritization (*RTP*) aims to order tests such that tests that are more likely to fail are run earlier. Prior RTP techniques—which we call *hierarchy-unaware (HU)*—ignored an important aspect: real test suites are organized hierarchically, and individual tests belong to composites that can be hierarchically nested. Prior RTP work overlooked the runtime cost to switch across hierarchical test compositesand used the APFD_c metric, which represents the runtime of tests till test failures, to rank orders generated by RTP techniques. However, APFD_c can misleadingly rank orders if their runtimes differ (e.g., two orders may have different numbers of composite switches and, consequently, runtimes). To account for runtime differences, we propose a new metric, HAPFD_c. Unlike APFD_c, HAPFD_c enables proper comparison of test orders with different runtimes by "extending" runtimes as needed. To reduce the cost of composite switching, we introduce *hierarchy-aware (HA)* RTP by presenting *meta-techniques* that first prioritize composites and then tests within composites. We evaluate HA RTP on test classes in multi-module Java and Maven projects from two large datasets used in prior work. The results show that our HA RTP improves both HAPFD_c values and time-based metrics over HU RTP.**

***Index Terms*—regression test prioritization, test interleaving**

I. Introduction

Regression testing is an important activity to check whether software changes lead to test failures. Researchers have developed many techniques to improve regression testing, and several surveys [1–5] present overviews of the proposed techniques. Regression Test Prioritization (**RTP**) [6, 7] aims to order, i.e., prioritize, tests in a test suite to find test failures sooner rather than later. The motivation is to provide faster feedback to developers, so they can debug test failures [8] as soon as possible. Conceptually, RTP techniques use information from one or more historical runs of the test suite, or from recent changes, to prioritize the test suite for the current changes. Various techniques use different kinds of information, e.g., code coverage [6, 9], historical failures [10], timing information [8], and black-box information [11], along with different kinds of technologies, e.g., machine learning [5, 12, 13], information retrieval [14, 15], and peer sharing [16].

Since the two seminal papers [6, 7], RTP has been studied with increasingly realistic experiments, substantially improving four main aspects of the studies. Specifically, to evaluate RTP techniques, earlier work used (1) automatically generated tests instead of manually written tests [17], (2) simulated software evolution instead of real evolution [18], (3) mutants or manually seeded faults instead of real test failures from continuous-integration systems [13, 14, 19–22], and (4) metrics based on the number of test runs till test failures, such as Average Percentage of Faults Detected (**APFD**) [9], instead of metrics based on the *runtime* of tests till test failures, such as cost-cognizant APFD ($\mathbf{APFD_c}$) [23].

However, all RTP techniques from prior work have ignored the fact that software projects organize tests hierarchically, akin to the composite design pattern [24], and switching test execution across composites incurs runtime cost. We call prior work *Hierarchy-Unaware (**HU**)* RTP. We define a *test composite* as a set of tests that share the same running configuration. For example, most Java projects use a testing framework, such as JUnit [25] or TestNG [26], and a build system, such as Maven [27] or Gradle [28]: individual JUnit/TestNG test methods belong to test classes, which themselves belong to Maven/Gradle modules that provide the running configuration[1]. Thus, we view the test suite for a multi-module Maven project as *several test composites*, one for each module. Running test classes in a prioritized order incurs additional cost when consecutive tests belong to different modules.

While newer RTP work evaluates techniques using metrics (e.g., APFD_c [23]) that better represent real time, their experiments [13, 14, 16, 19, 20, 30] ignored the runtime cost of Switching Across Tests Composites (***SATC***), which includes not only the runtime to launch a new virtual machine but also the runtime to load classes, set up and tear down tests, etc. In fact, the SATC cost is closely related to the cost of running each test in a new JVM: studies [31–35] have shown that such cost can be orders of magnitude higher than running multiple tests in one JVM. The problem of SATC costs has been considered in pairwise testing [36], but not in prior RTP work. By ignoring SATC costs, prior RTP work can *incorrectly rank RTP techniques* [13, 14]. Only one prior study [14] used

[1]Our evaluation uses Java projects with JUnit and Maven, but the hierarchical organization is widespread in other programming languages, testing frameworks, and build systems, e.g., pytest [29] for Python has test functions that belong to test classes that belong to test files that belong to test directories.

a non-zero cost (but a low, constant value of 5.8 milliseconds) for switching between any two test classes, whether from the same or different modules. However, we find that the SATC cost can substantially affect the *test-suite runtime*—the total runtime between starting to run tests and receiving the test results—when the same test suite is run in orders with different SATC costs (Section VI-B).

To properly evaluate RTP techniques, we derived a new metric, *Hierarchy-aware* $APFD_c$ (***HAPFD*$_\mathbf{c}$**), which enables the comparison of test orders with different test-suite runtimes. Traditionally, researchers generated orders with different RTP techniques and compared the orders' APFD_c values to rank RTP techniques. If two (or more) orders have the same test-suite runtime, then comparing them by APFD_c gives the same ranking as comparing them by the average time to faults [37]. However, different orders of the same test suite often have different runtimes. HAPFD_c improves on APFD_c to enable comparison of test orders with different test-suite runtimes, intuitively by "extending" faster test orders to match the slowest test order from a given set of orders (Section II).

To reduce the high SATC cost, we introduce *Hierarchy-Aware (**HA**)* RTP, which first prioritizes test composites (e.g., Maven modules) and then tests within composites (e.g., JUnit test classes). To prioritize composites, we present a set of *meta-techniques*. HA RTP applies these meta-techniques to prioritize composites and applies prior RTP techniques to prioritize tests within each composite. We use the term "meta-techniques" as they can be combined with many prior RTP techniques. The specific meta-techniques that we present utilize the test prioritization scores already provided in the two datasets used in our evaluation [14, 19]. Unlike HU RTP, which can interleave tests across composites, HA RTP does not interleave tests across composites and produces orders with the least possible number of composite switches (Figure 1).

We compare our meta-techniques on two large datasets [14, 19] that include test failures observed on Travis CI for dozens of open-source projects over multiple years. The datasets are used in recent papers [13, 14, 19, 20, 38]. From these two datasets, we obtain 2,357 *jobs*, where each job runs the test suites from a multi-module Maven project and contains real test failures. Using these jobs, we compare our meta-techniques and find that the meta-technique that greedily puts the test with the lowest RTP score last (i.e., choose the "worst" test and all other tests in its module to run last) performs the best. Moreover, to compare HA and HU RTP when the SATC cost is actually measured, we obtain 43 (newer) jobs from 28 projects and run their tests. For these jobs, we compare the orders from HA RTP with the orders from HU RTP [13, 14, 19]. The results show that using HA RTP to reduce the SATC cost leads to different rankings of RTP techniques (e.g., the best RTP technique changes) and higher HAPFD_c values, due to the number of module switches: on average, 33.6 for HU RTP vs. just 5.8 for HA RTP.

Our paper makes not only technical contributions to RTP but also a methodological contribution for research on this topic, by *running the generated test orders to properly account for their runtimes* and not ignoring substantial SATC runtime costs. Recommendations for improving research methodology have a long tradition [39]. Some examples on related topics include automated repair [40], defect prediction [41], mutation testing [42], test-suite reduction [43], and statistical analysis [44]. In fact, even RTP has been advancing over time, e.g., moving from artificial software evolution or faults to real evolution [18] and real failures [13, 14, 19].

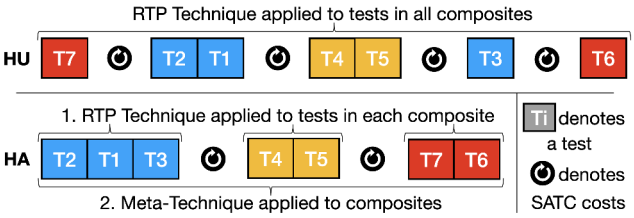

Fig. 1. HU vs. HA RTP example with three composites. Tests with the same color belong to the same composite.

Overall, this paper makes the following main contributions:
Test Hierarchy: We point out an important aspect of hierarchical test organization, ignored by prior RTP research.
Meta-Techniques: We propose meta-techniques to make prior RTP techniques hierarchy-aware, reducing their SATC cost.
Evaluation: We empirically show that hierarchy-aware orders are better than hierarchy-unaware orders in multiple aspects.

Our implementation of the HA RTP and the scripts that we use for our empirical evaluation are publicly available [45].

II. Metric Selection

Prior RTP work often used cost-cognizant Average Percentage of Faults Detected (**APFD$_\mathbf{c}$**) [4, 23, 46–49] and recently used Average Time To Fault (**ATTF**) [37] to rank test orders. A higher APFD_c value should indicate that a test order is better, i.e., finds all faults in the test suite *earlier* on average. APFD_c is calculated as the area under the curve of the percentage of faults detected against the percentage of test-suite runtime. ATTF is calculated as the average time to detect all the faults in a test order. Section II-A has more formal definitions.

One desirable property of APFD_c that ATTF lacks is *normalization of values*. Specifically, APFD_c normalizes the value to the unit range (0.000,1.000), making it easier to compare RTP techniques across different projects and test suites. On the other hand, one desirable property of ATTF that APFD_c lacks is the *ranking of test orders based on real time*, which is what developers care about. APFD_c may not rank two (or more) orders of one test suite the same as ATTF if the orders do not have the same runtime. In fact, orders can have substantially different runtimes, e.g., in our experiments, HA and HU orders differ by up to 438%.

Figure 2 shows a scenario where APFD_c misleadingly ranks test orders. Consider two test orders, O_1 and O_2, of a test suite with four tests: $C_1.T_1$, $C_1.T_2$ in composite C_1, and $C_2.T_1$, $C_2.T_2$ in composite C_2. In both orders, $C_1.T_1$ and $C_2.T_1$ fail and find different faults. The figure shows the individual test runtimes. Each SATC costs 30 units of time. We

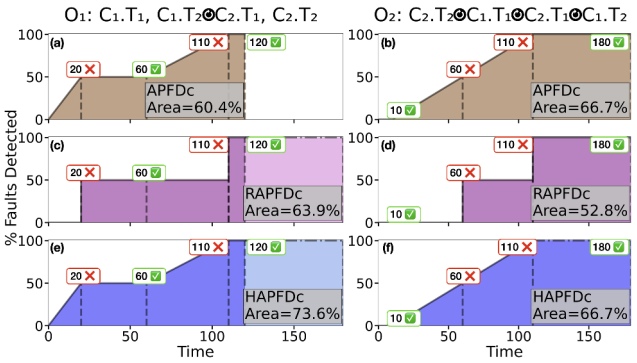

Fig. 2. APFD_c, RAPFD_c, and HAPFD_c for two orders O_1 and O_2; tests $C_1.T_1, C_1.T_2, C_2.T_1, C_2.T_2$ have runtimes 20, 40, 20, 10, respectively; SATC cost is 30, indicated by circle with arrow; $C_1.T_1$ and $C_2.T_1$ fail. APFD_c misleadingly ranks O_2 as better with a higher APFD_c value even though O_2 finds faults slower than O_1. Unlike APFD_c, RAPFD_c and HAPFD_c extend O_1 to reach $R = 180$ units of time to properly rank O_1 as better. Unlike RAPFD_c, APFD_c and HAPFD_c assume that the % of faults detected grows linearly for failing tests.

add each SATC cost to the runtime of the *next* test, because it starts later. The test-suite runtimes for O_1 and O_2 are 120 and 180 units of time, respectively. Therefore, the testing resource constraint R [50] is set to 180. As seen from the figure, O_1 is better than O_2 as it finds the first fault faster (at time 20 vs. 60) and the second fault in the same amount of time (110). However, according to APFD_c values, it would *appear* that O_2 is better as it has a higher APFD_c value because O_2 finds the second fault at 61% (110/180) of its test-suite runtime, while O_1 finds it at 92% (110/120) of its test-suite runtime.

Inspired by Wang et al. [50], we derive HAPFD_c based on APFD_c as a way to correct this anomaly. According to Wang et al. [50], their RAPFD_c metric can rank the techniques fairly given a testing resource constraint even if the corresponding test order runtimes differ. They do so by "extending" faster-running test orders as if their test-suite runtimes were the same as the slowest-running test order, or by "trimming" slower-running orders so that they can actually fit in the resource limit. HAPFD_c follows this intuition of RAPFD_c and extends the test order to a resource limit by appending a dummy passing test to the order. Figure 2 shows an example comparing HAPFD_c, APFD_c, and RAPFD_c. To compute HAPFD_c for O_1, we first extend its runtime to be the same as O_2, and compute the APFD_c on the extended order (Sub-figure (e)). Sub-figures (e) and (f) show that HAPFD_c finds O_1 is indeed better than O_2 in finding faults earlier. HAPFD_c differs from RAPFD_c (Sub-figures (c) and (d)) in that HAPFD_c follows APFD_c and assumes that the percentage of the faults detected grows linearly with the execution of the failing test. In summary, HAPFD_c preserves the positive aspects of APFD_c and RAPFD_c to ensure a fair comparison when considering test orders with different runtimes.

A. HAPFD_c *Formula*

The majority of RTP papers [13, 14, 16, 19, 20, 30] (1) used runtime to measure test costs and (2) considered the fault severity as constant across all faults. With these two common assumptions, the APFD_c formula [23] becomes

$$APFD_c = \frac{\sum_{i=1}^{m}(\sum_{j=TF_i}^{n} t_j - \frac{1}{2}t_{TF_i})}{m \times \sum_{j=1}^{n} t_j} \tag{1}$$

where m is the number of faults, n is the number of tests, and TF_i is the first place where the i^{th} fault is detected.

This APFD_c formula does not make it explicit that the test cost can differ across test orders. Let T be a test suite with n tests. Let O be an order (permutation) of all tests from T. To make the test cost explicit, we define $c_O(.)$ (or $c(.)$ when O is clear from the context) as the cost of a test t ($c_O(t)$) or a test suite T ($c_O(T)$) under the order O. With $c(.)$, we rewrite $APFD_c$ (for an order O) as

$$\begin{aligned}\text{APFD}_c &= \frac{\sum_{i=1}^{m}\left(\sum_{j=TF_i}^{n} c(t_j) - \frac{1}{2}c(t_{TF_i})\right)}{m \times c(T)} \\ &= 1 - \frac{\sum_{i=1}^{m}(\sum_{j=1}^{TF_i} c(t_j) - \frac{1}{2}c(t_{TF_i}))}{m \times c(T)}\end{aligned} \tag{2}$$

To allow for proper ranking of different test orders even when their runtime differs, we set the constraint from APFD_c to be the longest runtime among all the test orders under consideration, i.e., $R = \max c_O(T)$.

$$\text{HAPFD}_\text{c} = 1 - \frac{\sum_{i=1}^{m}\left(\sum_{j=1}^{TF_i} c(t_j) - \frac{1}{2}c(t_{TF_i})\right)}{m \times R} \tag{3}$$

We can then rewrite HAPFD_c to RAPFD_c [50], which assumes that the faults are detected only at the end of a failing test execution as

$$\text{RAPFD}_\text{c} = 1 - \frac{\sum_{i=1}^{m}\sum_{j=1}^{TF_i} c(t_j)}{m \times R} \tag{4}$$

To provide some more intuition about HAPFD_c and APFD_c metrics, we relate them to the ATTF metric, which averages the time for all the failures (assuming that each failure identifies a unique fault [37]):

$$ATTF = \frac{\sum_{i=1}^{m}\left(\sum_{j=1}^{TF_i} c(t_j) - \frac{1}{2}c(t_{TF_i})\right)}{m} \tag{5}$$

Substituting ATTF into the APFD_c and HAPFD_c formulas:

$$\text{APFD}_c(O) = 1 - \frac{ATTF(O)}{c_O(T)} \tag{6}$$

$$\text{HAPFD}_\text{c}(O) = 1 - \frac{ATTF(O)}{R} \tag{7}$$

The traditional APFD_c depends on the runtime and the ATTF of the order O. HAPFD_c retains the normalization property of APFD_c, while depending on only ATTF and R, which is irrelevant to the runtime of the current order. We prove that $HAPFD_c$ ranks test orders the same as $ATTF$ for all possible sets of orders and show how RAPFD_c and HAPFD_c compare to each other on our website [45].

Algorithm 1: HU & HA RTP of hierarchical test suites

```
1  // allTestsIn and allCompositesIn are helpers that
2  //    find all tests and composites, respectively, in a test suite
3  // testsIn finds direct test children of a composite
4  Function HU_RTP(suite):
5  |   return sorted(allTestsIn(suite), RTPtechnique)
6  Function HA_RTP(suite):
7  |   // sort the composites according to the meta-technique
8  |   composites = sorted(allCompositesIn(suite), score)
9  |   sortedTests = [ ] // empty list
10 |   for composite in composites do
11 |   |   // sort the tests in each composite individually
12 |   |   tests = testsIn(composite)
13 |   |   sortedTests.append(sorted(tests, RTPtechnique))
14 |   return sortedTests
15 Function score(composite):
16 |   tests = testsIn(composite)
17 |   return metaTechnique(map(RTPtechnique, tests))
```

III. HA and HU RTP Algorithm and Meta-Techniques

We next precisely describe Hierarchy-Unaware (**HU**) and Hierarchy-Aware (**HA**) RTP of hierarchical test suites that consist of tests and composites. A *test* is an individual atomic test, and is what the composite design pattern [24] calls a "leaf". A *composite* has a list of "children" that can each be either a test or (recursively) another composite. A *(test) suite* is the top-level composite. HU RTP techniques treat the suite as a list of tests, ignoring the composites to which the tests belong; as a result, they can produce test-suite orders that *interleave tests* from different composites. In contrast, HA RTP considers the hierarchical structure of the test suite and produces test-suite orders that do *not interleave tests* from different composites.

The principle of HA RTP is general and can apply to various existing RTP techniques, but because our evaluation datasets [14, 19] used RTP techniques that assign a numeric prioritization *score* to each test, we instantiate HU and HA RTP utilizing the prioritization scores, shown in (Python-like pseudo-code) Algorithm 1. Following the terminology of the composite design pattern [24], a *component* is either a test or a composite. `HU_RTP` first recursively finds all the tests in the test suite and then orders the tests based on their prioritization scores assigned by the specific RTP technique used (line 5). `HA_RTP` first recursively finds all the composites and then sorts them by their scores (line 8), which are determined by (1) the scores of their direct test children (not including tests that belong to their child components) and (2) the meta-technique used to aggregate individual tests' scores (Section III-A). HA effectively "flattens" the hierarchical structure among composites (but *not* among composites and tests), as common in modern test runners, e.g., Maven [27] or Gradle [28]. In each composite, HA RTP orders the tests by their scores, and it appends the tests from different composites in a global order, following the established composite order and test orders within composites (lines 10-13). All sorting is done in the *descending* order of scores.

In theory, developers could run tests for each composite in parallel when testing a project with a hierarchy. The discussion of HA or HU RTP is less meaningful in this context, because each composite can be run individually. However, studies [51, 52] found that parallelization can lead to concurrency issues that undermine the accuracy of the testing result. Thus, this work considers only running composites sequentially.

A. Meta-Techniques

We propose four meta-techniques that compute a composite's score by aggregating the scores of its test children: Highest Total First (**HTF**) uses `sum(scores)`, Highest Average First (**HAF**) uses `average(scores)`, Highest Score First (**HSF**) uses `max(scores)`, and Lowest Score Last (**LSL**) uses `min(scores)`. We call them "meta-techniques" because they can be combined with existing RTP techniques to adapt them from HU to HA.

The intuition is as follows. HTF is similar to the traditional "total" techniques [9] that first run conceptually the largest test. HAF is similar to the traditional "cost-cognizant" techniques [23] that consider the potential value of the test relative to its cost – in our case, a composite relative to the number of tests; we do not explicitly include cost in HAF because many RTP techniques already include cost. HSF greedily prioritizes the test with the highest score first in the test-suite order, while obeying the constraint of HA. LSL is the dual of HSF, greedily putting the test with the lowest score last.

IV. Specializing HA RTP for Maven

The general principles of HA RTP apply in all cases that involve hierarchical organization of test suites with test composites. In this section, we focus in more detail on multi-module Maven projects, because Maven is the most popular build system for Java projects [53] and our two evaluation datasets include such projects. In fact, multi-module Maven projects are prevalent. Of the top-starred 100 Java, Maven-based projects on GitHub, 73 had multiple modules [45] with an average of 19 modules per project.

We next describe how our general HA terminology corresponds to Maven projects. A *test* is a (JUnit) *test class*, and a *composite* is a *Maven module*. Specializing Algorithm 1 for multi-module Maven projects, the RTP techniques assign a score to a test class, and our meta-techniques assign a score to a Maven module, based on the scores of test classes in the module. By default, running `mvn test` at the top level visits the modules one by one to run each test suite. For each module, Maven creates a new JVM, runs the tests, and shuts down the JVM at the end. The creation and shutdown of JVMs incur runtime costs. The *SATC cost* between modules includes runtime cost to launch a new JVM[2], load classes, potentially load other files from disk into memory, perform just-in-time compilation, set up and tear down a test, etc. The overall switch runtime can greatly vary due to module switches in test orders with and without interleavings. Test orders with interleavings can have much longer runtime due

[2]While the cost to launch a JVM is small (< 100 millisecond on a modern JVM), the other costs can be substantial, hence in most Maven modules, all tests run in one JVM rather than each test in an isolated JVM [31–35, 54].

TABLE I
EFFECTIVE TECHNIQUES FROM PRIOR WORK [13, 14].

Technique	Criteria from Peng et al. [14]
CCHIR	execution time, historical failures, and IR score
CCH	execution time and historical failures
HIR	historical failures and IR score
CCIR	execution time and IR score
QTF	execution time
OptIR	information-retrieval (IR) score
HIS	historical failures
Technique	**Criteria from Elsner et al. [13]**
MFFr	historical (test, file)-failures and failures
MFF	historical (test, file)-failures
AD	average duration (execution time)
LT	most recent pass-to-fail transition
HIS	historical failures

to their larger numbers of module switches compared to those without interleavings (Section VI-B).

To run test-suite orders that interleave composites, developers can either (1) invoke a test runner multiple times and run consecutive test classes from the same module together, referred to as *Multi-JVM mode*; or (2) invoke a test runner once and run all test classes together, referred to as *One-JVM mode*. The latter can be much faster than the former, due to the SATC cost. However, attempting to run all tests in one JVM can introduce many false positives and false negatives (Section VI-E), because different modules can have different classpaths and working directories. Our experiments focus primarily on the former, motivated by the fact that machine cost is much cheaper than that of manual test-failure inspections [55]. We describe an approach to invoke a test runner multiple times that is most favorable, in terms of runtime, for HU orders that interleave classes from different modules (Section V-B2).

V. EVALUATION SETUP

We aim to answer the following research questions (RQs):
RQ1: How do our meta-techniques compare?
RQ2: How do HA and HU RTP compare?
RQ3: Can APFD_c mislead comparisons?
RQ4: Can ignoring SATC costs mislead comparisons?
RQ5: How do Multi-JVM and One-JVM modes compare?

Our overall goal is to evaluate which meta-technique produces the best HA RTP and understand whether assumptions and metrics from prior work may provide misleading results.

A. Study Subjects

As we are the first to study HA RTP, we discuss next how we build a dataset for our study.

1) RQ1: To answer RQ1, we use two public datasets, one provided by Peng et al. [14], which we call IRBRTP because it was originally used to evaluate information-retrieval (IR) based RTP techniques, and RTPTorrent provided by Mattis et al. [13, 19]. We selected these two datasets because they are recent and among the largest datasets for RTP.

Modules and Jobs Selection. Both datasets are constructed from GitHub projects that use Java and Maven, filtering for projects that used Travis CI [56], to obtain a dataset with real test failures instead of artificial ones. Each build on Travis CI can have multiple jobs, which typically run the same code version but with different commands or same commands in different environments. Each job has its own individual overall result (pass, fail, or error) and may run test suites for a project with one or more Maven modules. Each test class in a test suite has its own result (pass, fail, or error).

Because our meta-techniques (described in Section III-A) order the modules, we select all jobs from the two datasets that have test classes in more than one module. IRBRTP contains all the necessary data (e.g., test coverage, runtime, fault history) to reproduce the numbers reported in the original paper [14]; we could reproduce the results from Peng et al. [14] up to the last reported digit. To understand the SATC costs, we need to know the module that each test class belongs to. Interestingly enough, IRBRTP already includes the module name for each test class—the authors computed this information but *ignored* it during prioritization. IRBRTP has in total 2,980 jobs in 123 projects, with 1,368 jobs in 71 projects having test classes in more than one module. Upon inspection of these jobs, we find that 404 jobs in the dataset have a wrong mapping from the test class names to modules, because one test class name can appear in multiple modules, and IRBRTP did not carefully resolve such cases. We make the list of projects and jobs with incorrect mappings publicly available [45] and share it with the authors. Finally, we obtain 964 jobs from 66 projects from IRBRTP.

RTPTorrent also contains a wealth of data However, RTPTorrent does not include the Maven module names for tests. In addition, the raw logs for some of the jobs are not available. We develop an approach using Z3 [57] to compute likely module names and publicly release an extension [45] of RTPTorrent to facilitate future RTP research, in particular for developing and evaluating future HA RTP. We omit the details of the approach due to space limit. The extension contains the module assignment for 1,393 jobs from RTPTorrent.

An important point is that the 964 jobs from IRBRTP and 1,393 jobs from RTPTorrent do *not* overlap. In fact, these jobs come from a non-overlapping set of projects (66 from IRBRTP and 13 from RTPTorrent) except for one shared project that has no common job in the two datasets (because they focus on different time periods, with IRBRTP having generally newer jobs than RTPTorrent).

RTP Techniques. Both datasets were used to compare several RTP techniques with the information from the *CI order*—the default order run by the job, whose results are shown in the Travis logs. Based on the CI order included in the datasets, with the runtime and the expected execution result for each test, one can compute APFD_c for the orders generated by various techniques. For IRBRTP, the authors identified seven test prioritization techniques as the most effective (Table I). The original RTPTorrent paper [19] aimed mainly to release a dataset and had a relatively small comparison. Elsner et al. [13] presented a much bigger evaluation using RTPTorrent. They identified five RTP techniques as the most effective, one of which overlaps with a technique from Peng et al. [14] (Table I).

TABLE II
STATISTICS FOR EACH JOB FOR RQ2-RQ5. "TSR" IS THE TEST-SUITE RUNTIME (IN SEC) AVERAGED ACROSS ALL HU ORDERS.

ID	Project	# Test Class Fails / Total	# JVMs HA	# JVMs HU	TSR HU
J1	abel533/Mapper	10 / 41	2	9.3	8.3
J2	apache/incubator-dubbo	1 / 142	14	77.6	175.0
J3	apache/incubator-dubbo	1 / 142	14	80.1	176.5
J4	apache/incubator-dubbo...	1 / 13	2	4.9	9.5
J5	apache/incubator-dubbo...	1 / 13	2	4.9	9.5
J6	apache/incubator-dubbo...	1 / 13	2	4.9	9.5
J7	aws/aws-sdk-java	1 / 190	3	49.7	210.0
J8	aws/aws-sdk-java	1 / 190	3	47.3	210.8
J9	aws/aws-sdk-java	1 / 190	3	48.4	211.2
J10	demoiselle/framework	2 / 8	2	2.4	4.0
J11	elasticjob/elastic-job-lite	1 / 89	3	37.7	54.1
J12	gchq/Gaffer	1 / 75	3	18.7	55.5
J13	google/auto	1 / 31	4	14.0	21.5
J14	google/auto	1 / 31	4	16.3	23.5
J15	hs-web/hsweb-framework	1 / 28	11	21.9	38.7
J16	jtablesaw/tablesaw	2 / 47	2	5.1	5.8
J17	LiveRamp/hank	1 / 63	3	27.3	175.6
J18	lukas-krecan/JsonUnit	1 / 17	5	13.4	8.0
J19	lukas-krecan/JsonUnit	1 / 17	5	13.4	8.1
J20	lukas-krecan/JsonUnit	1 / 17	5	13.4	8.0
J21	lukas-krecan/JsonUnit	1 / 17	5	13.7	8.1
J22	magefree/mage	1 / 816	6	26.4	189.8
J23	mitreid-connect/OpenID...	1 / 38	3	20.9	10.4
J24	ModeShape/modeshape	6 / 189	5	72.3	253.2
J25	networknt/light-4j	2 / 16	4	10.7	4.4
J26	ocpsoft/rewrite	74 / 151	17	107.3	76.7
J27	ocpsoft/rewrite	63 / 140	17	93.4	63.0
J28	ocpsoft/rewrite	73 / 150	17	98.3	70.0
J29	ocpsoft/rewrite	73 / 150	17	99.1	68.5
J30	ocpsoft/rewrite	33 / 67	9	29.3	15.0
J31	onelogin/java-saml	2 / 15	2	4.0	7.6
J32	onelogin/java-saml	2 / 15	2	4.0	7.7
J33	orbit/orbit	2 / 60	5	13.7	105.7
J34	pippo-java/pippo	1 / 11	2	2.7	1.9
J35	prometheus/client_java	2 / 36	16	26.0	30.1
J36	protegeproject/protege	4 / 44	2	4.4	4.6
J37	rapidoid/rapidoid	1 / 82	9	50.3	190.7
J38	RIPE-NCC/whois	41 / 320	7	179.6	149.9
J39	sismics/reader	2 / 25	3	7.1	96.6
J40	spring-projects/spring...	4 / 117	2	31.0	31.8
J41	spring-projects/spring...	4 / 117	2	28.4	31.4
J42	st-js/st-js	4 / 38	3	7.9	4.5
J43	teamed/qulice	1 / 5	3	3.9	13.5
Sum × 2 / Arith. Mean × 3		**428 / 3976**	**5.8**	**33.6**	**66.5**

We evaluate as many techniques as possible on both datasets using only the information provided by the authors. MFFr, MFF, AD, and LT from Elsner et al. [13] require information from passing builds, which IRBRTP does not have. On the other hand, CCHIR, HIR, CCIR, and OptIR from Peng et al. [14] require test IR information, which RTPTorrent does not have. In total, we evaluate all seven technique from Peng et al. [14] on IRBRTP, and five techniques from Elsner et al. [13] and two techniques from Peng et al. [14] on RTPTorrent.

2) RQ2-RQ5: As RQ2-RQ5 all involve the actual test-suite runtimes, we use the same dataset for them (but a different one than RQ1). Unlike RQ1 where the number of module switches for each order is exactly the same for all meta-techniques, the number of module switches between HA RTP and HU RTP can differ for RQ2-RQ4 (average of 5.8 switches for HA compared to 33.6 switches for HU). Similarly, for RQ5, the two modes to run tests have a different number of switches (i.e., One-JVM mode has zero switches while Multi-JVM mode can have many). These differences in the number of module switches can influence the overall test suite runtimes.

TABLE III
SELECTION PROCEDURE TO GET 43 JOBS FROM IRBRTP.

Filter Applied	# Jobs	# Projects
All projects	2,980	123
Multi-module projects	1,368	71
Recent 5 SHAs	$\leq$ 111	71
Compile successfully	64	45
Corrected orders in IRBRTP	58	39
Reproducible test failures	44	28
No duplicate	43	28

To obtain actual test suite runtimes, we look for jobs that can be run. We cannot run all the jobs because of dependency issues as some of them are several years old. We use the IRBRTP dataset instead of RTPTorrent as the former is more recent and more likely to compile and run. Of the 123 projects in IRBRTP, we start with selecting 71 that have more than one module with test classes. We try compiling each of these project using the five most recent GitHub commit SHAs to obtain the most recent SHA in which the project compiles. We find that 45 projects successfully compile in one of the five most recent SHAs. We do not attempt to compile more than the five most recent SHAs because projects that do not compile on the most recent versions are less likely to compile on even older versions (e.g., 39 of our 45 projects compiled on the most recent SHA, while the other 6 projects compiled between the second and fifth most recent SHA). We then filter out 6 projects that have a wrong mapping from the test class names to modules in the IRBRTP dataset.

For each of the remaining 39 projects, we run 36 test orders on the project and SHA of each job: 28 orders generated by combining 4 meta-techniques (Section III-A) and 7 RTP techniques from Table I, 1 CI order from the original dataset, and 7 orders generated directly by the 7 (HU) RTP techniques. We call the orders directly generated by the 7 RTP techniques *HU orders*, and the others *HA orders*. We run each test order five times to filter out tests that exhibit flaky test outcomes [58] and to filter out jobs that have no failure in every order. We obtain 44 jobs (from 28 projects) after such filtering. We then inspect and filter out 1 job that has the same 36 orders. In total, we obtain a dataset of 43 jobs from 28 projects. Table II shows statistics for these jobs, and Table III summarizes how we obtained these jobs.

We run all the timing experiments on various isolated virtual machines with the same configuration: 4CPUs, 8GB RAM, 2.5 GHz/3.2 GHz clock speed, and Intel Xeon processor. The experiment is conducted under Java 1.8.0_311 and Maven 3.8. To reduce runtime noise from affecting our results, we run every order five times. We do not run more because we observe that the runtime of each order is relatively stable in five runs—the average Coefficient of Variance [59] of the runtimes among all orders from all 43 jobs is 0.04 seconds.

Table II shows the real times to run the tests for these jobs. These times are obtained with our scripts that aim to *minimize*

the time for HU techniques. Running without our scripts, e.g., using the existing `mvn` commands, would make these times longer. One can question (1) whether relatively short times (up to a few minutes) make it relevant to prioritize these test suites, and (2) whether HA provides benefit over HU only for short-running test suites. For (1), we note that these same datasets were used in multiple recent studies on RTP [13, 14, 19, 20]; the key contributions are novel algorithms/techniques that are expected to scale well (or even better) on longer-running test suites. We also note that our experiments run each test suite for dozens of various configurations (e.g., the choices of techniques and meta-techniques, plus repeating experiments five times), so the overall machine time for experiments is vastly greater than the runtime for one test-suite run. For (2), in Section VI-B, we study and find that the benefit that HA provides over HU does *not* go down with the test-suite length; if anything, the relationship is slightly positive for these jobs (but the correlation is not statistically significant).

B. Methodology

1) RQ1: For the jobs from the two datasets (Section V-A1), we use the selected techniques to reproduce the HU experiments [13, 14, 19] and add our HA experiments. We follow all experimental settings from Peng et al. [14] for both datasets: breaking ties based on the CI test order rather than randomly, prioritizing newly added before existing tests, adding a small overhead for test runtimes (especially for tests whose runtime was seemingly 0), using specific settings for information-retrieval techniques, etc.

To evaluate RQ1, we rely on APFD_c [23] as it is the most popular RTP metric that takes the test-suite runtimes into account. We do not evaluate our proposed metric, $\mathrm{HAPFD}_\mathrm{c}$, because we are comparing only meta-techniques, which have no interleavings. If the number of interleavings is similar, then the test-suite runtimes of different orders are similar, and consequently, each order's $\mathrm{HAPFD}_\mathrm{c}$ is similar to APFD_c.

2) RQ2-RQ5: To evaluate RQ2-RQ5, we use $\mathrm{HAPFD}_\mathrm{c}$ as it takes into account that different orders of the same test suite can have different runtimes. For RQ3, we compare APFD_c [23] to $\mathrm{HAPFD}_\mathrm{c}$. For RQ4, we evaluate how HA RTP compares to HU RTP when ignoring SATC costs.

For RQ2-RQ4, we run tests in only *Multi-JVM mode*, which invokes a test runner multiple times (i.e., runs test classes from the same module together before running test classes in another module). In RQ5, we evaluate how one may run all test classes in one JVM, referred to as *One-JVM mode*, and show why that is likely less desirable than Multi-JVM mode.

Multi-JVM Mode. A natural way to run the orders in Multi-JVM mode is to use the existing build systems such as Maven and Gradle, but these build systems do not currently support running orders that interleave tests from composites. For example, to run such an order in Maven, one has to invoke `mvn test` multiple times. Since `mvn test` does more tasks than just running the tests (e.g., checking coding style, checking re-compilation), launching multiple `mvn test` adds unnecessary runtime costs to HU orders. To fairly compare different orders (with or without interleavings), we do not use build systems to run the orders and instead run the orders in a more favorable way in terms of the SATC cost.

Our scripts to run Multi-JVM mode minimize the number of JVMs (and thus the runtime costs from JVM and test startups and teardowns) that need to be run for a given order, by running all consecutive tests from the same module in one JVM before switching to another JVM. For example, given the order $\langle T_1, T_2, T_3, T_4 \rangle$, where tests T_1, T_2, and T_4 belong to one module, and T_3 belongs to another module, our scripts run three JVMs: $\langle T_1, T_2 \rangle$, $\langle T_3 \rangle$, and $\langle T_4 \rangle$. To run the tests in each module, we run `java JUnitCore T_1 T_2...` with as many consecutive test classes as possible in the module directory. Each `java` command sets up a JVM, runs all tests, and tears down the JVM.

One-JVM Mode. One-JVM mode follows a similar process as Multi-JVM but runs all the tests in one JVM with the *concatenation* of the classpaths of all modules at the project base directory.

Timing. To collect timing information for each test class and the overhead between switching classes, we use a simple customized JUnit Wrapper that lets JUnit core run each test class one by one, and prints the test class info before running it. Our wrapper outputs the test class start time in the `testRunStarted` method, and collects its end time in the `testRunFinished` method.

3) All RQs: We use one-to-one failure-to-fault mapping for all the experiments, following the default from prior work [14]. We also aggregate the metric values across all projects when comparing RTP techniques. We obtain a metric value for each order generated by RTP techniques on each job. Following prior work [14], we (1) first compute metric value for each project as the arithmetic mean of the values for each job, thus obtaining a distribution of values for a technique; and (2) use two statistic tests to compare the distributions for the techniques. One statistic we use is the arithmetic mean of the values across all projects. (Computing the mean first for each project and then across projects provides an "unweighted" average [14], making the results more representative.)

Another statistical approach we use to compare the techniques is the Tukey's Honest Significant Difference test [60]. We aggregate the score of all the techniques (or meta+technique pairs), including a default run of the CI order, into a single batch, and apply the Tukey's test on them. The test compares multiple distributions of metric values to identify which differences are statistically significant. Specifically, the test assigns to each distribution one or more letters to indicate how it overlaps with the others. For example, if four techniques T_1, T_2, T_3, and T_4 obtain letters "A", "A", "AB", and "B", respectively, then T_1 and T_2 are significantly better than T_4, while T_3 partly overlaps with the other three and does not statistically significantly differ.

Tables IV and V show the results for each combination of meta-technique and RTP technique (HA RTP). Each cell shows the unweighted average metric value of the technique, and we highlight in yellow color the cells of the technique(s) from the

TABLE IV
COMPARING META-TECHNIQUES ON TWO DATASETS AND A VARIETY OF BEST TECHNIQUES FROM PRIOR WORK [13, 14].

	HTF	HAF	HSF	LSL
964 jobs from IRBRTP, **CI=.225**				
CCHIR	.612	.653	.635	.711
CCH	.564	.598	.566	.682
HIR	.594	.720	.665	.729
CCIR	.581	.616	.606	.683
QTF	.665	.585	.517	.637
OptIR	.512	.650	.601	.669
HIS	.497	.547	.537	.384
Average	**.575**	**.624**	**.590**	**.642**
1,393 jobs from RTPTorrent, **CI=.230**				
CCH	.583	.586	.582	.777
MFF	.730	.743	.731	.766
MFFr	.719	.730	.740	.747
HIS	.656	.681	.669	.737
LT	.670	.671	.604	.673
QTF	.647	.471	.456	.580
AD	.663	.476	.477	.580
Average	**.666**	**.622**	**.608**	**.694**

top group in Tukey's test results (i.e., technique(s) that obtain the sole "A" letter). The comparison of techniques should be primarily based on the statistical results of the Tukey's test, but we present the unweighted average for a simple comparison.

VI. EVALUATION RESULTS

A. RQ1: Comparison of Meta-Techniques

Table IV shows the APFD_c values for each pair of meta-technique and technique, in short *meta+technique pair*, averaged over all projects for each dataset. The average values are rather similar across all meta-techniques, and all are higher than the CI value, clearly showing the benefit of using any meta-technique. However, the Tukey's test for the APFD_c values shows that for each dataset only one meta+technique pair, LSL+HIR for IRBRTP and LSL+CCH for RTPTorrent, is in the top group, which we highlight in yellow color in each section of Table IV. The averages of the APFD_c values also show LSL as the best meta-technique, with the highest value for five out of seven RTP techniques for each dataset. The fact that LSL is the best for two different datasets, using different techniques (although three of the seven techniques overlap), increases our confidence that LSL is the best meta-technique.

LSL is the most counter-intuitive meta-technique, effectively prioritizing "worst" tests at the end rather than "best" tests at the start. Our finding that LSL is the best meta-technique is conceptually similar to the finding by Koru et al. [61], who report that to find faults faster in manual inspection, one should inspect source files by increasing size with larger files at the end because they are the "worst", having fewer faults relative to their size.

A1: The meta-techniques are similar, but LSL is often the best.

B. RQ2: Hierarchy-Aware vs. Hierarchy-Unaware RTP

We next compare our HA meta-techniques and the traditional HU RTP on the 43 jobs from IRBRTP where we could measure the actual test runtimes (including module switch cost). The first section of Table V shows the HAPFD_c values averaged over all projects. The Tukey's test for these HAPFD_c values shows that two meta+technique pairs (LSL+CCIR, HAF+CCHIR), both HA, are in the top group. Comparing the performance of HA and HU RTP, namely the first four columns and the last column in Table V, we find that for every technique, HA meta-techniques almost always achieve a higher HAPFD_c value than HU. Additionally, the Tukey's test indicates a statistically significant gap between the performance of HA and HU RTP. Due to space constraints, we show how Table V's results change with the RAPFD_c metric on our website [45]; LSL is still the best.

In practice, testers care about not only HAPFD_c (to find failures faster) but also test-suite runtime (especially if there is no failure). Figure 3 plots both HAPFD_c values and test-suite runtimes (geometric mean of test-suite runtime normalized

TABLE V
COMPARING HA AND HU RTP FOR 43 JOBS FROM IRBRTP USING VARIOUS METRICS AND SATC COSTS; 1ST SECTION SHOWS A *proper* COMPARISON; 2ND AND 3RD SECTIONS USE *misleading* COMPARISONS.

	HTF	HAF	HSF	LSL	HU
HAPFD_c, actual time & SATC cost, **CI=.586**					
CCHIR	.751	.801	.761	.790	.671
CCH	.720	.760	.732	.771	.649
HIR	.749	.771	.781	.756	.750
CCIR	.750	.797	.764	.812	.626
QTF	.769	.748	.748	.793	.613
OptIR	.724	.780	.745	.733	.697
HIS	.715	.729	.700	.694	.654
Average	**.740**	**.770**	**.747**	**.764**	**.666**
APFD_c, actual time & SATC cost, **CI=.311**					
CCHIR	.568	.615	.588	.633	.627
CCH	.521	.559	.536	.617	.598
HIR	.572	.630	.640	.583	.709
CCIR	.564	.605	.568	.659	.583
QTF	.586	.546	.533	.641	.560
OptIR	.529	.631	.551	.562	.657
HIS	.532	.550	.513	.496	.493
Average	**.553**	**.591**	**.561**	**.599**	**.604**
(H)APFD_c, CI time & const SATC cost, **CI=.274**					
CCHIR	.687	.720	.686	.799	.852
CCH	.631	.681	.612	.782	.835
HIR	.617	.759	.730	.719	.799
CCIR	.609	.658	.652	.800	.793
QTF	.734	.647	.575	.768	.755
OptIR	.516	.720	.609	.697	.772
HIS	.524	.570	.551	.400	.564
Average	**.617**	**.679**	**.631**	**.709**	**.767**

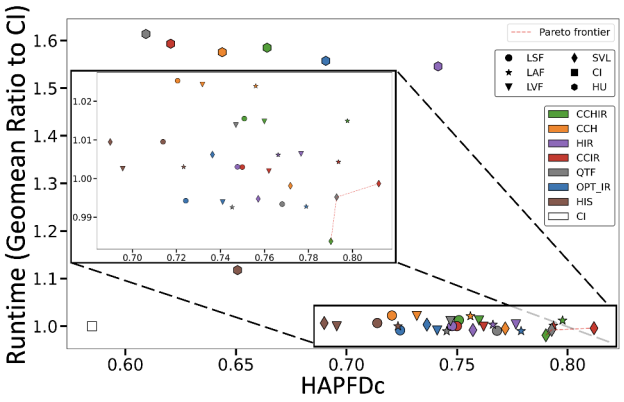

Fig. 3. Scatter plot of the test suite runtimes (ratio to CI) and HAPFD_c scores under different meta+technique pairs and the Pareto frontier.

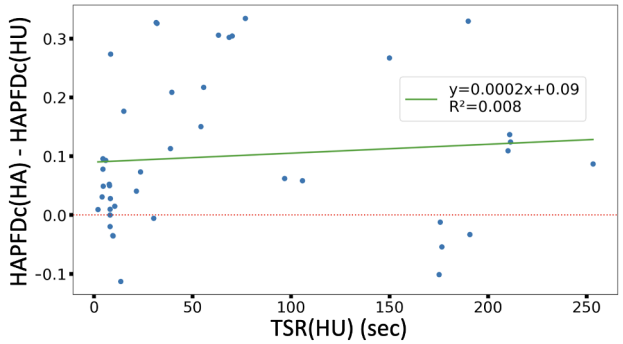

Fig. 4. Scatter plot showing the benefit of HA over HU against test-suite runtime (TSR) of HU techniques. The non-negative slope shows a non-diminishing benefit as TSR increases for bigger jobs.

to the run of the CI order) together, where each mark is a meta+technique pair, with the shape encoding the meta-technique and the color encoding the technique. We also show the Pareto frontier, which provides a set of optimal meta+technique pairs for both $\mathrm{HAPFD_c}$ value and test-suite runtime. The frontier contains LSL+CCIR, LSL+QTF, and LSL+CCHIR, which supports the earlier statistical test result that LSL is the best.

One may question whether the benefit of HA over HU generalizes for projects with larger test-suite runtimes. Figure 4 plots the benefit against the test-suite runtime. Each point corresponds to a job. The x coordinate is the average test-suite runtimes of HU techniques, and the y coordinate is the difference of the average $\mathrm{HAPFD_c}$ for HA and HU techniques (higher is better for HA) for the job. The red dotted horizontal line shows the $y = 0$ boundary where HA and HU perform equally on average. The green line shows the best line fit for the plot: it has a positive slope (though not high, 0.0002) suggesting that the benefit of HA over HU does not diminish with the increase of test-suite runtime. To evaluate the correlation between the benefit of HA over HU and the test-suite runtime, we calculate the Spearman coefficient [62], resulting in 0.27 for all techniques. The positivity further confirms that the benefit does not diminish.

A2: HA RTP outperforms HU RTP in terms of $\mathrm{HAPFD_c}$ and test-suite runtimes.

C. RQ3: Traditional Metric is Misleading

If we use APFD_c as the evaluation metric, we reach a different, *misleading* conclusion about HA vs. HU RTP. The second section of Table V shows the results. We apply the Tukey's statistical test only on values within each section not across the sections. According to the Tukey's test results for APFD_c values, many more meta+techniques, and some HU techniques, join top group, which does not match the results derived from $\mathrm{HAPFD_c}$ values. LSL does not stand out among meta-techniques, and moreover the difference between HA and HU shrinks, with HU appearing even slightly better.

While $\mathrm{HAPFD_c}$ and APFD_c values should not be directly compared, we can still contrast their values because both are normalized to .000–1.000. For HA meta-techniques, the APFD_c values are much lower than the corresponding $\mathrm{HAPFD_c}$ values. The reason is the test-suite runtimes, which are much higher ($\sim$1.5X on average) for HU orders than for HA orders. Effectively, when comparing HA orders with HU orders, R from $\mathrm{HAPFD_c}$ formula (3) is much larger than $c(T)$ from APFD_c formula (2). As a result, for HU, the APFD_c values are only slightly lower than the corresponding $\mathrm{HAPFD_c}$ values. Recall that APFD_c does *not* properly account for the test-suite runtime and, instead, uses for each order its own test-suite runtime as the normalizing factor in the denominator of the formula. Overall, the inconsistency of results when using APFD_c and $\mathrm{HAPFD_c}$ metrics indicates that using APFD_c as the key metric can distort the comparison of RTP orders and techniques.

A3: We find that APFD_c and $\mathrm{HAPFD_c}$ do *not* get the same results and that APFD_c can misleadingly rank test orders and RTP techniques, hence $\mathrm{HAPFD_c}$ is needed.

D. RQ4: Importance of SATC Cost

If we assume that the SATC cost is constant, we also reach a different, even *more misleading* conclusion about HA vs. HU RTP. The approach common in prior work [13, 14, 19] that evaluated different techniques based on the data from Travis logs [56] assumes the runtime of each test is the same as it was in the Travis log. Additionally, Peng et al. [14] add the overhead for switching between any two test classes to be a constant (5.8ms), even for classes from different modules, because the overhead information is missing in Travis logs. They use the APFD_c metric, but in their model (i.e., assuming test runtimes to be the same across orders, with constant SATC cost of 0 or almost 0), the test-suite runtime is the same across all orders, and thus $\mathrm{HAPFD_c}$ and APFD_c values are exactly the same. We use the notation (H)APFD_c to refer to this fact.

The third section of Table V shows the results for this model. The Tukey's test has HU CCHIR and HU CCH as the two best pairs. This result does not match the pairs that are in the top group when evaluating properly with $\mathrm{HAPFD_c}$ and actual time and SATC cost: in the first section of Table V, HU is never in the top group. In fact, in the first section, HU RTP is worse than *every* HA meta-technique for *every* one of the seven techniques. In contrast, in the third section, HU RTP (specifically CCHIR, CCH, and HIR) *appears* to be better than *every* HA meta+technique when we ignore the SATC cost.

While this model incorrectly compares HA and HU RTP, it still ranked HA meta+techniques *among themselves* (not against HU) about the same as the proper model (e.g., the HA meta+technique with the highest (H)APFD_c value is still from LSL). The reason is that HA meta-techniques generate prioritized orders that take about the same test-suite runtime because they have the same minimal number of module switches (switching into and out of each module exactly once), which validates our comparison in Table IV.

A4: Prior work often assumes zero or constant SATC cost, which misleadingly finds that HU RTP outperforms HA RTP.

E. RQ5: One-JVM vs. Multi-JVM Mode

When we run the One-JVM mode for 43 jobs and compare the results with the Multi-JVM mode, we find that One-JVM can lead to a non-trivial percentage of false positives (tests that fail in One-JVM but pass in Multi-JVM) and false negatives (tests that pass in One-JVM but fail in Multi-JVM). To avoid non-determinism (or flaky tests [58]) from affecting One-JVM, we run each order in One-JVM five times and take the intersection of false positives from the five runs and the union of false negatives from the five runs. The average False Discovery Rate (FDR, the number of false positives divided by the number of failures, ignoring the jobs that show no failure in One-JVM) of One-JVM for each meta+technique pair, across all 43 jobs is above .35. The average False Negative Rate (the number of false negatives divided by the number of actual failures) is .105 for all pairs.

We find that 30 jobs (69.8% of 43 jobs) have at least one false positive or false negative. We inspect a sample of false positives from the One-JVM mode and find several reasons.

Classpaths: Tests from different modules may require conflicting classpaths. This issue often manifests in exceptions, such as `ClassDefNotFound` or reflection not finding some fields or methods [63].

Directories: Tests may expect to be run in a specific directory, e.g., to find some resource files for the module. For example, in job J42, the test `AnnotationsTest` reads a Java source file from the path `src/...` and generates some Javascript code for it. The path for this file from the project's top-level directory is `generator/src/...`. When run in the One-JVM mode, the test fails because it cannot find the file.

Build configurations: Tests from different modules may use different test runners, require different setups/teardowns, etc. that are handled by Maven. When tests from different modules are run together, it is difficult to provide correct configurations.

Program states: In One-JVM mode, the tests may create different program states than in Multi-JVM mode. The tests may then pass in some program states but fail in others. Such tests are often called order-dependent (OD) flaky tests [32]. Lam et al. [54] proposed RTP techniques aware of OD tests.

We also inspect and find the causes of some false negatives in the One-JVM mode. For example, in job J36, the `setUp` of the `GOProfile_TestCase` test reads a file with the path `protege-desktop/src/...`. This test passes only when it is run from the top-level directory. In fact, the test actually fails in `mvn test` and the Multi-JVM mode, because they run the test in the module directory, `protege-desktop`, and thus the test cannot find the file. The failure is masked in One-JVM because it runs the test in the project's top-level directory.

Test-Suite Runtime Differences. To understand the differences of the two modes, we focus on the 13 jobs that have no false positive or false negative. We do not compare test-suite runtimes for runs with different test failures, because tests may run much slower or much faster when they fail. We find that One-JVM reduces the time by ∼15% over Multi-JVM HA runs, and substantially (∼43%) over the Multi-JVM HU runs.

Overall, One-JVM does not appear to be a practical alternative. The false positive rates are high; ∼34.5% of test failures are not real, while developers usually tolerate under 10% of false positives, e.g., in static analysis tools [64]. Additionally, the speedup of ∼15% does not appear motivating enough when developers may prefer test reliability; Candido et al. [51] report a similar finding for test parallelization.

A5: One-JVM negatively affects test reliability and is not used in practice, despite the speedups that it could provide.

VII. Threats to Validity

We evaluate on a limited number of jobs and projects, so the results we derive may not generalize to other projects. To mitigate this threat, we select two datasets from prior work [13, 14, 19], which are recent and among the largest RTP datasets.

We run each test order five times to mitigate the effect of noise. We also remove any jobs and flaky-test classes that have different test outcomes in different runs to prevent them from affecting the results. Although we run tests in isolated virtual machines with identical configurations, there can be fluctuations in the test runtimes due to physical machine differences and other workloads on the machines. To mitigate such concerns, we measure the runtime variance between different runs of the same job to verify that our result is stable. The projects that we study are relatively small. To mitigate scalability concerns, we check that HA benefits do not diminish with the increase of test-suite runtime.

Another threat to validity is that some jobs in the datasets from prior work did not run all project modules. Maven by default stops execution for the first module for which some test fails. The modules that would have run after the failing module may have all their tests pass or some tests fail. Thus, our results could differ if we have also used those modules.

VIII. Conclusions

We have pointed out an important but ignored aspect of hierarchical test organization and its impact on test-suite runtime. More importantly, our results show that *proper evaluations of RTP should account for test orders with different SATC costs and thus different test-suite runtimes*. We propose a new metric, $\mathrm{HAPFD_c}$, that allows properly comparing test-suite orders with different test-suite runtimes. We propose four meta-techniques that adapt existing hierarchy-unaware (HU) RTP techniques to become hierarchy-aware (HA), and our evaluation shows that Lowest Score Last (LSL) is often the best. Moreover, our evaluation shows that HA orders are better than HU orders in many aspects. We hope that our positive results will motivate more work on HA RTP (e.g., techniques that tolerate SATC cost for prioritization by allowing the interleaving of tests across different composites).

Acknowledgments

We thank Toni Mattis for discussing RTPTorrent. This work was partially supported by NSF grants CCF-1763788, CCF-1956374, and CCF-2338287. We acknowledge research support from Dragon Testing, Microsoft, and Qualcomm.

References

[1] E. Engström and P. Runeson, "A qualitative survey of regression testing practices," in *PROFES*, 2010.

[2] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Software Testing, Verification & Reliability*, 2012.

[3] R. H. Rosero, O. S. Gómez, and G. Rodríguez, "15 years of software regression testing techniques – A survey," *International Journal of Software Engineering and Knowledge Engineering*, 2016.

[4] Y. Lou, J. Chen, L. Zhang, and D. Hao, "Chapter one - A survey on regression test-case prioritization," in *Advances in Computers*, 2019.

[5] R. Pan, M. Bagherzadeh, T. A. Ghaleb, and L. Briand, "Test case selection and prioritization using machine learning: A systematic literature review," *Empirical Software Engineering*, 2022.

[6] G. Rothermel, R. Untch, C. Chu, and M. Harrold, "Test case prioritization: An empirical study," in *ICSM*, 1999.

[7] W. Wong, J. Horgan, S. London, and H. Agrawal, "A study of effective regression testing in practice," in *ISSRE*, 1997.

[8] D. Saff and M. D. Ernst, "Reducing wasted development time via continuous testing," in *ISSRE*, 2003.

[9] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *TSE*, 2001.

[10] J.-M. Kim and A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments," in *ICSE*, 2002.

[11] C. Henard, M. Papadakis, M. Harman, Y. Jia, and Y. Le Traon, "Comparing white-box and black-box test prioritization," in *ICSE*, 2016.

[12] A. Bertolino, A. Guerriero, B. Miranda, R. Pietrantuono, and S. Russo, "Learning-to-rank vs ranking-to-learn: Strategies for regression testing in continuous integration," in *ICSE*, 2020.

[13] D. Elsner, F. Hauer, A. Pretschner, and S. Reimer, "Empirically evaluating readily available information for regression test optimization in continuous integration," in *ISSTA*, 2021.

[14] Q. Peng, A. Shi, and L. Zhang, "Empirically revisiting and enhancing IR-based test-case prioritization," in *ISSTA*, 2020.

[15] R. K. Saha, L. Zhang, S. Khurshid, and D. E. Perry, "An information retrieval approach for regression test prioritization based on program changes," in *ICSE*, 2015.

[16] R. Cheng, L. Zhang, D. Marinov, and T. Xu, "Test-case prioritization for configuration testing," in *ISSTA*, 2021.

[17] H. Do, G. Rothermel, and A. Kinneer, "Empirical studies of test case prioritization in a JUnit testing environment," in *ISSRE*, 2004.

[18] 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 *ICSE*, 2016.

[19] T. Mattis, P. Rein, F. Dürsch, and R. Hirschfeld, "RTPTorrent: An open-source dataset for evaluating regression test prioritization," in *MSR*, 2020.

[20] A. S. Yaraghi, M. Bagherzadeh, N. Kahani, and L. C. Briand, "Scalable and accurate test case prioritization in continuous integration contexts," *Transactions on Software Engineering*, 2023.

[21] S. Elbaum, G. Rothermel, and J. Penix, "Techniques for improving regression testing in continuous integration development environments," in *FSE*, 2014.

[22] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige, "Reinforcement learning for automatic test case prioritization and selection in continuous integration," in *ISSTA*, 2017.

[23] S. Elbaum, A. Malishevsky, and G. Rothermel, "Incorporating varying test costs and fault severities into test case prioritization," in *ICSE*, 2001.

[24] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1994.

[25] "JUnit," 2024. [Online]. Available: https://junit.org

[26] "TestNG," 2024. [Online]. Available: https://testng.org/doc/documentation-main.html

[27] "Maven," 2024. [Online]. Available: https://maven.apache.org

[28] "Gradle," 2024. [Online]. Available: https://gradle.org

[29] "pytest," 2024. [Online]. Available: https://docs.pytest.org

[30] S. Wang, J. Nam, and L. Tan, "QTEP: Quality-aware test case prioritization," in *ESEC/FSE*, 2017.

[31] K. Muşlu, B. Soran, and J. Wuttke, "Finding bugs by isolating unit tests," in *ESEC/FSE*, 2011.

[32] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin, "Empirically revisiting the test independence assumption," in *ISSTA*, 2014.

[33] J. Bell and G. Kaiser, "Unit test virtualization with VMVM," in *ICSE*, 2014.

[34] J. Bell, "Detecting, isolating, and enforcing dependencies among and within test cases," in *FSE DS*, 2014.

[35] P. Nie, A. Celik, M. Coley, A. Milicevic, J. Bell, and M. Gligoric, "Debugging the performance of Maven's test isolation: Experience report," in *ISSTA*, 2020.

[36] S. Kimoto, T. Tsuchiya, and T. Kikuno, "Pairwise testing in the presence of configuration change cost," in *International Conference on Secure System Integration and Reliability Improvement*, 2008.

[37] L. Chen, F. Hassan, X. Wang, and L. Zhang, "Taming behavioral backward incompatibilities via cross-project testing and analysis," in *ICSE*, 2021.

[38] F. Dürsch, P. Rein, T. Mattis, and R. Hirschfeld, "Learning from failure: A history-based, lightweight test prioritization technique connecting software changes to test failures," Universität Potsdam, Tech. Rep., 2022.

[39] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*, 2012.

[40] R. Just, C. Parnin, I. Drosos, and M. D. Ernst, "Comparing developer-provided to user-provided tests for fault localization and automated program repair," in *ISSTA*, 2018.
[41] M. Tan, L. Tan, S. Dara, and C. Mayeux, "Online defect prediction for imbalanced data," in *ICSE*, 2015.
[42] T. T. Chekam, M. Papadakis, Y. Le Traon, and M. Harman, "An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption," in *ICSE*, 2017.
[43] A. Shi, A. Gyori, S. Mahmood, P. Zhao, and D. Marinov, "Evaluating test-suite reduction in real software evolution," in *ISSTA*, 2018.
[44] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *ICSE*, 2011.
[45] "Hierarchy-aware regression test prioritization," 2024. [Online]. Available: https://sites.google.com/view/testprioritization
[46] Y. Singh, A. Kaur, B. Suri, and S. Singhal, "Systematic literature review on regression test prioritization techniques," *Informatica*, 2012.
[47] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Software Testing, Verification and Reliability*, 2012.
[48] M. Khatibsyarbini, M. A. Isa, D. N. Jawawi, and R. Tumeng, "Test case prioritization approaches in regression testing: A systematic literature review," *Information and Software Technology*, 2018.
[49] O. Dahiya and K. Solanki, "A systematic literature study of regression test case prioritization approaches," *International Journal of Engineering & Technology*, 2018.
[50] Z. Wang and L. Chen, "Improved metrics for non-classic test prioritization problems," in *SEKE*, 2015.
[51] J. Candido, L. Melo, and M. d'Amorim, "Test suite parallelization in open-source projects: A study on its usage and impact," in *ASE*, 2017.
[52] H. Yuan, J. Lin, W. Lam, and A. Shi, "Test scheduling across heterogeneous machines while balancing running time, price, and flakiness," in *ICSME*, 2024.
[53] "Java programming," 2024. [Online]. Available: https://www.jetbrains.com/lp/devecosystem-2023/java
[54] W. Lam, A. Shi, R. Oei, S. Zhang, M. D. Ernst, and T. Xie, "Dependent-test-aware regression testing techniques," in *ISSTA*, 2020.
[55] K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy, "The art of testing less without sacrificing quality," in *ICSE*, 2015.
[56] "Travis CI - Test and deploy with confidence," 2024. [Online]. Available: https://travis-ci.com
[57] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *TACAS*, 2008.
[58] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *FSE*, 2014.
[59] B. S. Everitt and A. Skrondal, *The Cambridge dictionary of statistics*, 2010.
[60] J. W. Tukey, "Comparing individual means in the analysis of variance," *Biometrics*, 1949.
[61] A. G. Koru, K. El Emam, D. Zhang, H. Liu, and D. Mathew, "Theory of relative defect proneness," *Empirical Software Engineering*, 2008.
[62] C. Spearman, "The proof and measurement of association between two things," 1961.
[63] Y. Wang, R. Wu, C. Wang, M. Wen, Y. Liu, S.-C. Cheung, H. Yu, C. Xu, and Z. Zhu, "Will dependency conflicts affect my program's semantics?" *Transactions on Software Engineering*, 2021.
[64] C. Sadowski, J. Van Gogh, C. Jaspan, E. Söderberg, and C. Winter, "Tricorder: Building a program analysis ecosystem," in *ICSE*, 2015.