O!SNAP: Cost-Efficient Testing in the Cloud

Alessio Gambi
Saarland University, Germany
Email: gambi@st.cs.uni-saarland.de

Alessandra Gorla
IMDEA Software Institute, Spain
Email: alessandra.gorla@imdea.org

Andreas Zeller
Saarland University, Germany
Email: zeller@cs.uni-saarland.de

***Abstract*—Porting a testing environment to a cloud infrastructure is not straightforward. This paper presents O!SNAP, an approach to generate test plans to cost-efficiently execute tests in the cloud. O!SNAP automatically maximizes reuse of existing virtual machines, and interleaves the creation of updated test images with the execution of tests to minimize overall test execution time and/or cost. In an evaluation involving 2,600+ packages and 24,900+ test jobs of the Debian continuous integration environment, O!SNAP reduces test setup time by up to 88% and test execution time by up to 43.3% without additional costs.**

I. INTRODUCTION AND MOTIVATION

Test early, test often: Modern software development processes mandate frequent automated testing in order to get timely feedback on the quality of a system. But what if the automated tests run too long? As an example, consider the Debian Linux distribution, where the current continuous integration (CI) solution adopted by Debian presents test results back to developers after their code changes. Developers working on the `pdns` package for example have to wait for *over 13 hours* on average for the test results.[1]

The more time testing takes, the longer it takes developers to recognize and fix problems, and the longer it takes to deploy well-tested patches to users [1]. Hence, techniques to speed up test execution are very much needed in practice.

One way to speed up tests execution is to allocate several machines for parallel execution, for instance using a cloud infrastructure [2]. Testing in the cloud is a challenging problem though: While one can, in principle, allocate N machines to run N tests all in parallel, and thus run all tests in as little time as possible, this would induce huge costs due to setting up test environments on all N machines [1] and paying for all cloud resources [3]. On the contrary, running all tests on just one machine would bring down costs to a minimum, but would not make test execution more efficient in terms of time. The problem thus is to execute tests in a way that *balances out both cost and time.*

In this paper, we present O!SNAP, a technique to *automatically generate plans to cost-efficiently execute tests in the cloud.* O!SNAP takes as input the list of tests to execute, the list of available virtual machine images, and additional configuration parameters, such as the cost model adopted by the cloud provider. It produces as output a *test execution plan* suggesting which virtual machines to use, how to schedule test executions, and when to create more suitable virtual machines.

[1]Data extracted from the Debian continuous integration system, `debci`.

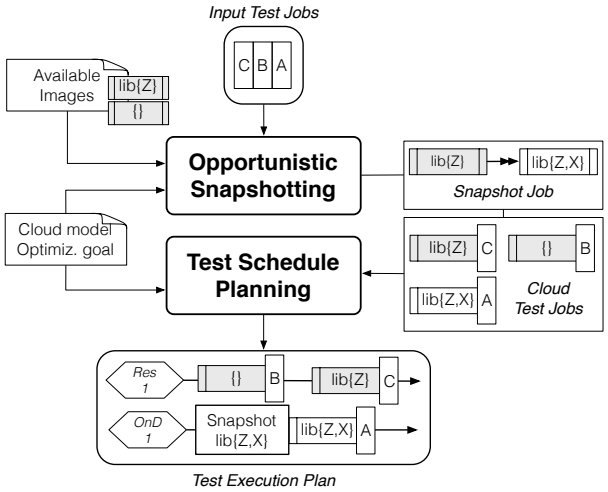

Fig. 1. O!SNAP optimization pipeline to plan cost-efficient tests executions in the cloud: 1) Opportunistic snapshotting identifies which new images are worth creating (*snapshot jobs*), and maps test jobs to images (*cloud test jobs*). 2) Test schedule planning computes the *test execution plan*.

O!SNAP works as a two-staged pipeline, as Figure 1 illustrates. It starts with *opportunistic snapshotting*, which aims to maximize the reuse of virtual machine images across test executions and build new images that limit the effort of setting up test environments. Next, *test schedule planning* computes the test execution plan aiming to minimize the overall test execution time and cost by interleaving the creation of new images and the execution of tests.

Test plans generated by O!SNAP can lead to significant savings. Compared to the current solution implemented in the Debian development, for continuously testing 10 software packages over a period of several days O!SNAP allows to reduce test execution time by 56.7%. As O!SNAP allows users to state cost versus time preferences, it also determines that if one is willing to pay slightly higher costs (c.a. 3$ per day),[2] test execution time can be reduced to up to 65.9%. And compared to a massively parallelized execution, O!SNAP shows that execution costs can be reduced by more than 87% if one is willing to increase execution time by only 5%.

Note that O!SNAP is orthogonal to other testing optimization strategies such as reducing the size of the test suite (e.g., [4]), prioritizing potentially more informative tests (e.g., [5]), or reusing partial test results (e.g., [6]). Hence,

[2]We use the Amazon EC2 price for `t2.medium` general purpose instances of EU (Ireland) Region.

O!SNAP can be combined with other optimization strategies to obtain even greater savings.

The remainder of this paper is organized as follows. After discussing modern cloud infrastructures and their cost models (Section II), we make the following contributions:

1) A characterization of the problem of planning cost-efficient test executions in the context of cloud-based testing environments (Section III).
2) The definition and assessment of O!SNAP, a novel technique to plan for test executions tailored to cloud-based testing environments (Section IV).
3) An empirical study over 2,600+ packages and 24,900+ test jobs of the Debian distribution, showing that O!SNAP can significantly speed up test executions while limiting the costs of using a cloud infrastructure (Section V).

After discussing related work (Section VI), we conclude the paper and present direction for future research (Section VII).

II. INFRASTRUCTURE AS A SERVICE

Cloud computing is the standard solution to effectively leverage large pools of remote resources and services. Cloud offerings come in three forms: as Software-as-a-Service (*SaaS*), where users access (remote) desktop-like software applications entirely managed by the cloud; as Platform-as-a-Service (*PaaS*), where users deploy proprietary code on cloud managed run-time execution environments; and as Infrastructure-as-a-Service (*IaaS*), where users can access system level virtualized resources.

In this paper, we focus on IaaS, which lets users programmatically control running virtual machines, also called *instances*. In the next sections we describe how to use instances to cost-efficiently parallelize test executions.

Instances start from predefined disks called *images*, and might resort to different combinations of computational power, memory, and network resources which define their *type* and *size*. For example, Amazon EC2, one of the main cloud providers nowadays, offers instances of type `t2` in different sizes, such as `nano` with 1 CPU and 0.5GB of memory, and `medium` with 2 CPUs and 4GB of memory.

Users can suspend instances and resume them later, or they can terminate the instances for good. Termination of instances frees physical resources, but comes at the price of loosing all the data stored inside the instances. Additionally, users can *snapshot* instances, that is, users can store the content of instance disks as new images which can be used later to start new instances with the backed up state.

Despite offering different resources and cost models, cloud providers share similar pay-per-use business models. Customers pay according to various parameters, which usually relate to the amount, type and size of requested instances, and the running time of each instance. Cloud providers metering accounts for the running time of instances by atomic *billing time units* (BTU). Thus, if a BTU amounts to one hour, the cost of using an instance for one minute is equivalent to the cost of using the instance for 59 minutes. Costs also depend on how instances are accessed, either on-demand or under a reservation. On-demand instances are requested and terminated on-the-fly, and can be used without prior investment; as a consequence, they do not incur fixed costs, and users pay only for the amount of consumed BTU. On the contrary, reserved instances are paid upfront a fixed fee that covers a given time frame (e.g., one year). Customers can then use them as much as they want without paying extra costs within the reservation period, but cannot get rid of them.

III. TEST EXECUTIONS IN THE CLOUD

Running tests in a cloud-based infrastructure allows developers to execute tests in parallel across several instances, and thus allows to consequently improve the overall efficiency of the testing process. However, porting test executions to the cloud may also incur several problems. First of all, virtualization introduces some performance overhead during the test execution. Then, using freshly allocated instances might delay the actual execution of the tests since each instance must be booted and suitably configured before tests execution. For example, configuring an instance for testing the `libreoffice-1:5.1.3 rc1-1` package in Debian requires the installation of 897 packages. Additionally, the execution of tests in the cloud leads to costs that might be hard to predict precisely. Thus, in practice, leveraging large pools of instances to speed up test execution might result in excessive high costs, given that instances are billed using BTU, and the time to configure instances before test executions may be significant and cannot be parallelized.

Our goal is to *make test execution in the cloud cost-efficient*. In essence, this means to find the right balance between parallelizing test executions, allocating multiple instances, and reducing instance configuration costs by carefully selecting which images run the tests.

Planning for cost-efficient test executions in the cloud can be formalized as a multi-objective optimization problem: Given the *test jobs* to execute with their *dependencies*, the available *images*, and the available *reserved instances*, the goal is to find a *test execution plan* that specifies the deployment and order of execution of the test jobs, and the amount of additional on-demand instances to use.

Finding a cost-efficient test plan stands between two conflicting goals: Deploying test jobs on the available reserved instances, which limits the costs but also limits parallelization; and, scheduling test jobs to run on additional instances, which might reduce test execution time but yields higher costs.

IV. THE O!SNAP APPROACH

To plan cost-efficient test executions in the cloud we adopt a divide-and-conquer approach: We split the original optimization problem in two simpler problems and solve them in a pipeline, as Figure 1 depicts. First, *opportunistic snapshotting* identifies the most appropriate images for running test jobs such that test jobs share as many test dependencies as possible; this reduces the test setup time, but might require the creation of new images. Then, *test schedule planning* schedules test job

executions to minimize test execution time and cloud resources cost.

A. Opportunistic Snapshotting

Opportunistic snapshotting takes place before executing the test jobs, and aims to find suitable images for running them. This includes a task to identify test dependencies (e.g., system libraries) that different test jobs may share, and, potentially, a task to create new cloud images to host such dependencies. These images can thus be readily used to execute the test jobs, with the advantage that the setup time diminishes since the execution environment already provides some of —if not all— the test dependencies.

As an example, given two hypothetical software packages to test (A and B) and an empty image ($I_{\mathcal{E}}$), the parallel execution of A and B in the cloud would: (i) start two instances using $I_{\mathcal{E}}$; (ii) download and install in both instances all the test dependencies for A, resp. B; and, (iii) execute the tests. The availability of an existing image ($I_{A \cup B}$), hosting the test dependencies requested by both A and B, would save execution time. In fact, the use of $I_{A \cup B}$ instead of $I_{\mathcal{E}}$ would require no setup at all during test execution. However, some dependencies required by A and B may not be *co-installable* [11]. This would make it unfeasible to create $I_{A \cup B}$, and, in turns, would make the tests for packages A and B impossible to run on the same instance. Nevertheless, it would still be possible to create images that contain only the non-conflicting dependencies of A and B, and use those images to speed up test execution.

The standard practice to create customized images like $I_{A \cup B}$ is snapshotting. Snapshotting requires starting an instance from a *parent* image, installing the dependencies in that instance, and storing its disk as a new image. This process has a cost that depends on which snapshot technology cloud providers use, which images are available and can be used as parent, and the amount of dependencies to install, among other factors. For example, creating $I_{A \cup B}$ starting from $I_{\mathcal{E}}$ as parent requires more effort than creating $I_{A \cup B}$ from either I_A or I_B. Consequently, choosing which snapshots to use has an impact on the overall cost-efficiency of the test execution. The main goal of opportunistic snapshotting is to find suitable snapshots that, despite their cost, are worth creating.

Opportunistic snapshotting takes as inputs the test jobs ($\mathcal{T}$) with their required dependencies (D_t) and expected test execution time (p_t); the available images ($\mathcal{I}$) with their provided dependencies (D_i); and the data describing the cloud, such as the time to create snapshots ($\mathtt{T}_{SN}$). Given such inputs, opportunistic snapshotting computes a mapping associating test jobs to images which shall execute them, and it identifies new images to create, if necessary. To do so, opportunistic snapshotting creates a *network flow model* that represents the structural dependencies among the possible images that can be snapshotted, and the potential deployments of test jobs onto those images. Our intuition is that the flow distribution in the network can model the execution of test jobs under different configurations. Hence, opportunistic snapshotting can use this model to estimate the time for setting up instances and creating new images, and, by computing the flow distribution which yields the minimum test setup time, it can find suitable images to run test jobs and identify which images are worth creating.

More formally, the network flow model is a directed acyclic graph in which edges have unlimited capacity and non-negative cost, and vertices have integer demand. This graph contains one special node, called *source* (`S`), which generates all the flow; several *image nodes* ($\mathtt{I}_\mathtt{j}$) which represent available and potential images; and, several *test nodes* ($\mathtt{T}_\mathtt{i}$) which correspond to the input test jobs and absorb one unit of flow each. The graph also contains three types of edges: *snapshot* edges, which connect the source to image nodes and model the creation of new images; *installation* edges, which connect only image nodes and model the installation of dependencies into instances; and, *test deployment* edges, which connect image nodes to test nodes. Installation and test deployment edges connect nodes according to the *subset relation* of their dependencies: the dependencies declared in the starting node must be entirely contained in the ones declared in the target node. Additionally, installation edges have *proportional* semantics, i.e., they incur a cost that is proportional to the amount of flow traversing them, while snapshot edges have *take-or-leave* semantics, i.e., they incur a fixed cost only when flow traverses them.

Given a feasible flow distribution, opportunistic snapshotting recovers the mapping between test jobs and images, either existing or potential, by following each flow backwards, i.e., from test nodes to the source node. If an image does not exist, then opportunistic snapshotting finds the most convenient parent image to use for creating it by computing the shortest path passing only through already existing images from `S` to that image node.

As our evaluation shows, opportunistically creating snapshots reduces the test setup time independently of how test jobs are parallelized or scheduled. However, parallelization and scheduling, as well as the allocation of computing resources to test executions, strongly affect the cost-efficiency of testing in the cloud. To deal with these limitations, we complement opportunistic snapshotting with test schedule planning, which we describe in the next section.

B. Test Schedule Planning

Test schedule planning computes test execution plans ($\mathbb{S}$) that fulfill the constraints about test jobs execution and creation of new images defined by opportunistic snapshotting. We formulate the test schedule planning as an optimization problem by means of Integer Linear Programming (ILP), and because of this, we refer to our test schedules as the *ILP Scheduler*.

The input of test schedule planning includes a set of *jobs*, both cloud test jobs and snapshot jobs, to schedule ($\mathcal{J}$); and, a *cloud cost model*, which defines the resource usage costs and the billing time unit (`B`). Additionally, test schedule planning requires a *goal model* that specifies the *objective function* (O).

We model jobs using two integer variables that capture their starting (t) and ending time (T) as follows:

$$T_j = t_j + \mathtt{p}_j,\ \forall j \in \mathcal{J}$$

where parameter $\mathtt{p}_j$ represents jobs duration, and $t_j \geq 0$. We represent the overall test execution time (T_{exec}) and cost (C_{exec}) by means of two integer variables, and define the objective function O around them:

$$O = \alpha \cdot T_{\text{exec}} + \beta \cdot C_{\text{exec}},\ \alpha, \beta \in \mathbb{R}_0^+$$

where the weights α and β enable developers to express the relative importance of test execution time and cost; for example, the setting ($\alpha = 1, \beta = 0$) states that developers aim to "minimize the execution time no matter the costs."

We define T_{exec} *indirectly* as the largest value across jobs ending time, and C_{exec} *directly* as cumulative cost for running the test jobs on the instances:

$$T_{\text{exec}} \geq T_j,\ \forall j \in \mathcal{J}$$

$$C_{\text{exec}} = \sum_{j \in \mathcal{J}} (\mathtt{c}_{R} \sum_{m \in R} \left\lceil \frac{\mathtt{p}_j}{\mathtt{B}} \right\rceil r_{j,m} + \mathtt{c}_{OD} \sum_{m \in OD} \left\lceil \frac{\mathtt{p}_j}{\mathtt{B}} \right\rceil o_{j,m})$$

where $\left\lceil \frac{\mathtt{p}_j}{\mathtt{B}} \right\rceil$ is the 'billable' time for running the job j; $\mathtt{c}_{R}$ and $\mathtt{c}_{OD}$ are the unitary costs for reserved (*R*) and on-demand (*OD*) instances; and, the binary variables $r_{j,m}$, resp. $o_{j,m}$, model the deployment of jobs to reserved, resp. on-demand, instances. These variables hold true iff job j runs on instance m.

We subject $r_{j,m}$ and $o_{j,m}$ to the following constraints to allow only feasible deployments:

$$\sum_{j \in \mathcal{J}} r_{j,m} \leq |\mathcal{J}|, \forall m \in R \qquad \sum_{j \in \mathcal{J}} o_{j,m} \leq 1, \forall m \in OD$$

$$\sum_{m \in OD} o_{j,m} + \sum_{m \in R} r_{j,m} = 1,\ \forall j \in \mathcal{J}$$

These constraints state that reserved instances can run multiples jobs, but they cannot run more jobs than the available ones; on-demand instances can run at most one job; and, jobs can be deployed on one and only one instance.

We account for the relative order in which jobs are executed using a binary variable $y_{j,k}$ for each pair of jobs as follow:

$$y_{j,k} + y_{k,j} \leq 1,\ \forall j,k \in \mathcal{J} \qquad y_{i,j} = 1,\ \forall i,j \in \mathcal{J} \text{iff } \mathtt{y}_{i,j} = 1$$

where $y_{j,k}$ holds true iff job j executes before job k, and the binary parameter $\mathtt{y}_{j,k}$ captures the order of execution imposed by the creation of snapshots.

Finally, we express global constraints on the test execution plan to enforce that only one job at a time can run on each reserved instance, and that jobs can start only when previous jobs have completed:

$$y_{j,k} + y_{k,j} \geq r_{j,m} + r_{k,m} - 1,\ \ \forall j,k \in \mathcal{J}, j > k, \forall m \in R$$

$$t_k \geq t_j + \mathtt{p}_j (\sum_{m \in R} r_{j,m} + \sum_{m \in OD} o_{i,m}) - \mathcal{K}(1 - y_{j,k}), \forall j,k \in \mathcal{J}, j \neq k$$

where $\mathcal{K}$ is a *large* integer constant that implements the so-called Big-M method, a standard method to translate "if-then-else" logical constraints into integer linear constraints [12].

V. Evaluation

This evaluation aims to quantify the improvements in terms of cost-efficiency that O!Snap achieves, and addresses the following research questions:

RQ1 Do opportunistically created snapshots reduce the test setup time?

RQ2 Does O!Snap improve the cost-efficiency of testing in the cloud compared to state-of-the art solutions?

We conduct our experiments on a PC with 8-core 4.00GHz i7-6700K CPU and 64GB of RAM running Debian 8.4.

A. Test Subjects

As test subjects we consider 42,867 software packages of the Debian amd64 architecture ecosystem [13]. We selected the Debian ecosystem because Debian periodically releases information about past test executions, such as the test execution time and the list of test dependencies, which O!Snap requires as input.

For this evaluation, we consider the software packages which are currently managed by Debian's continuous integration system, `debci` [14]. In particular, we extracted the tests data of the last trimester of 2015, a period of normal operation which gives us an intuition of the potential benefits of O!Snap in typical working conditions. From the available data on all the 28,286 test jobs, we removed data about broken packages, packages with missing test execution information, and *outliers*. We consider as outliers those packages that declare either too few (less than thirty) or too many (more than a thousand) test dependencies. Our final dataset contains data about 24,901 test jobs, which test 2,619 packages and involve 12,444 unique dependencies.

We implemented custom scripts to measure the time to setup each and every package in the dataset, and we compute the costs of using cloud resources in USD ($) by utilizing the Amazon EC2 pricing schema [15]. In particular, we considered the price for using `t2.medium` instances in the EU Region (Ireland). According to Amazon, such instances are meant for general purpose workloads such as running software tests.

While conducting our study, we make the following simplifying assumptions about cloud instances: instances can run one test job at the time, and they are only of type `t2.medium`.

B. RQ1: Test Setup Time Reduction

We start our evaluation with RQ1: *Do opportunistically created snapshots reduce the test setup time?* Table I shows the average reduction of the test setup time (col. $\bar{I}_{\%}$) by applying the opportunistic snapshotting technique to randomly selected software packages in the Debian dataset (col. *Tests*). For the sake of completeness, the table also reports statistics about the dependencies among these packages, such as the average amount of *Unique* and *Shared* dependencies.

First, we randomly draw n packages and compute their test setup time, i.e., the overall time to download and install all the required dependencies. Next, we compute the opportunistic snapshots for the packages under test. Then, we compute the

TABLE I
REDUCTION IN THE AVERAGE TEST SETUP TIME

Tests	**Dependencies**		$\bar{\mathbf{I}}_{\%}$
n	Unique #	Shared #	%
5	597	75	78.98
10	1013	65	81.77
20	1452	24	85.65
50	2015	5	88.08

test setup time when these snapshots are in place. Finally, we compute $\bar{\mathbf{I}}_{\%}$ according to the following equation:

$$\bar{\mathbf{I}}_{\%} = (1/N) \sum_{i=1}^{N} (T^{(i)} - T^{(i)}_{\text{O!SNAP}})/T^{(i)} \times 100 \qquad (1)$$

where T and $T_{\text{O!SNAP}}$ identify the cumulative setup time, without and with snapshots respectively, for the n tests, and N is the number of repetitions of the experiment. We set $N = 30$ to increase the statistical significance of the result.

We can observe that $\bar{\mathbf{I}}_{\%}$ has always positive values. This means that opportunistic snapshots always speed up the test setup. Additionally, we observe that the average speed up increases as the number of input tests and the number of unique dependencies increase. Interestingly, opportunistic snapshotting achieves this result despite the number of shared dependencies diminishes as the number of tests increases. Our answer to RQ1 is thus clear:

Opportunistic snapshotting strongly reduces the time to setup tests in the cloud, and makes test setup up to 88% faster.

C. RQ2: Cost-Benefit Analysis of O!SNAP

Our next research question is RQ2: *Does* O!SNAP *improve the cost-efficiency of testing in the cloud compared to state-of-the art solutions?* The use of snapshots is beneficial since it reduces the time to setup tests in the cloud; however, creating snapshots takes time. This might delay the test execution, especially if snapshots are created *on-line*, i.e, on the critical execution path, since all the tests depending on a given snapshot would be delayed until such snapshot is ready. In the context of continuous testing, snapshots can be reused in subsequent test executions, and this allows to reduce overall snapshotting costs. Therefore, we study costs and benefits of opportunistic snapshotting over an extended period of time.

Table II shows the average test execution time to run all the test jobs, i.e., the makespan (*Time*), average usage of reserved (r) and on-demand (o) instances, and the additional costs for using on-demand instances (*Add. Cost*) over the observation period. In particular, r measures the average utilization of the reserved resources that are already paid for: Better schedulers fully utilize these resources. Instead, o assesses the ability of schedulers to *burst-out* [3] the test execution for a price: Better schedulers use additional resources wisely to reduce the execution time while keeping execution costs as small as possible.

TABLE II
COMPARISON OF THE AVERAGE EXECUTION TIME, RESOURCE USAGE, AND ADDITIONAL COSTS FOR EXECUTING TESTS IN THE CLOUD.

Scheduler	**Time**	**Instance #**		**Add. Cost**
name	sec	r	o	$
Sequential	10104.74	1.00	0.00	0.00
Min Load	10104.74	1.00	0.00	0.00
	5895.33	2.00	0.00	0.00
	4088.15	4.00	0.00	0.00
	3456.16	8.00	0.00	0.00
O!SNAP-**cost**	4373.97	1.00	0.00	0.00
	3562.48	2.00	0.00	0.00
	3477.62	4.00	0.00	0.00
	3451.00	8.00	0.00	0.00
O!SNAP-**time**	3449.42	1.00	3.47	91.00
	3450.06	2.00	1.03	21.30
	3448.85	4.00	0.41	8.38
	3450.21	8.00	0.03	0.26
Max Parallelism	3303.88	1.00	4.51	713.51
	3303.88	1.80	3.71	560.34
	3303.88	3.04	2.48	355.75
	3303.88	4.98	0.54	75.17

The results reported in the tables refer to the case $n = 10$.

First, we randomly draw n packages under test and replay the submission of the corresponding test jobs over the observation period. Next, for each test submission we compute a test schedule according to the scheduling algorithm and a given amount of reserved instances. Then, we measure the makespan, resources usage, and additional costs of the test execution, and aggregate them across all the submissions in the observation period and across multiple runs ($N = 30$).

We repeat the experiment for different amount of packages under test ($n \in \{5, 10, 20\}$) and allocated reserved instances ($r \in \{1, 2, 4, 8\}$), but report only the results for $n = 10$ since we noted that other values of n resulted in similar results. To better contextualize the results, we consider different schedulers and opposite configurations of O!SNAP. As schedulers we consider: **Sequential**, which executes all the test jobs on the same reserved instance; **Min Load**, which dispatches test jobs to the least loaded instances; and, **Max Parallelism**, which deploys test jobs on either free or new instances. As O!SNAP configurations we consider: O!SNAP-**cost**, which privileges execution costs over time ($\alpha = 0$, $\beta = 1$); and, O!SNAP-**time**, which privileges execution time over cost ($\alpha = 1$, $\beta = 0$).

We observe that O!SNAP can effectively trade execution time for costs and vice-versa: O!SNAP-cost achieves lower costs but longer test execution time, while O!SNAP-time achieves faster executions at higher costs. Additionally, we observe that, compared to schedulers which do not *burst-out*, O!SNAP achieves faster test executions at moderate costs; moreover, compared to massively parallelized test executions, it achieves similar test execution times but at lower costs. We thus conclude with the answer to RQ2:

Compared to standard solutions, O!SNAP *can reduce test execution time by up to 56.7%, and test execution costs by more than 87%.*

D. Threats to Validity

Where **external** validity is concerned, our technique uses real data about previous test executions to predict test execution time and costs under varying execution settings (i.e., schedules and resource allocation); and, the predicted performance may not translate to actual one. To guard against these threats, we used the official Debian data about past test executions, and we empirically measured tests' setup time on a system similar to the one currently in use by Debian.

Where **internal** validity is concerned, bugs in the Java prototype could cause problems with our results. To guard against these threats, we tested our prototype using small datasets, on which results could be manually verified.

Where **construction** validity is concerned, our measurements include the total execution time and the cost for executing the entire set of test jobs to define the cost-efficiency of each technique. Other factors, such as the cost for storing snapshots and the execution of test jobs under different cloud setups are not considered, and may be relevant.

VI. Related Work

O!SNAP enables the effective reuse of test setups among different test executions in the cloud by leveraging opportunistic snapshots, an idea initially introduced by Gambi et al. [16]. This paper builds on top of that initial idea and shows how opportunistic snapshotting can be combined with ILP scheduling, a standard solution in cloud computing, to achieve cost-efficient test executions.

Zhi et al. [17] propose *hierarchical virtual machines fork* a novel type of virtual machines that can be forked during test execution to run concurrently other tests from the same state of execution. Like O!SNAP, hierarchical virtual machines improve the efficiency of test execution; however, unlike O!SNAP, they require the availability of a non-standard virtualization technology and hierarchical test cases, which strongly limit the applicability of the approach in practice.

Beside cloud-computing, several authors show how the reuse of partial test results improves the efficiency of test execution. At the unit test level, Khalek and Kurshid [6] propose *abstract undo operations* as a mechanism to effectively share common initial executions among tests, while Bell and Kaiser [7] propose *VMVM*, a lightweight virtualization container that avoids to execute test initialization code by directly setting the application state to the one required by the test.

VII. Conclusions

During software development, there is a time for continuous testing, running and re-running tests at low cost, and a time for fast testing, getting results in a short time frame. With O!SNAP, developers can choose at any time whether they want to optimize for time, for cost, or a mix of the two. As our evaluation shows, O!SNAP can achieve significant cost and/or time savings over naive serial or parallel executions.

O!SNAP is a preliminary work and, despite the positive initial results it achieved, there are some relevant aspects of this research that we plan to address next. In particular, our ongoing work includes extending the empirical evaluation to include real test runs and widely used continuous integration systems, such as Travis CI [18]. Additionally, we plan to investigate alternative techniques, such as evolutionary algorithms and Pareto-based techniques, to solve the optimization problems at the core of O!SNAP.

VIII. Acknowledgment

This work was supported by the ERC Advanced Grant SPECMATE, by the EU FP7-PEOPLE-COFUND project AMAROUT II (n. 291803), by the Spanish project DEDETIS, and by the Madrid Regional project N-Greens Software (n. S2013/ICE-2731).

References

[1] M. Hilton et al., "Usage, costs, and benefits of continuous integration in open-source projects," in *Proceedings of the International Conference on Automated Software Engineering*, ser. ASE 2016, 2016, pp. 426–437.

[2] J. Penix, "Large-scale test automation in the cloud (invited industrial talk)," in *Proceedings of the International Conference on Software Engineering*, ser. ICSE 2012, 2012, pp. 1122–1122.

[3] M. Armbrust et al., "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, Apr. 2010.

[4] S. Elbaum et al., "Techniques for improving regression testing in continuous integration development environments," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '14, 2014, pp. 235–245.

[5] D. Marijan et al., "Test case prioritization for continuous regression testing: An industrial case study," in *Proceedings of the IEEE International Conference on Software Maintenance*, ser. ICSM '13, 2013, pp. 540–543.

[6] S. Khalek and S. Khurshid, "Efficiently running test suites using abstract undo operations," in *Proceedings of the International Symposium on Software Reliability Engineering*, ser. ISSRE '11, 2011, pp. 110–119.

[7] J. Bell and G. Kaiser, "Unit test virtualization with VMVM," in *Proceedings of the International Conference on Software Engineering*, ser. ICSE '14, 2014, pp. 550–561.

[8] M. Mao and M. Humphrey, "A performance study on the vm startup time in the cloud," in *Proceedings of the International Conference on Cloud Computing*, ser. CLOUD '12, 2012, pp. 423–430.

[9] Amazon Web Services, "Amazon EC2 purchasing options," https://aws.amazon.com/ec2/purchasing-options.

[10] P. Leitner and J. Cito, "Patterns in the Chaos – study of performance variation and predictability in public IaaS clouds," *ACM Transactions on Internet Technology*, vol. 16, no. 3, pp. 15:1–15:23, Apr. 2016.

[11] J. Vouillon and R. D. Cosmo, "On software component co-installability," *ACM Transactions on Software Engineering and Methodology*, vol. 22, no. 4, pp. 34:1–34:35, Oct. 2013.

[12] D. Bertsimas and J. Tsitsiklis, *Introduction to Linear Optimization*, 1st ed. Athena Scientific, 1997.

[13] A. Terceiro, "An introduction to the Debian continuous integration project," http://softwarelivre.org/terceiro/blog/an-introduction-to-the-debian-continuous-integration-project.

[14] "Debian continuous integration," http://ci.debian.net.

[15] "Amazon EC2 pricing," https://aws.amazon.com/ec2/pricing/.

[16] A. Gambi et al., "Improving cloud-based continuous integration environments," in *Proceedings of the International Conference on Software Engineering - Volume 2*, ser. ICSE '15, 2015, pp. 797–798.

[17] J. Zhi et al., "The case for system testing with swift hierarchical VM fork," in *Proceedings of the Workshop on Hot Topics in Cloud Computing*, ser. HotCloud '14, 2014.

[18] "Travis CI - test and deploy with confidence," https://travis-ci.com/.