One Size Does Not Fit All: In-Test Workload Adaptation for Performance Testing of Enterprise Applications

Vanessa Ayala-Rivera
Lero@UCD, School of Computer Science, University College Dublin, Ireland
vanessa.ayalarivera@ucd.ie

Maciej Kaczmarski
Lero@UCD, School of Computer Science, University College Dublin, Ireland
maciej.kaczmarski@ucdconnect.ie

John Murphy
Lero@UCD, School of Computer Science, University College Dublin, Ireland
j.murphy@ucd.ie

Amarendra Darisa
IBM Ireland, Dublin, Ireland
darisaam@ie.ibm.com

A. Omar Portillo-Dominguez
Lero@UCD, School of Computer Science, University College Dublin, Ireland
andres.portillodominguez@ucd.ie

ABSTRACT

The identification of workload-dependent performance issues, as well as their root causes, is a time-consuming and complex process which typically requires several iterations of tests (as this type of issues can depend on the input workloads), and heavily relies on human expert knowledge. To improve this process, this paper presents an automated approach to dynamically adapt the workload (used by a performance testing tool) during the test runs. As a result, the performance issues of the tested application can be revealed more quickly; hence, identifying them with less effort and expertise. Our experimental evaluation has assessed the accuracy of the proposed approach and the time savings that it brings to testers. The results have demonstrated the benefits of the approach by achieving a significant decrease in the time invested in performance testing (without compromising the accuracy of the test results), while introducing a low overhead in the testing environment.

CCS CONCEPTS

• **General and reference** → **Performance**; • **Software and its engineering** → Software testing and debugging;

KEYWORDS

Performance; Testing; Workload; Analysis; Automation

ACM Reference Format:
Vanessa Ayala-Rivera, Maciej Kaczmarski, John Murphy, Amarendra Darisa, and A. Omar Portillo-Dominguez. 2018. One Size Does Not Fit All: In-Test Workload Adaptation for Performance Testing of Enterprise Applications. In *ICPE '18: ACM/SPEC International Conference on Performance Engineering, April 9–13, 2018, Berlin, Germany*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3184407.3184418

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICPE '18, April 9–13, 2018, Berlin, Germany
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5095-2/18/04…$15.00
https://doi.org/10.1145/3184407.3184418

1 INTRODUCTION

Performance is a crucial dimension of quality and a major concern of any software project. This is especially true at the enterprise level, where system performance can play a critical role in achieving companies' goals (e.g., trading systems, airlines' websites). However, despite the efforts invested in performance engineering tasks, it is common that performance issues occur and materialise into severe problems with serious business consequences (e.g., outages on production or even cancellation of software projects). For instance, a survey conducted among information technology executives documented that half of them had experienced performance problems in more than 20% of their managed applications [12]. This situation has been exacerbated by the introduction of recent trends in information technology (e.g., Cloud Computing and Big Data) which have augmented the complexity of enterprise-level applications, complicating, even more, the performance testing of such applications [18].

A particularly complex challenge in the area is that a considerable number of performance issues, occurring at enterprise-level applications, are workload-dependent [32]. Even though existing performance testing tools (e.g., Apache JMeter [1]) can be used to detect these types of issues, this is usually ineffective because these tools use static (i.e., pre-configured) workloads. Thus, they rely on the expertise of human testers to set an adequate workload that can reveal the performance issues that might exist in the Application-Under-Test (AUT). Typically, testers use "standard" workloads (e.g., based on their own knowledge and previous experience, or based on corporate policies), which might not be sufficient in order to identify issues that may not surface even on relatively large workloads [17]. This is because it is often unclear how large is large enough for a workload to exhibit such issues (as the appropriate workload can be different for each application, or even for different versions of the same application).

An example that illustrates well the motivation of this work is the outage exhibited by the Skype network in December 2010 [6]. It lasted approximately 24 hours and affected more than 20 million users worldwide (around 90% of the Skype users at that time). The analysis performed during its fixing revealed that it was caused by an insufficient amount of performance testing [25]. In particular,

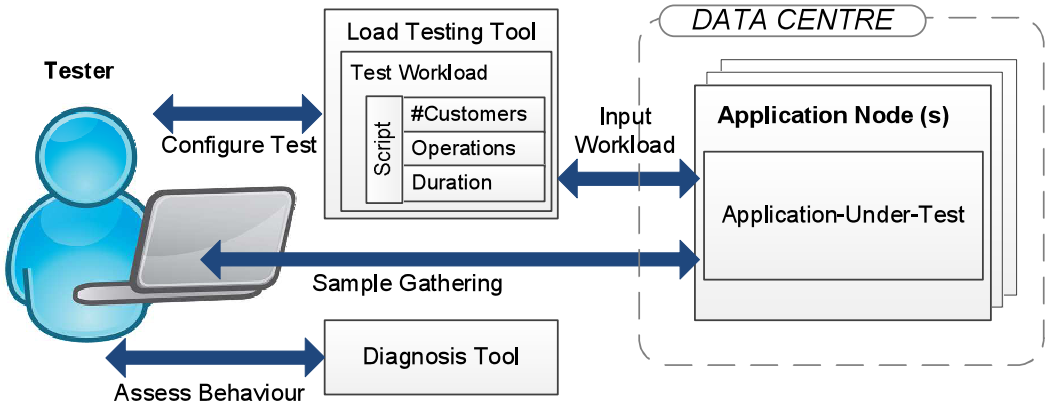

Figure 1: Performance Testing - Contextual view

an untested significantly high workload peak caused an overload of the offline messaging service, which started sending delayed responses. A specific version of the Skype clients could not process well the delayed messages and crashed. These clients included around 25% of the super-nodes (special Skype nodes that work as address books to connect the calls). Consequently, the remaining super-nodes received a traffic 100 times bigger than usual. As the super-nodes have a built-in mechanism to avoid having a huge impact on the host system, this made more super-nodes to shut down, creating a vicious cycle that ended affecting 90% of the Skype users. This scenario clearly illustrates how conducting performance testing with the right amount of workload is important to identify any performance bugs that exist on a system. Otherwise, the consequences might be very serious (like the outage experienced by Skype users). Similarly, research works have also documented the (potential) magnitude of this problem. For instance, a recent research study investigated 109 performance issues and found that 41 of them occurred due to incorrect workload assumptions [14].

Contributions. To address this challenge, our research has focused on developing techniques that improve the identification of workload-dependent performance issues, as well as their root causes, in order to increase the productivity of testers (hereinafter referred as users) by reducing the effort and expertise required in this process. In a previous work [15], we proposed an automated approach which dynamically adapts the workload used by a testing tool. However, that preliminary version was based on heuristic policies derived from the studied AUT; hence, it was not practical for real-world usage (as it was not application-independent). In this paper, we propose a new set of adaptive policies which leverage performance metrics, retrieved from the underlying AUT and evaluated in real-time, to self-configure the test workload according to the specific application behaviour. Such automated policies manage (i) when (i.e., time) and for how much (i.e., amount) the test workload needs to be modified, and (ii) to which application functionality (from the tested one) the workload will be applied. We also extend our approach to support the existing functional dependencies in the tested transactions involved, as well as to determine to which operations the workload will be more useful. As a result of the previous strategies, the need of the testers to manually configure an appropriate test workload is eliminated. Finally, we conduct a practical validation of the approach (denominated DYNAMO) consisting of an implementation prototype and a series of experiments using three different applications. They evaluate the productivity benefits (i.e., time savings) that can be achieved by using DYNAMO, as well as the amount of overhead (i.e., computational costs) introduced to the test environment.

The rest of this paper is structured as follows: Section 2 presents the background and the related work. Section 3 explains the proposed approach, while Section 4 describes the experimental evaluation and results. Finally, Section 5 presents the conclusions and future work.

2 BACKGROUND AND RELATED WORK

Performance testing is an important type of testing which aims to assess whether or not an application (i.e., AUT) will be able to perform its business functionality under a given workload [13, 19]. As shown in Fig. 1, a performance test run typically involves the execution of a performance testing tool (e.g., Apache JMeter [1]) during a certain period of time (usually several hours, or even days, in an industrial scenario) in order to apply a desired test workload to the AUT. In this context, a test workload is traditionally composed of a number of concurrent customers (normally virtual), as well as a set of functional operations (e.g., search, buy, sell), mimicking an expected type of real usage of the AUT. In order to identify performance issues, users commonly collect performance-related counters (e.g., response time, throughput) periodically during the test execution, so that their trendings and behaviours can be analysed through time. Finally, users commonly utilise some type of diagnosis tool (e.g., IBM WAIT [8]) in order to deepen their analysis of the gathered information.

The literature on performance testing has shown a variety of approaches to improve this process from different perspectives: Some research works have centred on automating the tasks related to benchmark an application (from a performance testing perspective). For instance, the work on [26] presents a framework to automatically conduct a set of typical application-related benchmarking tasks, including a workload generator. Another solution is described

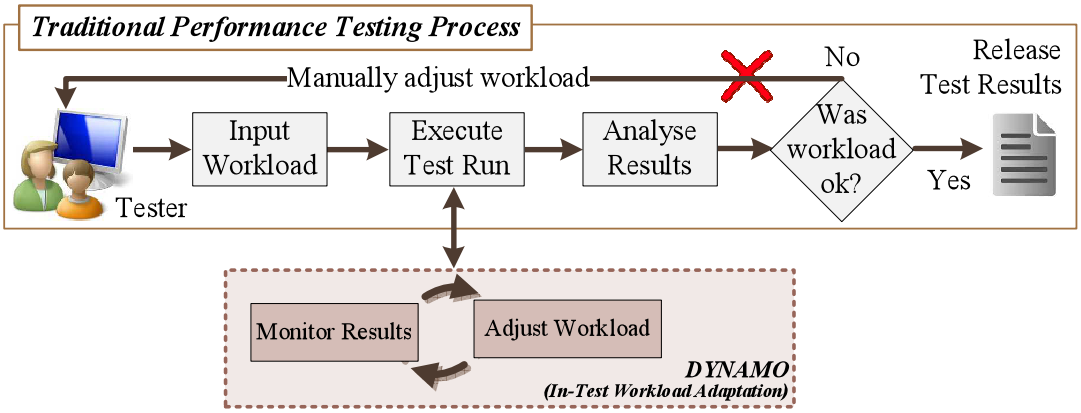

Figure 2: DYNAMO - Contextual View

in [27], which presents a cloud-based benchmark-as-a-service platform that includes a component to create traffic for the benched application. Meanwhile, the work on [29] proposes an approach to automate the extraction of workload specifications (from production logs) in order to be reused in the performance testing of session-based applications; while the work discussed in [9] aims to produce easy-to-process output (from a controlled performance testing run) in order to create a performance model of the tested application.

Meanwhile, other research works have proposed techniques to facilitate the identification of potential performance issues. For example, the authors of [16] proposed a performance test framework tailored to the particular needs of the dynamic multi-tenant cloud environments. Similarly, the work on [23] presents a technique to identify the early warning signs that typically precede a relevant performance degradation in a system. Moreover, some other works have centred on generating useful synthetic testing data [10, 11], or on providing techniques that can reduce the expertise required in order to efficiently automate the usage of the diagnosis tools in the performance testing domain [20]. In contrast to these works, which aim to improve other facets of performance testing, our solution addresses the particular need of setting a suitable test workload for a particular application; hence, successfully isolating a user from the complexities of determining such workload.

Finally, some research works have proposed the use of pre-configured workloads [28, 30] in order to simulate a real customer behaviour using Markov chains. Although these approaches can successfully mimic the desired test workload (which is based on the modelled customer behaviour), there is no guarantee that the mimicked workload is sufficient to identify any existing performance issues in the tested application. In contrast, our research work aims to address that particular challenge in the performance testing domain.

3 PROPOSED APPROACH: DYNAMO

In this section, we provide the overview of our solution and describe the processes involved, its architecture, and supporting policies.

3.1 Overview

The goal of this work was to develop an automated approach (i.e., DYNAMO) that could dynamically adjust the test workload (used in the performance testing of applications) to the specific characteristics of the underlying AUT. The aim is to shield the user from the complexities of identifying a suitable test workload, as they are normally application-dependent and can even change between versions of the same application. In this manner, users can improve their productivity as well as maximise valuable testing resources and time (as they are usually limited due to project constraints, such as budget or schedule).

As explained in Section 1, DYNAMO is motivated by the fact that current testing tools need to be manually configured with an appropriate test workload in order to avoid negative impacts on the accuracy of the test run's outputs (e.g., overlooking any relevant workload-dependant issues). This is because, if an inappropriate configuration is used, the tools might fail to obtain the desired outputs, resulting in significant time wasted. This scenario is exemplified in Fig. 3 (presented for illustrative purposes only, as the actual workload curves are application-specific), which shows how not all test workloads are typically useful to identify the workload-dependent issues existing within a system. If the workload is "too low", the issues might not surface (hence the users would overlook them). Likewise, if the workload is "too high", the test environment would get saturated. If this occurs, most of the identified issues would be caused by the environment saturation, rather than being actual application performance issues.

DYNAMO addresses these types of problems by actively adapting the workload used by a performance testing tool, so that it stresses more the application functionality which is suspicious of having a performance issue in order to have more certainty about whether or not a bug exists. This action would enhance the results obtained by a performance test run, which can be a very time-consuming activity (as discussed in Section 2). As a consequence, DYNAMO leads to a better utilisation of the available resources (in terms of workload) for a performance testing tool in order to maximise its results (e.g., identification of performance bugs or more certainty about the achieved Service Level Agreements). Internally, DYNAMO leverages policies to automatically monitor the effectiveness of the

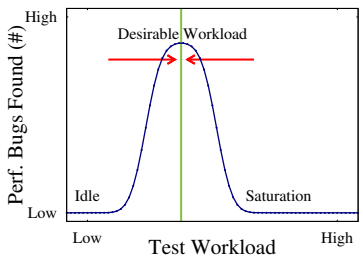

Figure 3: Bugs vs. Test Workload Trade-off

workload used by the performance diagnosis tool. Furthermore, our approach relies on a set of diagnostic metrics, which are evaluated in real-time, to determine when a workload adjustment is required for a specific transaction under test.

Figure 2 depicts the contextual view of our solution within the traditional performance testing process (discussed in Section 2). There, it can be noticed how DYNAMO enhances this process. This is achieved by proactively monitoring the intermediate test results (in real-time) in order to automatically adjust the workload (if needed) during the test run execution. Consequently, the usage of DYNAMO eliminates the need of (potentially costly) trial-and-error test cycles, which would traditionally require the intervention of the users to manually adjust the utilised test workload.

3.2 Core Process

DYNAMO performs a series of steps (i.e., core process) during the execution of the AUT. The process is depicted in Fig. 4. It starts by initialising the set of input parameters as well as the selected policies. There are two different types of policies that can be configured: (1) A diagnosis policy, which defines the criteria that will be used to decide if a transaction is suspicious of suffering a performance issue, the data sources required to perform the assessment (e.g., performance metrics), and any other specific information required to execute the policy. (2) An adjustment policy, which defines the rules to adjust (either increase or decrease) the workload whenever a change is required. These policies are described in more detail in Section 3.4.

DYNAMO requires at least one policy of each type. The encapsulation of these two types of business logic into configurable policies allows our approach to be easily extensible. The aim is that multiple policies can be developed which can be used to fulfil the requirements of different use cases. In order to fully configure DYNAMO, the tester needs to: (i) indicate how long the test run will be executed (i.e., test duration); (ii) indicate an assessment interval in order to specify how often the diagnosis policy will be evaluated to determine if a workload adjustment is needed; (iii) indicate which diagnosis and adjustment policies will be used (among the available alternatives); (iv) provide any inputs required by the chosen policies.

Once the initialisation phase finishes, the process starts the following cycle, which is performed in parallel to the performance test run execution: First, the logic awaits the configured assessment interval during which the AUT has processed a certain amount of transactions (as per an initial test workload) before any diagnosis is carried out. Next, a new set of samples is collected (based on the data sources defined in the diagnosis policy). After the collection finishes, the process checks if any transaction is suspicious of suffering a performance issue (as dictated by the criteria defined in the selected diagnosis policy). Then, if any transaction is suspicious of suffering a performance issue, the workload gets automatically adjusted. Such adjustments are controlled by the chosen adjustment policy. This process iteratively continues until the performance test run finishes. Finally, any errors that might occur are internally reported and handled.

3.3 Architecture

The design of our approach is complemented by a component-based architecture. There are three main components that compose the core process of DYNAMO (depicted in Fig. 4): The generic component contains all the functionality which is independent of the policies (e.g., the control logic of the core process). The other two components are the *action* and *decision makers*, which encapsulate the logic related to the adjustment of the workload and the diagnosis of performance issues, respectively. This architecture was designed with the aim of minimising the code changes required to extend the approach (e.g., to support other performance test tools, such as IBM RPT [4]). Along with this line of thinking, at an architectural level, the components are only accessed through interfaces. This is exemplified in Fig. 5, which presents the high-level class hierarchy of the action and decision maker components. There, it can be seen that each package contains a main interface to expose the set of supported actions, as well as an abstract class which contains all common functionality (with respect to the tools). Then, the hierarchy is easily extendable to support specific types of policies. For instance, one decision maker can leverage common performance metrics (e.g., response time, throughput, error rate) in order to assess the health of the AUT from a customer's perspective. Alternatively, another decision maker can assess the health of the AUT from a server's perspective, by measuring the amount of consumed resources (as shown in Fig. 5).

3.4 Supported Policies

In the following paragraphs, we describe the set of diagnosis and adjustment policies that DYNAMO supports:

3.4.1 Adjustment Policies. This type of policy defines the actions to follow for modifying the test workload used by the performance testing tool (once a diagnosis policy has determined that an adjustment action is required). Among the alternative approaches to develop adjustment policies for DYNAMO, we have initially focused on the following two:

- A *full house* policy, in which any modifications made to the test workload affects all transaction types (either all types increasing or decreasing at once by the same workload amount). This approach can be useful in the scenario when a new enterprise application is about to be released to production. Since there is no historical data regarding the application's performance, its reliability remains undetermined. Therefore, it is reasonable to assume that all transactions are equally likely of experiencing workload-dependent performance issues.

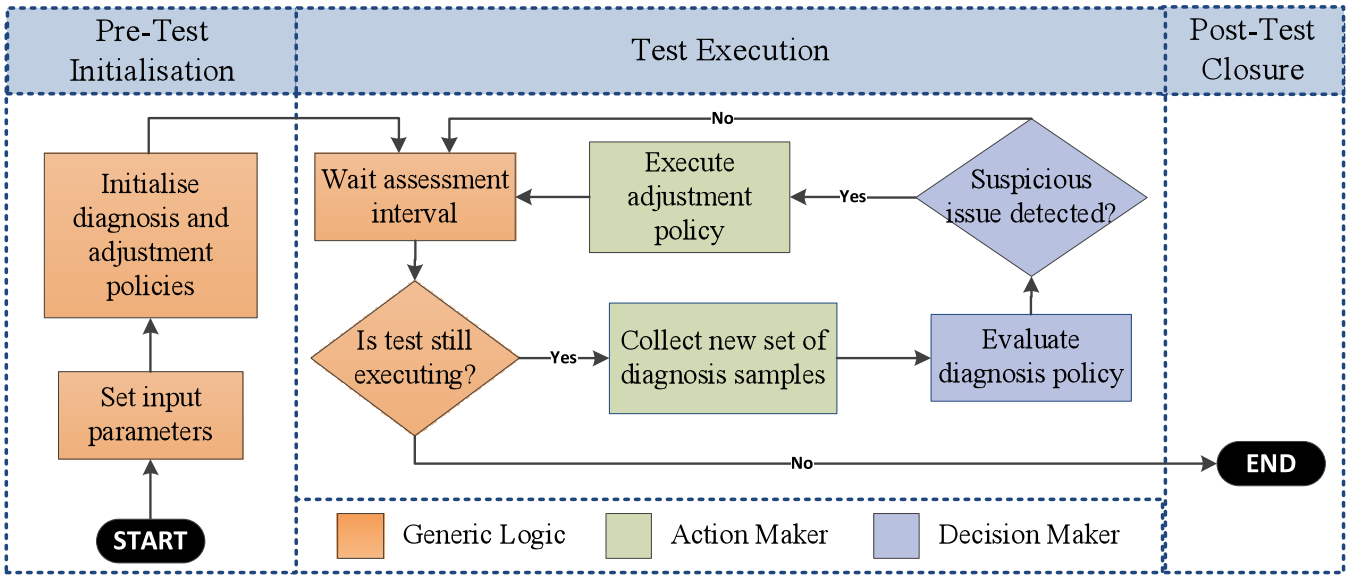

Figure 4: DYNAMO - Core Process

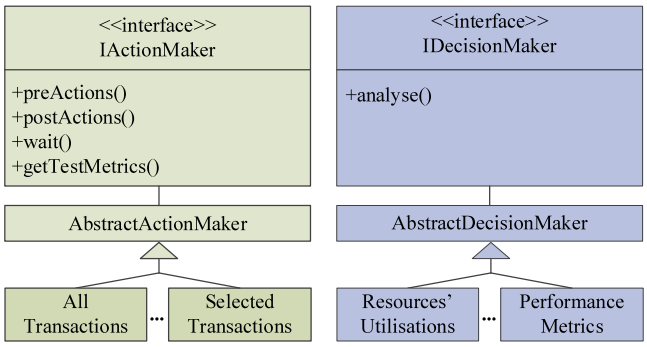

Figure 5: DYNAMO - Class Hierarchy of Decision and Action Makers

- A *mix and match* policy, in which only a subset of the tested transaction types are increased. This approach would be useful in cases when only some of the transactions are suspicious of suffering a performance issue (i.e., they are workload-dependent). It is important to remark that this policy is also responsible for preserving the functional dependencies existing among the transactions. For instance, consider that an online shopping store website is the AUT. The *"purchase"* functionality has exhibited abnormal behaviour (based on its performance indicators). Thus, it has been determined to adjust its test workload with the aim of stressing this part of the application. In this scenario, the functional dependencies of *"purchase"* can be the *"login"* and *"add to cart"* operations, as both actions need to occur before being able to buy a product. Hence, the workload of these three transaction types will be equally adjusted as they are dependent.

3.4.2 Diagnosis Policies. This type of policy is used to detect if a transaction type in the AUT is suffering a performance degradation. As a result, the test workload used by the performance testing tool might be adjusted (as specified by the adjustment policy used). Among the possible approaches to develop diagnosis policies for DYNAMO, we have initially concentrated on one policy based on the *error rate* performance metric. This policy was designed to leverage the observed behaviour that, even though it is expected that some errors may occur when processing client requests, these errors will considerably increase when the load has reached a point that exceeds the application's ability to deliver its service. The policy is also inspired by concepts of supervised machine learning, in which the test execution is divided into two phases: The first phase is exploratory, where the main goal is to identify which transactions are the most workload-sensitive; the second phase is operational, and it is mainly focused on stressing, as much as possible (and within the constraints of the test environment), those transactions that have been previously identified (in phase one) as workload-sensitive. The ultimate goal of such strategy is to maximise the number of performance issues that can be identified by a single performance test run.

This policy requires the following inputs:

(1) Phase ratio: As the policy is composed of two phases, this optional parameter indicates what percentages of the total test duration will be used for the first and second phases, respectively. If the values for this parameter are not configured, a default ratio of 20/80% will be used. This default value has been taken from the Pareto law, which states that, for many events, roughly 80% of the total effects come from 20% of the causes [24].
(2) Initial workloads: As a starting calibration point, the user needs to indicate a known low workload (e.g., 50 customers) and a relative ratio where the workload sensitivity might

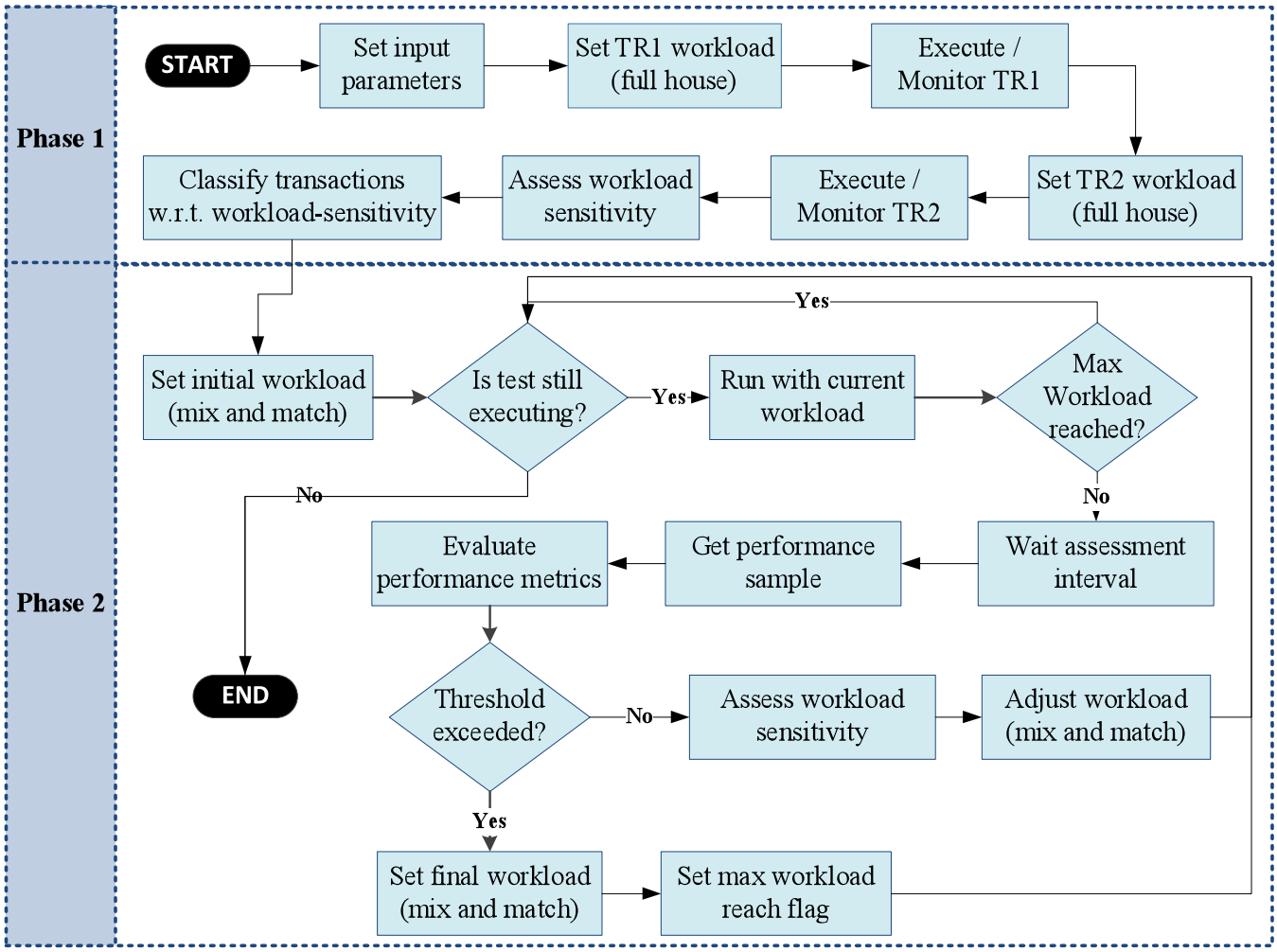

Figure 6: DYNAMO - Error rate-based Adjustment Policy

be noticeable (e.g., 10 times). Understanding that the user might not know the exact test workload, it is only assumed that s/he can still provide an educated estimation.

(3) Error rate threshold: This optional parameter defines the upper bound value that indicates that the system has become saturated. If this parameter is not configured, a default value of 90% will be used (as this is a value commonly identified as saturation point [19]).

(4) Functional dependencies: This is the set of dependency relationships that might exist between the tested transactions (e.g., a purchase cannot occur without first logging into the system). If configured, such dependencies will be considered by the policy's logic to keep the test workload consistent (by propagating the dependencies to the applicable adjustment policy).

(5) Workload-sensitivity identification strategies: This parameter defines which strategy (among the ones supported for each of the two phases) will be used to assess the workload-sensitivity of the transactions to determine if their workload needs to be adjusted. They are explained in the following paragraphs (as part of the phases' descriptions).

The core process of the *error-based adjustment policy* is depicted in Fig. 6. Below, we explain the two main phases that compose the execution of the policy.

Phase 1: Here, the goal is to find what transactions, among the overall set of functional transactions been tested, are workload-sensitive (WkS) or workload-insensitive (WkI). First, all the elements required to properly execute the policy are initialised. Next, two initial Test Runs are conducted (TR1, TR2); each test lasts half of the time configured for phase 1. TR1 is executed using the initial test workload provided by the user (i.e., a low workload such as 50 customers), while TR2 uses an incremented test workload that results from multiplying the initial low workload by the relative ratio parameter (i.e., 500 customers, assuming a ratio of 10 and an initial workload of 50 customers). Internally, this phase leverages the *full house* adjustment policy (i.e., all transactions are equally modified at the same time). This policy is suitable for this scenario because, at this stage, it is still uncertain which transactions are

workload-sensitive and which are not. During the execution of the test runs, a set of performance metrics is gathered (i.e., response time, throughput, error rate). Once TR2 finishes, the process will assess the workload-sensitivity of the transactions by comparing the differences (i.e., deltas) between the performance metrics gathered from TR1 and TR2.

Two different strategies for workload-sensitivity identification are supported: (1) *Absolute*, in which all the transactions whose delta percentage is greater than or equal to a pre-configured value (e.g., 20%) are tagged as WkS; the rest are tagged as WkI. (2) *Relative*, in which all the transactions are sorted based on their deltas and only the top N transactions (either using an absolute number, such as 5, or a percentage, such as 20%) are tagged WkS; the rest are tagged as WkI. It is worth mentioning that the functional dependencies among the transactions are respected. This means that, if a WkI transaction has a functional dependency with a WkS transaction, the WkI transaction will be altered to tag it as WkS in order to respect that dependency. The final output of phase 1 is the set of transactions tagged as either WkS or WkI.

Phase 2: Here, the aim is to increase the test workload of the WkS transactions as much as possible without reaching the saturation point of the system. When phase 2 starts, the transactions will use an initial workload using the *mix and match* adjustment policy. For WkI transactions, this will be half of the workload used in TR2 of phase 1, while the WkS transactions will use the full workload. These values are configurable, however, their default values have been defined with the aim of accelerating the identification of performance issues. For instance, the last workload used in TR2 of phase 1 has already proven to be effective as it was used to identify the WkS transactions. Also, since the adaptive logic of phase 2 only applies to WkS transactions, WkI transactions can use the initially configured workload during the rest of the test execution. In that manner, they do not spend resources that can be invested in the testing of WkS transactions.

Once the initial workloads are set, the following loop will be executed until the configured test duration has passed: First, the process awaits the current assessment interval, so that the test run can make use of the current test workload for some time before evaluating whether or not it is the right workload for the particular application's behaviour. Once the interval has elapsed (and assuming that the maximum suitable workload has not been reached), a sample of the performance metrics is retrieved from the test loader tool (i.e., the error rate for all WkS transactions that comprehend the most recent assessment interval). Then, the deltas for the sampled error rates are calculated for each WkS transaction (as previously explained in phase 1). These deltas are then used to determine if the workload of any individual transaction type requires being adjusted. Before doing this, DYNAMO first needs to assess if the test run has not reached the saturation point yet. In order to assess this, the average error rate is compared to the configured threshold. If the average error rate (across all tested transactions) is higher than the threshold, it means that the test environment is already saturated. In this case, a final adjustment to the test workload is done by rolling it back to the one used before the last increment (as such workload is the highest one reached before exceeding the error rate threshold). Also, a flag is set to indicate that the maximum workload point has been reached, so that DYNAMO stops modifying the workload.

On the contrary, if the saturation point has not been reached yet, the process checks which of the WkS transaction(s) will be increased. To determine this, the policy can use three different selection strategies (inspired by the commonly-known load balancing algorithms, random and round-robin): (1) *Random:* Here, the transaction whose workload will be increased is randomly picked. This can be either a uniform or a weighted random selection. In the uniform mode, all the transactions have the same probability of being selected. In the weighted mode, the transactions are weighted so that the probability of each transaction to be selected is determined by its relative weight. For example, based on their performances, the worst performing transaction would have more chances to be chosen. (2) *Maximum*: Here, the workloads of the top N transactions that have the worst performance are increased. In order to avoid a "selfish" behaviour, the workload of a transaction cannot be increased in two consecutive adjustment rounds. (3) *Minimum*: Here, the workloads of the top N transactions that have the best performance are increased. This alternative tends to be fairer (than maximum) because it is more likely that the transaction whose workload was increased will be the worst performer in the next round. Hence, this strategy gives other transactions the chance to have their workloads increased (without the need to keep track of the adjusted transactions, as in the maximum strategy).

The loop continues until the performance test run finishes. It is worth mentioning that any exceptions are internally reported and handled. Furthermore, similarly to the previous phase, the functional dependencies that exist among the transactions are always respected. This means that, if the workload of a transaction needs to be modified (as per any of the previously discussed adjustment strategies), all its functional dependencies must also be modified accordingly.

4 EXPERIMENTAL EVALUATION

4.1 Experimental Setup

Our experiments aimed to evaluate the benefits brought by DYNAMO (e.g., its bug accuracy and time savings), as well as the costs of its usage (e.g., its computational resources). All experiments were carried out in an isolated test environment to prevent environmental noise (so the entire load was controlled). The environment was composed of two virtual machines (VMs): One acted as application node (running Apache Tomcat 6.0.35 [2], a widely used Java Application Server); while the other VM acted as load tester (running Apache JMeter 2.9 [1], a popular performance testing tool). Each VM had 4GB of RAM, 2 CPUs at 2.20GHz, Linux Ubuntu 12.04L, and OpenJDK JVM 7 with a 1.6GB heap. From a technical perspective, we built our prototype on top of the JMeter tool, developing it in Java [21]. This was done in order to make our solution highly portable, as there are Java Virtual Machines (JVM) available for most contemporary operating systems.

As AUT, we used three different applications:

(1) PetStore, an e-commerce application commonly used in the literature [19, 22]. It is composed of 11 different business operations. They are described in Table 1.

Table 1: PetStore Operations

Name	Description
Index	Welcome page.
Add user	Page to register a new user.
Login	Page to perform the login action.
Login button	Process to validate the user and password.
Search product	Search page to find particular items.
Select product	This action will display the details of the chosen product.
Update cart	Add the selected product to the shopping cart.
New order	Page displayed after the first item is added to provide general information about the order.
Add payment	Page where the payment information is captured.
Confirm order	The final step in the purchase process, where the order is confirmed.
Log out	Link to perform the logout action.

(2) m-PetStore, a modified version of Petstore that mimics the scenario of an evolving AUT in which ten performance issues were injected (a blend of lock contention, I/O latency, and deadlock bugs) following a strategy previously used in other works [20].
(3) DaCapo, one of the Java benchmarks most widely-used in the literature, offering a wide range of 14 real-life programs from different business domains [3] (shown in Table 2). To enable the execution of any DaCapo program from within a test script, a wrapper JSP was also developed and deployed on the application node. Finally, each program execution was considered a transaction.

As evaluation criteria, we adopted the following metrics and units: Throughput (tps), response time (ms), error rate (%), CPU (%) and memory (MB) utilisations. Also, the analysed performance issues were retrieved from the outputs of IBM WAIT (popular Java diagnosis tool used due to its strong analytic capabilities to detect performance bugs [8, 31]). In order to do this, WAIT was fed with Javacores [5] (snapshots of the JVM state). They were generated with the native Linux *kill* command (i.e., no instrumentation was required to create them) and collected every 30 secs (following a sampling interval commonly used in the industry [19]). Finally, a 2-hour test duration (per test run) was used to reflect realistic test conditions. Besides, three types of test runs were performed:

(1) The first type used the traditional approach of static workloads (as per common industrial practices [7]). For this purpose, all the AUTs started with a close-to-idle scenario an ended with a saturated environment. However, since the AUTs have diverse functional behaviours, their saturation points were different. Consequently, the workload ranges (as well as the workload increments) varied per AUT. For PetStore, the workload range was [100..1800] in increments of 100; for m-PetStore, it was [100..3500] in increments of 100 (up to 2300) and then 300; and for DaCapo, it was [200..4400] in increments of 200 (up to 3600) and then 400. These configurations exemplify the challenges typically experienced by users to select an appropriate test workload.
(2) The second type of run used the preliminary version of DYNAMO based on heuristic policies derived from PetStore (referred as h-DYNAMO [15]).
(3) The third type of run used the work proposed in this paper which adopts our new adaptive logic (DYNAMO). It involved a 5-minute assessment interval and a 20%-80% phase-ratio. The initial workload for both PetStore versions was 100 (with a ratio of 4), while for DaCapo was 800 (with a ratio of 2). Additionally, the error rate threshold was set to 8%, using the *relative* strategy for phase 1 (to homogenise DYNAMO's configuration, as the AUTs were composed of different numbers of functional transactions), and the *minimum* strategy for phase 2 (to give all WkS transactions a fair possibility of getting stressed).

Table 2: DaCapo Programs

Name	Description
avrora	It simulates a set of programs running on a grid of microcontrollers.
batik	It processes a set of vector-based images.
eclipse	It executes a set of performance tests in an eclipse development environment.
fop	It generates PDF files based on a set of XSL-FO files that are parsed and formatted.
h2	It executes a set of banking transactions against a database-centric application.
jython	It executes a set of python scripts in Java.
luindex	It indexes a set of documents.
lusearch	It performs a set of keyword searchs over a corpus of data.
pmd	It reviews a set of Java classes, looking for bugs in their source code.
sunflow	It renders a set of images.
tomcat	It executes a set of queries against a Tomcat server.
tradebeans	It executes a set of stock transactions, via Java Beans calls.
tradesoap	It executes a set of stock transactions, via SOAP calls.
xalan	It transforms a set of XML files into HTML files.

4.2 Experimental Results

In this section, we present the results obtained, discussing them in terms of the relevant perspectives: bug accuracy, time savings, and computational costs. Due to space constraints, we only present the most relevant results (as this experiment involved above 140 hours of test run executions).

4.2.1 Performance Bugs Analysis. Our analysis first focused on assessing the bug accuracy of all test runs. The obtained results are presented in Figs. 7, 8 and 9, which compare the number of

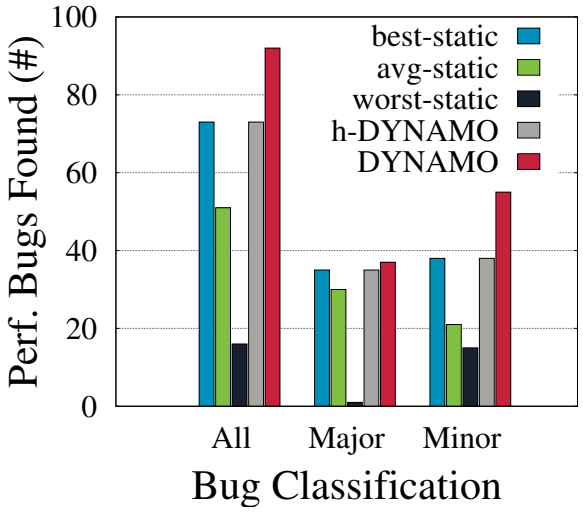

Figure 7: PetStore

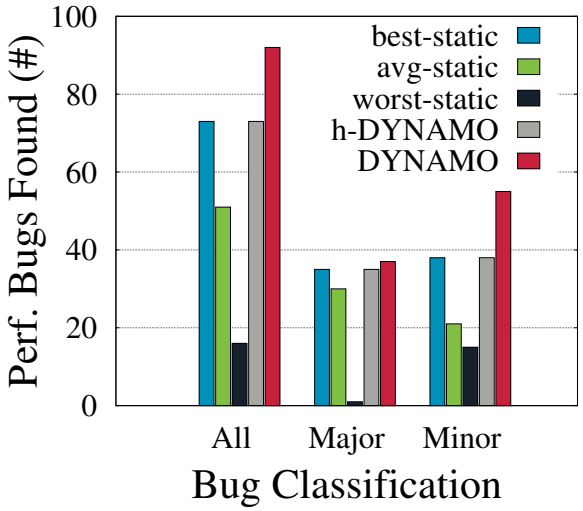

Figure 8: m-PetStore

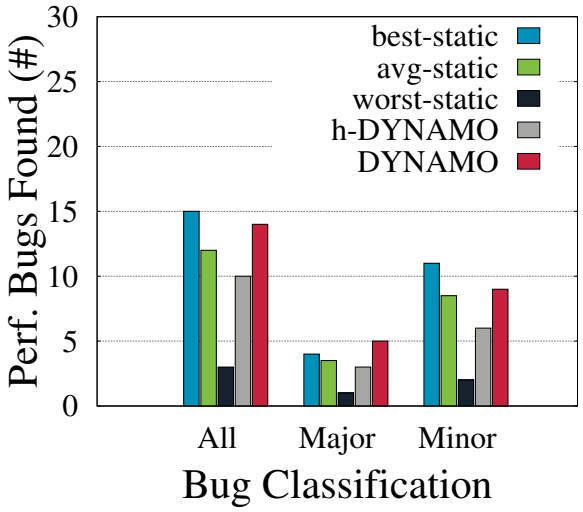

Figure 9: DaCapo

performance bugs found in each test run for the three AUTs. For the static type test run, we report the best (best-static), worst (worst-static), and average (avg-static) performing workload. It can be observed how DYNAMO worked well, as it was able to identify more bugs than all of the other test runs. The only exception was the best-static test run of DaCapo, which found one more bug than DYNAMO.

The improvements in bug identification accuracy achieved by DYNAMO were the result of increasing the test workload for those functional transactions which were the most workload-sensitive. This is because such workload adjustments provoked that the workload-sensitive transactions were considerably more frequently executed (compared to the static test runs) during DYNAMO's test run executions. These behaviours, which were captured by the sampled Javacores, allowed to better feed the diagnosis tool (i.e., WAIT), which was pushed to do a more detailed analysis of the AUTs (as those samples contained more information about the most workload-sensitive transactions, which typically are the main causes of the workload-dependent issues existing in an application). For instance, in the case of PetStore (whose results as shown in Fig. 7), DYNAMO's phase 1 identified 6 highly workload-sensitive operations (among the 11 which compose the application). This information allowed that the additional test workload (introduced during phase 2) could concentrate on those 6 operations. Thus, the workloads of those transactions were gradually increased (during the assessment intervals, based on their error rate deltas) until reaching a peak of 1300 concurrent virtual customers for the 3 transactions which were more frequently chosen for workload increase (i.e., search, select product, and add item). On the contrary, other transactions were only occasionally chosen (e.g., login), causing that the workload used to test them were relatively lower (e.g., login only reached a maximum of 700 concurrent virtual customers).

Similar trends were observed for the other two AUTs (i.e., m-PetStore and DaCapo), as comparable improvements in terms of bug accuracy were obtained (as shown in Figs. 8 and 9). However, some (expected) differences in terms of the results were obtained due to their diverse application behaviours. For example, the highest test workload reached by a transaction in m-PetStore was 1100 customers. Moreover, this test run was the only one (among those that used DYNAMO) which triggered a rollback action (i.e., decrease the test workload to the previously used amount) because the error rate threshold was exceeded at some moment of the test run. This exemplifies how even different versions of the same application can require considerably different test workloads. Finally, 4 transactions were the most workload-sensitive ones in DaCapo (i.e., avrora, batik, sunflow, and xalan), whose highest transaction-level workloads were 1400, 1400, 1700, and 1600 customers (respectively).

To offer a more comprehensive perspective of the results, we also performed a breakdown of the bugs by classifying them based on their frequency of occurrence. A bug was labelled as *major* if it occurred above 5% of the test run duration (i.e., 2 hours). Otherwise, it was considered as *minor*. This analysis confirmed the results discussed previously, as DYNAMO always outperformed worst-static, avg-static, and best-static (except in DaCapo). Regarding the best-static of DaCapo, the results showed an interesting finding: DYNAMO found more major bugs (typically the ones that matter

the most from a performance perspective). Therefore, best-static of DaCapo only surpassed DYNAMO in terms of minor bugs.

Moreover, it is worth noting that DYNAMO always outmatched h-DYNAMO, even for PetStore (which is the AUT from which h-DYNAMO was derived). This was the result of two main factors: Firstly, h-DYNAMO used the same workload for all types of tested transactions. However, this strategy was not optimal because not all transactions suffered performance issues. Thus, stressing more some types of transactions (i.e., those with suspected problems), while leaving the others to use lower test workloads (like DYNAMO did) produced better results in terms of bug accuracy. Secondly, h-DYNAMO did not adapt to the AUT's behaviour (e.g., its saturation point), but it only reused the test workload that was useful for PetStore. Not surprisingly, it was not an optimal test workload for the other AUTs. Finally, the differences between h-DYNAMO and DYNAMO were relatively small because the same test environment was used for all test runs. If an alternative (i.e., bigger) test environment were used, the differences would be far more notorious (as h-DYNAMO would not be able to escalate to the characteristics of the new test environment, while DYNAMO would do).

4.2.2 Testing Time. The second part of our analysis centred on assessing the time savings achieved with our solution. Since DYNAMO was able to adjust the workload during the test run execution, it avoided the need for costly trial-and-error test runs. More specifically, the user only required one test run (instead of the 18, 27, and 20 runs required to cover the full range of static workloads for PetStore, m-PetStore, and DaCapo, respectively). This means that DYNAMO reduced the duration of the total performance testing activities by an average of 95% across the three AUTs (as DYNAMO was able to avoid the execution of 17, 26, and 19 test runs, for PetStore, m-PetStore, and DaCapo, respectively). This is depicted in Figs. 10, 11 and 12, which compare the testing time of the static runs against DYNAMO for each AUT. h-DYNAMO is not included in the figures as it behaves similarly to DYNAMO in terms of execution time. Moreover, the differences in the total time of the static test runs across AUTs is the result of using different ranges of test workloads for each AUT (due to their diverse functional behaviours, as explained in Section 4.1).

Finally, in order to provide a more conservative analysis of these results, we have also included another series in the figures (i.e., bad static runs) in order to indicate the time invested in the static test runs where DYNAMO was better (in terms of bug accuracy). Even under this conservative analysis, significant time savings were obtained. This is because only one static test run (out of 65) obtained marginally better results than DYNAMO.

4.2.3 Cost Analysis. We also measured the computational resources required by DYNAMO in order to understand the costs of using our solution. We focused on the JMeter (load tester) node because DYNAMO resides there. The obtained results are shown in Figs. 13, 14, and 15, which depict the average CPU and memory utilisations during the test runs' executions. It can be noted how the test runs were more memory-intensive than CPU-intensive. The main factor behind the amount of consumed resources was the test workload (e.g., dictating the number of threads that are used to mimic the virtual customers' behaviours). Consequently,

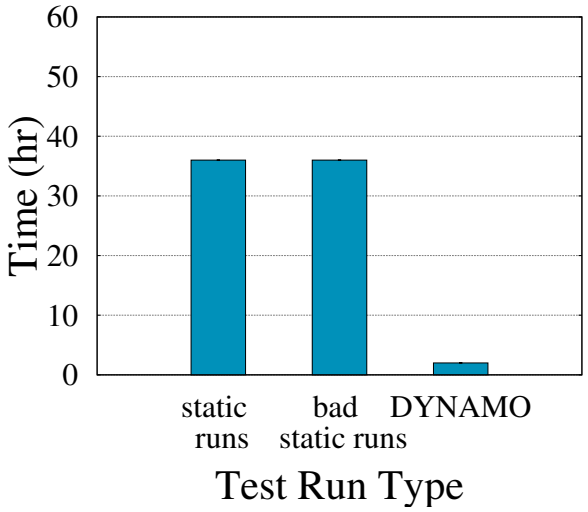

Figure 10: PetStore

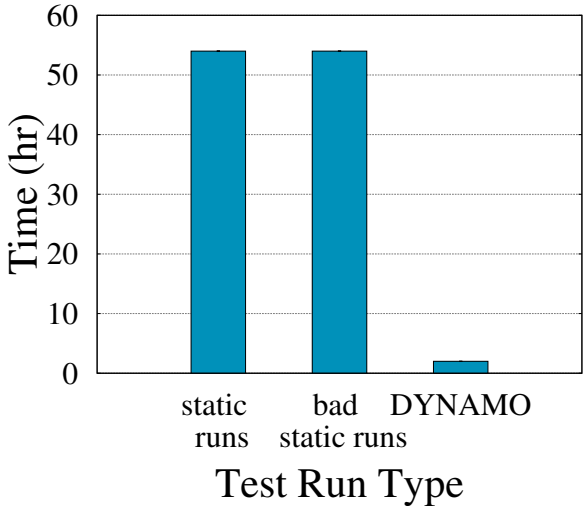

Figure 11: m-PetStore

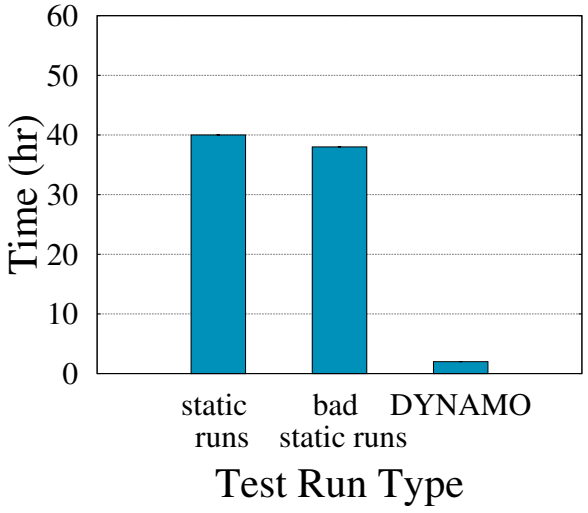

Figure 12: DaCapo

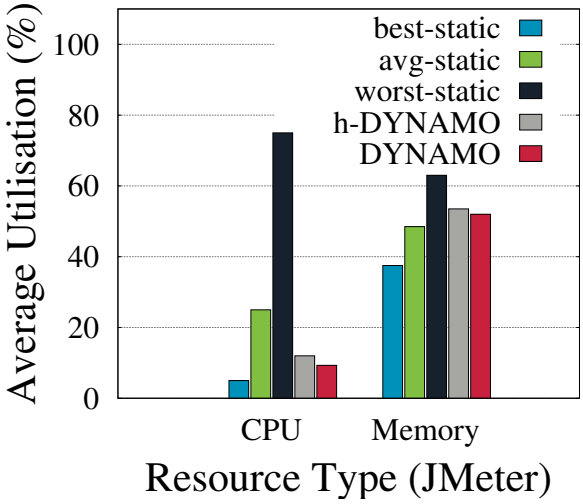

Figure 13: PetStore

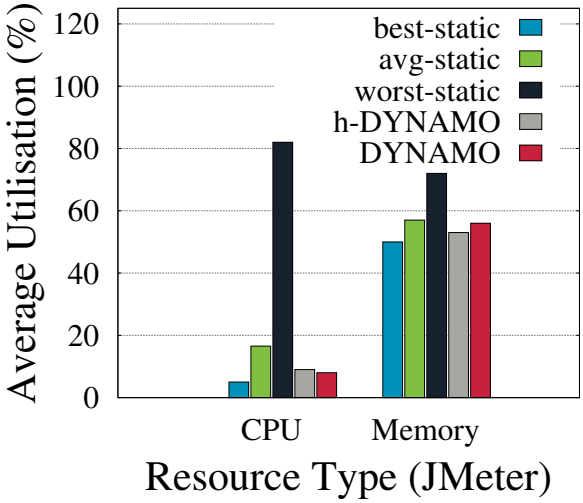

Figure 14: m-PetStore

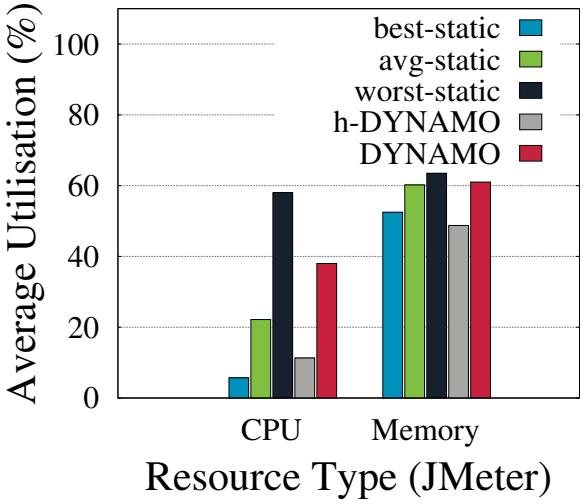

Figure 15: DaCapo

the amount of resources varied among the three AUTs (as each application reached a different optimal test workload).

In the case of both PetStore versions, since the average test workload (across all tested transactions) was relatively lower than DaCapo's (ranged between 900 and 1200 customers for PetStore and between 3000 and 3500 for DaCapo), the costs of DYNAMO were closer to the best-static (in terms of CPU) and average-static (in terms of memory). Meanwhile, for DaCapo, the costs were closer to the worst-static. It is important to highlight that the best-static was always the static test run with the lowest test workload (per AUT), which not surprisingly, derived in a poor performance in terms of bug accuracy. Finally, it can also be seen on the figures how h-DYNAMO exhibited the same computational costs across the three AUTs. This was the result of not adapting itself to the specific behaviour of each AUT.

4.2.4 Final Discussion. In our evaluation, we utilised three applications to assess, to a certain degree, the generality of the benefits and costs of DYNAMO. As the results have shown, DYNAMO offered substantial time savings across all the AUTs. Based on our in-depth analysis, it is expected that DYNAMO can yield similar results with other applications. However, additional experiments would provide more certainty about the broader applicability of our approach (e.g., the usefulness of its default values). It is also possible to conclude that similar bug accuracy results can be obtained when using other diagnosis tools, as long as they are capable of detecting the same types of performance bugs tested.

Additionally, the proposed approach assumes that the user can provide an educated estimation of a known low workload (e.g., 50 customers) and a relative ratio where the workload sensitivity might be noticeable (e.g., 10 times). We consider this is reasonable because the aim is only to identify which transactions are, relatively, more workload-sensitive than the others. Similarly, it is also assumed that the user knows the AUT well-enough to indicate the functional dependencies. For instance, that a user needs to log in before making a purchase, or that a logout cannot be done without first logging in. This is a fair input at enterprise level because that information is usually documented during the analysis phase of the software development cycle (e.g., in the requirements traceability matrix).

Regarding the strategy followed to conduct the static test runs, we explored the full spectrum of test workloads in our planned range (by gradually increasing the test workload linearly) because this is a practice commonly used in the industry [7]. Moreover, this strategy allowed us to have certainty of when the saturation point was reached (as it was application-specific). However, we understand that other test strategies might be used. For instance, instead of linearly increasing the static test workloads in a range, a user might prefer to start his/her performance test work by choosing two very different static workloads (within the planned range) and then, based on the results of those test runs, select only other workloads in the range that look most promising to reveal performance issues. Thereby, it might not be required to execute the same number of static test runs than those conducted in this paper. Hence, the results for the total testing time of the static approach (as well as the time savings achieved by DYNAMO) might be different from those obtained in our experiments. Despite those differences, from the outcome of our investigation, it is expected that the time

savings gained by using DYNAMO would be still significative. This is because, even in the extreme case of only requiring 2 static test runs (which would be very unlikely to achieve, as even an experienced user might struggle to easily identify the exact test workload needed for different AUTs), DYNAMO would still be able to save 50% of the testing time (by only requiring 1 test run, instead of the 2 used by the static approach).

Finally, we understand that, even though there is a rationale behind our chosen default values in our experimental evaluation, and they have proven useful for the AUTs used, they might not be applicable to all scenarios. Thus, we plan to explore the range of possible values for each of our configuration parameters (e.g., the ratio between phases 1 and 2) in order to develop guidelines that can help practitioners in the usage of DYNAMO.

5 CONCLUSIONS AND FUTURE WORK

This paper presented an automated approach (DYNAMO) which can adapt, in real-time, the workload used by a performance testing tool during the test run execution. Thus, it eliminates the need for manually identifying a suitable test workload, as well as costly trial-and-error test runs. A prototype was built on top of the JMeter tool and a series of experiments were conducted in order to assess DYNAMO's benefits and costs. Our experimental results have proved the usefulness of the approach by significantly reducing the testing time (by an average of 95%, compared to a range of static standard workloads), without compromising the accuracy of the test results. This is demonstrated by the fact that DYNAMO achieved a high bug accuracy (as it always identified more relevant performance bugs than the best test run counterparts, among the ones using static workloads). Moreover, only a moderate overhead was introduced by DYNAMO (i.e., the average CPU utilisation, in the node where DYNAMO resides, never exceeded 40%).

In terms of future work, our research will centre on investigating how best to extend the capabilities of the approach. For instance, by assessing which other types of metrics (other than the performance ones) can be leveraged to enhance the accuracy of the approach (e.g., the outputs of a diagnosis tool), or by reducing the number of required input parameters (e.g., the need of manually providing the functional dependencies existing among the tested operations). Additionally, we plan to keep assessing the benefits and costs of DYNAMO through broader experiments with the aim of strengthening its validation. For instance, by diversifying the composition and size of the test environments, the tested applications, and the duration of the test runs. The aim is to develop guidelines to help practitioners to configure/use DYNAMO more easily. Finally, we also plan to make the tool freely available (e.g., as a web service).

6 ACKNOWLEDGMENTS

This work was supported, in part, by Science Foundation Ireland grant 13/RC/2094 and co-funded under the European Regional Development Fund through the Southern & Eastern Regional Operational Programme to Lero - the Irish Software Research Centre (www.lero.ie).

REFERENCES

[1] Apache JMeter. http://jmeter.apache.org/. Last accessed: 2018-02-01.
[2] Apache Tomcat. http://tomcat.apache.org/. Last accessed: 2018-02-01.
[3] DaCapo Benchmark. http://dacapobench.org/. Last accessed: 2018-02-01.
[4] IBM RPT. http://www-03.ibm.com/software/products/en/performance. Last accessed: 2018-02-01.
[5] Javacores. http://www-01.ibm.com/support/docview.wss?uid=swg27017906&aid=1. Last accessed: 2018-02-01.
[6] Post-mortem on the Skype outage. http://beeyeas.blogspot.mx/2014/01/cio-update-post-mortem-on-skype-outage.html. Last accessed: 2018-02-01.
[7] Performance Workload Design. Technical report, IBM, 2013.
[8] E. Altman, M. Arnold, S. Fink, and N. Mitchell. Performance analysis of idle programs. *ACM SIGPLAN Notices*, 45(10), Oct. 2010.
[9] V. Apte, T. Viswanath, D. Gawali, A. Kommireddy, and A. Gupta. Autoperf: Automated load testing and resource usage profiling of multi-tier internet applications. In *ICPE*, 2017.
[10] V. Ayala-Rivera, A. O. Portillo-Dominguez, L. Murphy, and C. Thorpe. COCOA: A synthetic data generator for testing anonymization techniques. *Privacy in Statistical Databases*, 2016.
[11] M. W. Aziz and S. A. B. Shah. Test-data generation for testing parallel real-time systems. In *ICTSS*, 2015.
[12] Compuware. *Applied Perf. Management Survey*. 2007.
[13] Z. M. Jiang. Automated analysis of load testing results. *ISSTA*, 2010.
[14] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. *ACM SIGPLAN Notices*, 47(6):77–88, 2012.
[15] M. Kaczmarski, P. Perry, J. Murphy, and A. O. Portillo-Dominguez. In-test adaptation of workload in enterprise application performance testing. In *International Conference on Performance Engineering Companion*, 2017.
[16] N. Michael, N. Ramannavar, Y. Shen, S. Patil, and J.-L. Sung. Cloudperf: A performance test framework for distributed and dynamic multi-tenant environments. In *ICPE*, 2017.
[17] I. Molyneaux. *The Art of Application Performance Testing: Help for Programmers and Quality Assurance.* " O'Reilly Media, Inc.", 2009.
[18] A. O. Portillo-Dominguez, J. Murphy, and P. O'Sullivan. Leverage of extended information to enhance the performance of JEE systems. *Information Technology and Telecommunications Conference*, 2012.
[19] A. O. Portillo-Dominguez, P. Perry, D. Magoni, and J. Murphy. PHOEBE: an automation framework for the effective usage of diagnosis tools in the performance testing of clustered systems. *Software: Practice and Experience*, 2017.
[20] A. O. Portillo-Dominguez, M. Wang, J. Murphy, and D. Magoni. Automated wait for cloud-based application testing. *International Conference in Software Testing Workshops*, 2014.
[21] A. O. Portillo-Dominguez, M. Wang, J. Murphy, and D. Magoni. Adaptive gc-aware load balancing strategy for high-assurance java distributed systems. In *International Symposium on High Assurance Systems Engineering*, 2015.
[22] A. O. Portillo-Dominguez, M. Wang, J. Murphy, D. Magoni, N. Mitchell, P. F. Sweeney, and E. Altman. Towards an automated approach to use expert systems in the performance testing of distributed systems. *Workshop on Joining AcadeMiA and Industry Contributions to Test Automation and Model-Based Testing*, 2014.
[23] R. Ramakrishnan and A. Kaur. Technique for detecting early-warning signals of performance deterioration in large scale software systems. In *ICPE*, 2017.
[24] K. T. Rosen and M. Resnick. The size distribution of cities: an examination of the pareto law and primacy. *Journal of Urban Economics*, 8(2):165–186, 1980.
[25] D. Rossi, M. Mellia, and M. Meo. Evidences behind Skype outage. In *ICC*, 2009.
[26] P. Shivam, V. Marupadi, J. S. Chase, T. Subramaniam, and S. Babu. Cutting corners: Workbench automation for server benchmarking. In *USENIX Annual Technical Conference*, 2008.
[27] A. Tchana, B. Dillenseger, N. De Palma, X. Etchevers, J.-M. Vincent, N. Salmi, and A. Harbaoui. Self-scalable benchmarking as a service with automatic saturation detection. In *ICDSP*, 2013.
[28] A. Van Hoorn, M. Rohr, and W. Hasselbring. Generating probabilistic and intensity-varying workload for web-based software systems. *ICPE*, 2008.
[29] C. Vögele, A. van Hoorn, E. Schulz, W. Hasselbring, and H. Krcmar. WESS-BAS: extraction of probabilistic workload specifications for load testing and performance prediction - a model-driven approach for session-based application systems. *Software and Systems Modeling*, Oct 2016.
[30] E. Weyuker and A. Avritzer. A metric for predicting the performance of an application under a growing workload. *IBM Systems Journal*, 41(1):45–54, 2002.
[31] H. Wu, A. N. Tantawi, and T. Yu. A self-optimizing workload management solution for cloud applications. *ICWS*, 2013.
[32] E. Xiao, Xusheng. Context-Sensitive Delta Inference for Identifying Workload-Dependent Performance Bottlenecks. *ISSTA*, 2013.