Improving the Testing of Clustered Systems Through the Effective Usage of Java Benchmarks

A. Omar Portillo-Dominguez*, and Vanessa Ayala-Rivera*
*Lero@UCD, School of Computer Science
University College Dublin
Dublin, Ireland
{andres.portillodominguez,vanessa.ayalarivera}@ucd.ie

***Abstract*—Nowadays, cluster computing has become a cost-effective and powerful solution for enterprise-level applications. Nevertheless, the usage of this architecture model also increases the complexity of the applications, complicating all activities related to performance optimisation. Thus, many research works have pursued to develop advancements for improving the performance of clusters. Comprehensively evaluating such advancements is key to understand the conditions under which they can be more useful. However, the creation of an appropriate test environment, that is, one which offers different application behaviours (so that the obtained conclusions can be better generalised) is typically an effort-intensive task. To help tackle this problem, this paper presents a tool that helps to decrease the effort and expertise needed to build useful test environments to perform more robust cluster testing. This is achieved by enabling the effective usage of Java Benchmarks to easily create clustered test environments; hence, diversifying the application behaviours that can be evaluated. We also present the results of a practical validation of the proposed tool, where it has been successfully applied to the evaluation of two cluster-related advancements.**

***Index Terms*—Software Testing, Java, Clusters, Performance**

I. Introduction

Performance is a major concern of any software project. This is especially true at enterprise-level, as system performance plays a key role in using the software to achieve business goals [1]. However, it is not uncommon that performance issues do occur, even materialising into serious problems (e.g., outages on production environments or cancellation of projects). Many research studies have documented the magnitude of this problem. For example, the authors of [2] recently found 332 previously unknown performance problems in the latest versions of five mature open-source software suites. This situation can be explained by the pervasive nature of performance, which makes it hard to assess (and its issues hard to identify) because performance is influenced by every aspect of the design, code, and execution of an application.

Recently, cluster computing has gained momentum as a powerful and cost-effective solution paradigm for parallel and distributed processing [3]. Thus, the usage of clusters is becoming pervasive. Nonetheless, this shift to a distributed architecture has also augmented the complexity of the applications, considerably complicating all activities related to performance. Consequently, it is not surprising that achieving good performance under these conditions is commonly a challenging and time-consuming task.

To address this problem, multiple research efforts have been performed to develop techniques which can improve the performance of clusters. This has been done from different perspectives. For example, the authors of [4] presented a high-performance engine to protect the privacy of any sensitive information before releasing it to third-parties (e.g., public cloud services). Likewise, the work in [5] presented a mechanism to provide high reliability and availability for clustered web services. This is achieved by offering a fault-tolerance capability suitable to different classes of transactions. Alternatively, other works have aimed to improve the performance testing and analysis processes. For instance, the work in [6] proposed a technique to generate realistic synthetic data that can be useful to diversify the testing scenarios. Similarly, the work in [7] proposed a methodology to improve the selection of an appropriate analysis tool, best suitable for a particular task, based on a list of usage profiles and comparable criteria.

Moreover, conducting an extensive analysis of the efficiency and effectiveness of all proposed techniques is critical to identify the scenarios (and/or conditions) in which each technique can be useful. This is because a deep understanding of the costs and benefits of the techniques is key to understand their limits and overall practicability [8], [9]. In the particular case of clusters, a common challenge is that its intrinsic complexity makes the creation of an appropriate test environment particularly effort-intensive, and potentially costly, for researchers. Furthermore, unlike other research domains (e.g., Java technologies), the lack of a comprehensive suite of applicable easy-to-use benchmarks indirectly makes the evaluation of cluster-related advancements even harder.

To help tackle this problem, our research work has centred on developing solutions that can help researchers (hereinafter referred as users) to enhance the experimental evaluation of performance engineering techniques that aim to improve the performance of clustered systems. Specifically, this paper presents a tool that enables the easy usage of Java Benchmarks in the construction of cluster-based test environments. Consequently, decreasing the effort and expertise needed to create useful test environments to perform more robust testing. Additionally, we present a practical validation of the proposed tool, consisting of a prototype and the results of two performed case studies. The obtained results illustrate the benefits that can be obtained from using the tool.

II. State-of-the-Art

This section describes the relevant state-of-the-art w.r.t. the background and related work needed to understand this work.

A. Background

With an estimated business impact of a hundred billion dollars every year, Java is a predominant technology at the enterprise level [10]. It has helped organisations to speed up their development processes by leveraging its many object-oriented features. A Java Virtual Machine (JVM) is the run-time environment for Java applications. Internally, it interprets the binary format in which a Java program is compiled. As there are JVMs available for most contemporary operating systems, a Java program is highly portable. Besides, a JVM can be tuned at start-up time through multiple configuration parameters (e.g., to customise its memory management).

Benchmarking has proven to be a useful tool in many research domains. This is because it enables the fair comparison of alternative advancements which aim to solve the same problem by different means. Benchmarks are also helpful to assess the benefits and costs of using a particular advancement [11]. In the case of Java, there have been several efforts aiming to create realistic benchmarks suitable to evaluate the capabilities of JVMs (e.g., to assess their compliance with Java's specifications). Nowadays, two of the Java benchmarks most widely-used are DaCapo and SPECJVM. DaCapo has been sponsored by international companies like IBM, Intel, and Microsoft. Its latest version is 9.12 and it is composed of 14 programs (shown inTable I). They are all open source, real-world programs, and with non-trivial memory loads [12]. Meanwhile, SPECJVM has been developed by the Standard Performance Evaluation Corporation, and companies like HP, IBM and Oracle have also contributed to it. Its latest version is 2008 and it is composed of 10 programs (shown in Table II). They are a mixture of real-life programs and specialised ones developed to cover the core Java functionality [13].

B. Related Work

Multiple works have investigated ways to improve the processes involved in developing software solutions. For example, the work in [14] presents a variant of the popular Model-View-Controller design pattern specially tailored for its usage in the development of groupware. Moreover, the authors of [15] describes a web tool that can help to identify insecure software development aspects in Android applications. Likewise, the authors of [16] proposes a multi-lingual metric tool that can be customised (through user-defined metrics) to suit different scenarios. Meanwhile, the work in [17] presents a detailed comparison between the processes defined by the well-known Project Management Book of Knowledge and the actives defined by the Essence framework, highlighting the key overlaps and differences between both project management standards.

In the particular area of testing, many research works have aimed to enhance its involved processes. For instance, the work in [18] presented a technique to facilitate the process of identifying performance regressions. Besides, the authors of [9] introduced a framework to create realistic testing data, while the work in [19] presented a novel approach to improve the process of monitoring and collecting performance counters. Finally, other research works have aimed to reduce the expertise and effort needed to conduct useful testing. For example, by eliminating the need of manually configuring a diagnosis tool [20], or setting an appropriate test workload [21]. In contrast to these works, which have aimed to improve other aspects of the testing process, our tool has been designed with the aim of facilitating the creation of useful cluster test environments. This is achieved by enabling the re-usage of the most popular Java benchmarks in this scenario.

TABLE I
DaCapo Programs

Name	Description
avrora	It simulates programs running on a grid of microcontrollers.
batik	It processes vector-based images.
eclipse	It executes performance tests in an eclipse instance.
fop	It generates PDF files based on XSL-FO files.
h2	It runs banking transactions against a database application.
jython	It executes a set of python scripts in a Java environment.
luindex	It performs an indexing of documents.
lusearch	It performs a set of keyword searchs over a data corpus.
pmd	It reviews Java classes, looking for bugs.
sunflow	It performs rendering of images.
tomcat	It runs queries against a Tomcat application server.
tradebeans	It executes stock transactions using Java Beans calls.
tradesoap	It executes stock transactions using SOAP calls.
xalan	It transforms XML files into HTML format.

TABLE II
SPECJVM Programs

Name	Description
compiler	It compiles Java source files.
compress	It performs data compression on some test files.
crypto	It and decrypts files with diverse protocols.
derby	It runs queries on a Derby database instance.
MPEGaudio	It decodes a set of audio files.
scimark	It executes a set of floating point operations.
serial	It serialises and deserialises a set of primitives and objects.
startup	It executes each other program a single time.
sunflow	It runs a set of graphics visualisation operations.
XML	It transforms XML documents using style sheets.

III. Proposed Solution

In this section, we describe our proposed solution. We start by providing the context of the tool, then we discuss its internal workings, architecture, and the implemented prototype.

A. Overview

The main goal of our research work was to design a tool that could enable the out-of-the-box re-usage of Java benchmarks in the experimental evaluation of cluster-related advancements. By ensuring an easy and effortless building of useful test environments (based on these benchmarks), we seek to improve the productivity of researchers in this domain. In this context, usefulness means that the test environment is able to offer a broad range of different application behaviours, so that the experimental results obtained from such environment can help to derive, to a fair degree, generalisable conclusions (e.g., the performance gains that the tested technique offers, or its computational costs).

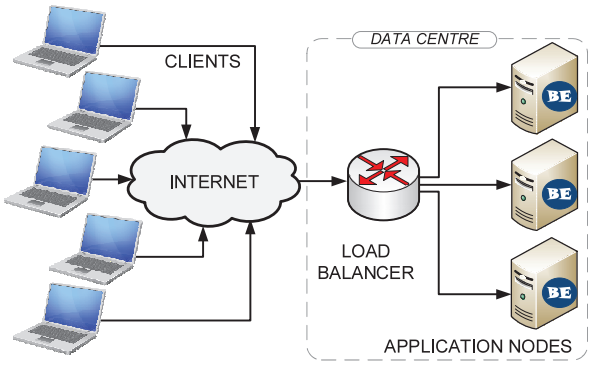

Fig. 1. Contextual Diagram

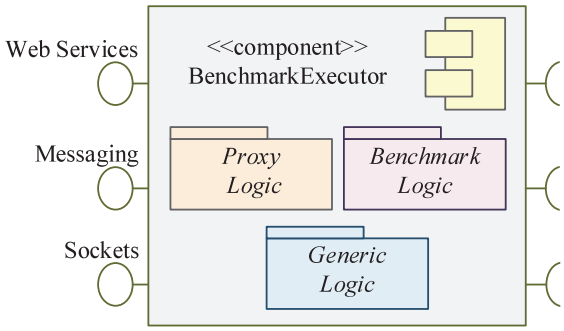

Fig. 2. Component Diagram

Among the range of potential use cases, our work has centred on enhancing the testing of clusters because variants of this distributed architecture are commonly used at enterprise-level. This scenario is exemplified in Fig. 1, which shows how a load balancer is typically responsible for distributing the incoming workload across the available application nodes. Thus, it is not only important to assess the behaviour of the application instances, but also of the load balancer (as it might have a major influence on the overall system performance).

B. Architecture

Our tool is based on the component-based architecture shown in Fig. 2. There, it can be seen how the tool (i.e., a benchmark executor) is composed of three main elements: Firstly, the *benchmark logic*, which contains all the functionality required to successfully execute the set of supported Java benchmarks. Secondly, the *proxy logic*, which contains the functionality that allows the tool to interface with different types of clients. Thirdly, the *generic logic*, which contains all functionality providing miscellaneous services which are independent of the supported set of proxies and benchmarks.

This architecture was designed with the aim of minimising the code changes required to extend the tool (e.g., to support other benchmarks, or types of clients). Following the same line of thinking, the components exclusively interface with each other through interfaces. This is exemplified in Fig. 3, which presents a simplified high-level class hierarchy of the tool. There, it can be seen how the tool internally follows an object-oriented design where the involved entities are modelled as a set of inter-related objects and interact with each other (through interfaces) within the tool. For instance, the figure shows how the benchmark component contains a main interface *IBenchmark* to expose the set of supported actions, as well as an abstract class which contains all the common functionality (at benchmark level). Then, the hierarchy of classes can be extended to support specific types of benchmarks (e.g., DaCapo or SPECJVM). A similar strategy is followed for the proxy logic, as different strategies to interface with the tool can be supported (e.g., a web interface or a messaging service). This design decision was taken to make the architecture highly extensible and easily maintainable.

Finally, the different components communicate through commands, following the well-known *Command* design pattern [22]. For instance, the controller class invokes a command to instantiate an access proxy, while the corresponding proxy class implements the logic in charge of actually initialising the proxy logic (e.g., creating an embedded instance of an application server, in the case of a web-based proxy). The internals of these components are explained in the following sections:

1) Generic Logic: In Fig. 3, it can be seen how the Controller acts as the MainClient, handling all the tasks associated with the initialisation of the tool. For instance, setting the configuration parameters provided by the user. This involves a set of access proxies, benchmarks and their respective configurations (e.g., a port and context in the case of a web proxy, or the path where the benchmark's executable is located). The Controller is also responsible for managing the life-cycle of the selected set of access proxies (i.e., initialisation and destroy methods), while the corresponding proxies are responsible for implementing the actual initialisation and cleaning processes. For instance, a proxy might require creating an instance of an embedded messaging service or a web application server (in the case of a messaging and web proxies, respectively) as part of its initialisation. Inversely, the proxies would need to release such resources during their destroying cycles (e.g., stopping the web server instance in the case of a web proxy).

It is also worth mentioning that the generic logic involves a set of utility services, which have been identified as potential useful logic for the supported set of proxies and benchmarks. For instance, a standardised logging mechanism, thread-based classes to consume and parse the verbose produced by the benchmarks (in their standard error and output streams), logic to safely execute and monitor the execution of the benchmarks in an independent process (also supporting diverse operating systems, such as Windows and Linux), or miscellaneous logic to execute the benchmarks through Java reflection.

Finally, the generic logic is also responsible for the instantiation of the different benchmark objects, which are later used by the access proxies to execute the benchmarks' programs. Internally, the required benchmark class (supporting a particular Java benchmark, and possible a specific operating system) is instantiated by a developed benchmark factory [22], which

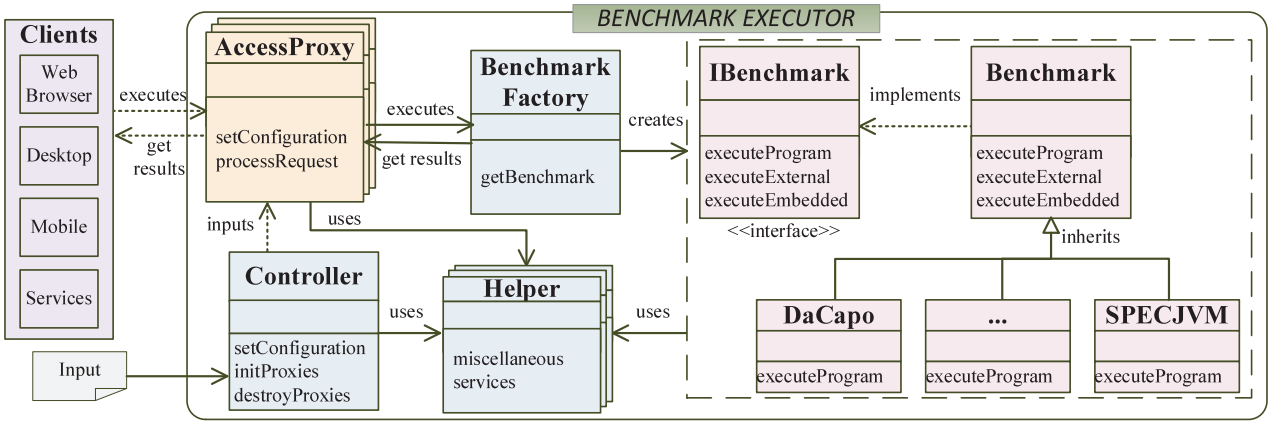

Fig. 3. Simplified Class Diagram

is responsible for handling the selection of the appropriate class to create the required object type per benchmark. This design decision was taken with the aim of isolating the rest of the tool from the complexities of initialising the correct class across the hierarchy of benchmark sub-classes (e.g., calling the right constructor).

2) Proxy Logic: Understanding that each type of proxy might require very different supporting logic to work properly, the architecture of our tool only enforces a set of life-cycle methods that homogenise the management of the proxies. They are defined on an interface that every proxy must implement and involved three methods: init, service, and destroy. The init method is responsible for initialising the proxy; the service method is responsible for processing the client requests and returns the success (or failure) of the operation, including the results of the same; while the destroy method is responsible for performing any cleaning that is needed to release whatever resources were obtained during the initialisation step.

This design strategy helps the tool to support a diverse range of access proxies, which might be applicable to different use cases and clients (e.g., web applications, web services, messaging servers, sockets, etc.). This strategy also allows supporting multiple similar proxies. For instance, following our web example, for some test scenarios, it might be better to have an embedded web application server (to simplify the installation of the tool in the test environment). Hence, an embedded web application server (such as Jetty [23]) can be easily integrated into the tool. Alternatively, a user might prefer to install the tool on top of an existing web application server (e.g., Tomcat [24]) because her testing methodology requires to validate that her research works under a real-life application server (like Tomcat). Therefore, the tool can implement a deployable servlet/JSP-based web application, which would act as an alternate web proxy.

3) Benchmark Logic: The core element of this component is the hierarchy of supported benchmarks. It starts with a main interface (to expose the supported commands), then an abstract class which offers the common logic across the benchmarks. In this part, it was important to identify shared characteristics among the investigated benchmarks in order to model them into a reusable hierarchy of classes. This is important not only to reduce the code repetition, but also to efficiently support other Java benchmarks (other than the analysed ones). For instance, the tool currently supports two alternate execution modes: *embedded* and *external*. In the embedded mode, the benchmark program executes within the same JVM of the tool. This can be useful to make the overall benchmark(s) behave as a more complex application. For instance, to stress more a memory-related advancement. To support this mode, it is important to execute the logic, while avoiding some behaviours that might affect the health of the tool. For example, to disable the calls that the benchmarks might do to the *System.exit()* method (e.g., by creating and setting a NoExitSecurityManager, which avoids the direct triggering of the exit and converts it into an exception, which can then be captured and reported as part of the benign errors of the benchmark program). Similarly, it is important to set any other dependencies that are needed (e.g., JVM properties). Also, the abstract benchmark logic needs to be thread-based, so that multiple instances of the benchmark programs can be run in parallel (to emulate a distributed application, which is a typical scenario in clustered environments). On the contrary, in the external mode, the benchmark program is executed in its own JVM (i.e., independent from the tool). This might be useful in cases where a user want to mimic an application distributed across multiple JVMs, or when a benchmark program has some dependencies that might not be modelled within the tool due to their technical characteristics (e.g., it might not be possible to execute a benchmark program involving an embedded web server - like Tomcat, from the DaCapo benchmark - from within a JSP-based web proxy, as that would involve running a Tomcat within another Tomcat).

The abstract benchmark class also contains a core process that executes the supported benchmarks. The first step involves to set-up the experimental environment that the benchmark needs to be executed successfully (i.e., any required dependencies that it might have). For instance, the benchmark might require the presence of a temporal directory (or a copy of the

Java ARchive that contains the benchmark, in the case of the external execution mode). Next, the execution of the chosen benchmark program begins. This step involves constructing the exact command required to run the benchmark (as each supported benchmark might have a diverse syntax as well as a set of supported options available). This logic is also responsible for handling the program's execution (e.g., the parsing/processing of its standard error and output streams). This level of abstraction ensures that the tool can be extensible for other Java benchmarks, as they usually share the same type of starting steps and processing functionality (e.g., they always have as entry point a pre-configured class and method). Once the benchmark program has started, the standard output and error streams of the benchmark program are parsed and processed (so that this information can be logged, if needed). Once the execution has finished, the benchmark logic reports the success (or failure) of the execution to the AccessProxy that called it. Finally, all the dependencies that have been previously set are rolled back (e.g., anything that has been copied into a temporal directory is deleted). The goal is to reduce the resource footprint of the tool by releasing any resources (e.g., hard disk) that are temporarily required to successfully execute a benchmark program. Finally, all errors and exceptions that might occur are internally handled.

C. Prototype

From a technical perspective, our prototype was developed in Java 7. This has been done with the aim of making our solution highly portable (hence maximising its portability and overall adoption), as there are JVMs available for most of the contemporary operating systems [25]. For this initial prototype, we concentrated on implementing a web proxy. This design decision was taken because the web application servers are a traditional Java business niche [8]. Furthermore, we implemented two web proxies: One proxy was developed by embedding a Jetty web servlet container [23], which is a popular open source solution used for enabling the machine to machine communications. This proxy exemplifies how our tool can simplify the configuration of a test environment, as it is a self-contain solution (i.e., without external dependencies to run successfully) that only needs to be installed in the application server nodes. The other proxy was developed by creating a set of JavaServer Pages (within a web application) that offer the same functionality (i.e., they allow the different supported benchmarks, through their respective set of programs, to be executed through HTTP requests). This alternate proxy exemplifies how our tool can be easily integrated with existing testing infrastructure (as this proxy enables the prototype to be executed within any Java web application server currently available in the industry).

Regarding its configuration, the tool relies on a set of Extensible Markup Language (XML) configuration files. XML was chosen for configuration purposes because it is a widely used and standard format that can also be easily understood by the user (e.g., it allows to add comments that can complement the description of the options). Fig. 4 presents an example of

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
    version="2.4">
  <welcome-file-list>
    <welcome-file>html/index.html</welcome-file>
  </welcome-file-list>

  <servlet>
    <servlet-name>dacapo</servlet-name>
    <servlet-class>org.ucd.pel.webbenchmark.DaCapoBenchmark_External</servlet-class>
    <init-param>
      <param-name>jvmPath</param-name>
      <param-value>/home/op1/ATP3_env/IBM_jre/bin</param-value>
    </init-param>
    <init-param>
        <param-name>jvmArgs</param-name>
        <param-value> </param-value>
    </init-param>
    <init-param>
      <param-name>benchmarkPath</param-name>
      <param-value>/home/op1/ATP3_env/web-benchmarks/external/dacapo</param-value>
    </init-param>
    <init-param>
      <param-name>benchmarkJarfile</param-name>
      <param-value>dacapo-9.12-bach.jar</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>dacapo</servlet-name>
    <url-pattern>/runDacapo.jsp</url-pattern>
  </servlet-mapping>
```

Fig. 4. Example of Configuration File

the tool's configurations (i.e., the configuration file of one of the web proxies, for the DaCapo benchmark).

In terms of Java benchmarks, we have initially concentrated on supporting the DaCapo and SPECJVM benchmarks. This is because, as discussed in Section II-A, they are two of the Java benchmarks most widely-used in the literature, as well as they offer a broad range of diverse programs that can be useful to diversify the testing of cluster-oriented advancements. Additionally, these benchmarks are composed of real life programs (unlike other Java benchmarks which have been synthetically generated) and which do not have trivial memory loads [26] (an important characteristic which exemplifies their potential usefulness in testing, as memory is an important computational cost aspect to consider when assessing the feasibility of any advancement for real-word usage).

Our analysis of the chosen benchmarks allowed us to develop the corresponding benchmark classes to support them. Additionally, we have configured their main identified attributes within the system. Tables III and IV summarise them. It is important to mention that their attributes can be classified into two main types: Those that are common across both benchmarks (suggesting that they are generic and potentially applicable to other benchmarks), and those that are specific to the benchmark. This is an important finding, which has led us to believe that the hierarchy of benchmarks might be extended with intermediate classes that encapsulate the similarities in behaviours and/or functionalities that some subsets of benchmarks might experience (like DaCapo and SPECJVM do). This idea, that we plan to explore in the near future, can be useful to make our tool more robust by making it easier to integrate other benchmarks to the supported ones.

Regarding the specific configurations, they are closely related to the specific usage of the programs within the benchmark. For instance, in the case of SPECJVM (whose programs are mainly based on timed executions), the identified specific attributes are: the time to perform a warm-up of the program, the number of iterations that will be carried out, and the time that each iteration will last. In contrast, as DaCapo is mainly iteration-driven, the main elements to control its execution are: the number of iterations, and the size of the test workload that

Fig. 5. Example of Tool Execution

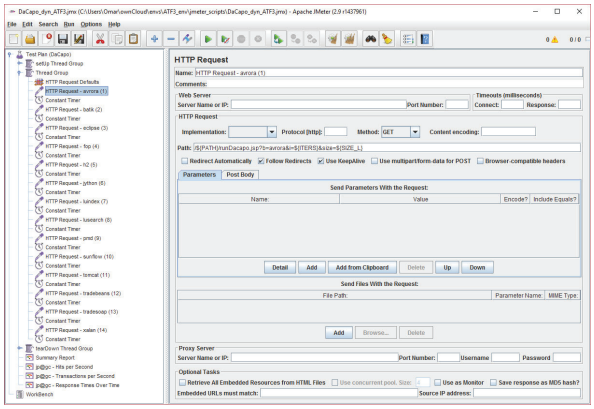

Fig. 6. Example of JMeter Test Script

it will use. Here, it is worth mentioning that each program has its set of supported sizes (ranging from a small to a huge sizes). As the ranges of supported test workloads vary across the 14 supported programs, we considered more appropriate to use the "default" test workload as the default value of its corresponding attribute (as this is one size common across all DaCapo programs).

Finally, it is worth mentioning that some specific attributes are similar in spirit (among the benchmarks), as they aim to address an equivalent need (e.g., the internal validation attribute which aims to assess, before the actual execution of the program, if its dependencies have been fulfilled). However, from the time being, these attributes have been classified as specific until we analyse other benchmarks (as we do not preliminarily considered that these attributes will be very commonly found in many Java benchmarks).

To complement our discussion of the developed prototype, Figs. 5 and 6 presents an example of the outputs of the tool, as well as an example of the test scripts that can be generated to test its functionality (i.e., the set of supported benchmarks and their programs). Putting aside the fact that the look-and-feel of the tool can be improved (which is one of our next-in-line goals), it can be noticed in Fig. 5 how the benchmark executions can be tailored to the set of supported attributes (as previously explained). In this example, we used a GET HTTP request because that makes the example more visual. However, the tool also supports POST HTTP requests. Moreover, the result of the execution (i.e., either success or failure) is presented, as well as the configuration that was used. This is a valuable level of debugging information (e.g., in cases where default values are used for some of the configuration parameters). Meanwhile, Fig. 6 exemplifies how the tool can be successfully applied to the testing of cluster-related advancements. In the figure, it can be seen a test script created in JMeter [27] (a popular performance testing tool). In this example, the whole set of DaCapo programs was used together to simulate an application composed of 14 different functional operations. Alternatively, a user might prefer to create individual test scripts in order to assess each program behaviour individually. Both of these testing strategies are exemplified in our experimental evaluation.

TABLE III
DACAPO CAPTURED ATTRIBUTES

Type	Name	Default Value
Generic	Name	DaCapo
Generic	Path	TOOL_BASE_DIR/benchmark/dacapo
Generic	Jarfile	dacapo-9.12-bach.jar
Generic	Directory Dependencies	None
Generic	JVM Properties	None
Generic	Supported Programs	All programs listed in Table I
Generic	Default Program	avrora
Generic	Main Class	org.dacapo.harness.TestHarness
Specific	internal validation	false
Specific	iteration	1
Specific	workload size	default

TABLE IV
SPECJVM CAPTURED ATTRIBUTES

Type	Name	Default Value
Generic	Name	SpecJvm
Generic	Path	TOOL_BASE_DIR/benchmark/specjvm
Generic	Jarfile	SPECjvm2008.jar
Generic	Directory Dependencies	lib, resources, redistributable_sources
Generic	JVM Properties	specjvm.home.dir
Generic	Supported Programs	All programs listed in Table II
Generic	Default Program	startup
Generic	Main Class	spec.harness.Launch
Specific	internal validation	false
Specific	warmup	1 minute
Specific	iteration	1
Specific	iteration time	1 minute

IV. EXPERIMENTAL EVALUATION

Here, we present the setup of the performed experiments, as well as the highlights of the experimental results obtained.

A. Experimental Setup

In this section, we describe the methodology that was used for our experimental validation, as well as the different elements that defined the set of experimental configurations used, including the evaluation criteria and the test environment.

1) Methodology: The performed experiments aimed to illustrate the benefits of using our developed tool in the testing of cluster-related advancements. To achieve this, we conducted two series of experiments. In the first series, we used the

tool as part of the evaluation of a load balancing strategy (describe next). In the second series, we used the tool in the evaluation of a performance testing framework (describe next). This dual methodology design was done with the aim of showing how the tool can be suitable for strengthening the experimental evaluation of different cluster-related solutions. Finally, it is important to mention that the aim is not to exhaustively evaluate the advancements, but to demonstrate how our tool can be useful to strengthen the validation of such types of advancements.

2) Load Balancing Strategy: To illustrate the benefits of the proposed tool, we firstly used it to experimentally evaluate the behaviour of TRINI [8]. TRINI is an adaptive load balancer strategy which aims to improve the performance of a clustered system by avoiding the impacts in the cluster's performance caused by the occurrence of the Major Garbage Collection (MaGC) events in the individual nodes (which are known to be a major cause of performance degradation in Java systems [28]).

In terms of experimental configurations, two different types of runs were performed. The first type used the original version of two popular load balancing algorithms: random (RAN), and round robin (RR). They were chosen not only because they are widely used in the industry, but also because TRINI supports their GC-aware counterparts (i.e., GC-RAN and GC-RR). Moreover, the second type of run precisely used these GC-aware algorithms. For these GC-aware algorithms, a value of 100 ms was used as *sampling interval* (SI). The SI is a configuration parameter required by TRINI, which defines how frequently it retrieves GC and memory samples from the clustered system (data which is internally used to predict when a MaGC event will occur in an individual node, information which is then used to decide if a node needs to be skipped, w.r.t. the incoming load, due to the closeness of a MaGC event). Finally, each test run lasted 60 minutes and used 100 concurrent virtual users.

3) Performance Testing Framework: To demonstrate how the proposed tool can be easily adjusted to different use cases, we also used it to experimentally assess PHOEBE [20], which is a policy-based adaptive framework that automates the configuration and usage of a diagnosis tool (e.g., WAIT [25]) in the performance testing of clustered applications. The aim is to enhance a tester's productivity by decreasing the effort and expertise needed to effectively use a diagnosis tool. For instance, PHOEBE is able to adapt during the performance test run the sampling interval (which is used to gather information from the system-under-test) in order to keep the introduced overhead caused by the gathering process under control (so that it does not compromise the results of the test).

In terms of experimental configurations, two different types of runs were performed. The first type used the diagnosis tool manually, involving the usage of a range of static (i.e., pre-configured) sampling interval (SI) values (i.e., 0.125, 0.25, 0.5, 1, 2, 4, 8 and 16 minutes) in order to have a baseline to which compare PHOEBE's results against. The second type of run used PHOEBE. It required the definition of some input configurations required by PHOEBE: A 20% response time threshold was defined (configuration parameter which defines the maximum amount of tolerable introduced overhead). Besides, a minimum SI and ΔSI were set to 30 seconds (configuration parameters which define the sampling interval space that is explored by the automated data gathering process). Finally, each test run lasted 24 hours in order to mimic realistic (i.e., industry-like) testing conditions.

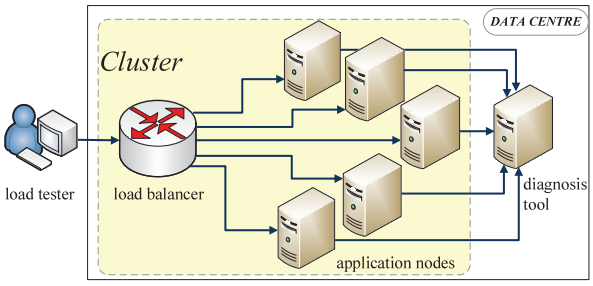

Fig. 7. Test Environment

4) Environment: All the performed experiments were done in an isolated test environment, so that the overall workload was controlled. This environment was composed of eight virtual machines (VM): A cluster of five application nodes with one load balancer, one diagnosis tool server, and one load tester node (as shown in Fig. 7). All the VMs had the following characteristics: 4 virtual CPUs @ 2.20GHz, 3GB of RAM, and 50GB of HD; running Linux Ubuntu 12.04L 64-bits, and a JVM 7 with a 1,600MB heap (i.e., memory). Each JVM was configured to initialise its heap to its maximum size (to keep it constant during the experiments), and the calls to programmatically request a Garbage Collection (i.e., the automatic memory management process) were disabled. Moreover, the VMs were located on a Dell PowerEdge T420 server equipped with 2 Intel Xeon CPUs at 2.20Ghz (12 cores/24 threads), running Linux Ubuntu 12.04L 64-bit, 96 GB of RAM, 2TB of HD, and using KVM [29] for virtualisation.

The load balancer node ran an Apache Camel [30] server, which is a well-known lightweight Java-based load balancer. The load tester node ran an Apache JMeter 2.9 [27] instance, which is a leading open source tool commonly used for application load performance testing. The application nodes ran an Apache Tomcat 6.0.35 [24], which is a popular open source Java web application server. The diagnosis tool used was WAIT, which has proven to be useful to identify performance issues in Java applications [31]. It is based on non-intrusive sampling mechanisms available at Operating System level (e.g., the top command in Unix) and the JVM, in the form of Javacores (which are snapshots of the JVM state, offering information such as locks, threads, and memory).

5) Evaluation Criteria: In terms of performance, our main metrics were the throughput per second (tps) and response time (ms). They were collected with JMeter. Regarding the response time, lower values are better; while higher values are better for throughput. In terms of resource utilisation, our main metrics were CPU (in %) and memory (in MB) utilisations. In

both cases, lower values are better. They were collected using the top command. In terms of testing productivity, our main metric was the number of bugs found. Here, higher values are also better. The bugs were obtained from the reports generated by the chosen diagnosis tool (i.e., WAIT).

B. Experimental Results

1) TRINI's results: In terms of the effort required to use the tool in our initial use case (i.e., performance optimisation in real-time), it was minimal: We only needed to install the tool in the application nodes. It involved placing the tool inside Tomcat's application directory (as we chose to use the web interface), as well as configuring the location of the JVM which would be used in the experiments (in our XML configuration files). As TRINI's implementation currently uses Java Management Extension (JMX) to interact with the monitored JVM, we used Oracle HotSpot JVM (as it offers more robust JMX support, compared to other JVM versions). JMX is a standard component of Java Standard Edition which offers a set of standard management components (i.e., MBeans) which allow the easy extraction of monitoring data (e.g., GC, memory, threads, etc).

This experiment also required the creation of a set of JMeter test scripts. Specifically, a JMeter test script was created for each one of the 23 programs that compose the DaCapo and SPECJVM benchmarks. This design decision was taken to diversify as much as possible the GC/memory application behaviours to be evaluated (as each program has its own GC/memory behaviour). Each script had some controlled diversity with respect to the test workload. The specific strategy followed was different for DaCapo and SPECJVM due to the differences in their behaviours (i.e., DaCapo is based on iterations, while SPECJVM is based on time). In the case of the DaCapo programs, the script varied the workload size between the different program calls (by iterating among the existing pre-defined workload sizes of DaCapo). In the case of the SPECJVM programs, the controlled diversity involved varying the warm-up and execution iteration times between program calls. It was done iteratively in the range between 30 to 90 seconds (in increments of 30 seconds).

Typically, an important evaluation perspective is to assess the generality of any obtained findings. For instance, any performance benefits (e.g., an increment in the throughput, or a decrement in the response time). Likewise, it is important to understand the costs of a proposed solution (e.g., introduced overhead or the amount of required computational resources). These two scenarios are exemplified in Figs. 8 and 9. Fig. 8 shows the performance improvements obtained by TRINI. A performance improvement for a specific metric (e.g., throughput) is defined as the difference between an experimental configuration using a GC-aware load balancing algorithm (e.g., GC-RAN) and its original counterpart (e.g., RAN). Regarding response time, an improvement implies a negative difference (as lower response time is better) in the range between 0% and 100%. Regarding throughput, an improvement implies a positive difference (as higher throughput is better) and has a value equal or greater than 0%. The overall results showed that TRINI worked well, as significant performance improvements were observed (in average) for both evaluated GC algorithms. More importantly, similar results were obtained, proving to a certain degree the generality of TRINI's improvements. A point worth noticing is the relatively high standard deviations (e.g., it ranged between 28% and 31% for the response time). They are the result of considerably diversifying the range of memory behaviours that were tested (i.e., the 23 programs that belong to the DaCapo and SPECJVM benchmarks). Such diversify would have been harder to achieve without our tool.

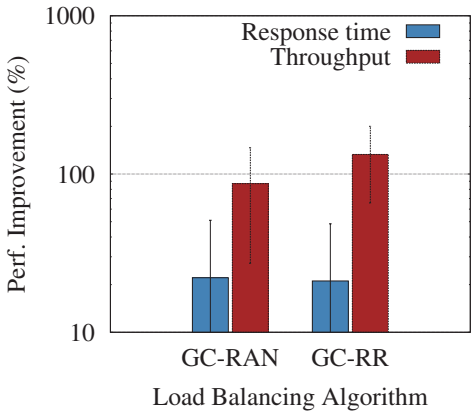

Fig. 8. Performance Improvements per Load Balancer

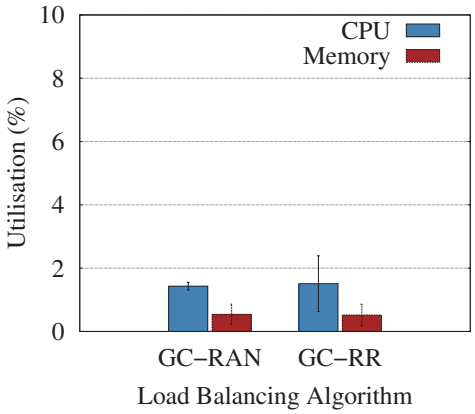

Fig. 9. Performance Improvements per Load Balancer

Similarly, Fig. 9 presents some interesting findings, this time in terms of the overhead introduced to the cluster: Unlike the performance gains discussed previously, which experienced a relatively high standard deviation, the costs of TRINI (i.e., CPU and memory utilisations) had a minimal impact in the clustered application (e.g., it introduced an overhead of approximately 1% in average CPU utilisation) regardless of the tested application behaviour (as reflected in a low standard deviation -i.e., less than 1%-). This is an important finding, as the amount of required computational resources is usually an important indicator of the practicability of the technique for a real-world usage. This low overhead is the result of the

design of the solution, where most of the work is done at the load balancer node (typically not a major concern, in terms of resource utilisation, as that is a node exclusively dedicated to load balancing). For that reason, the only source of overhead in the application nodes is the data gathering process of TRINI (i.e., through JMX), which periodically collects memory/GC samples from the monitored application nodes.

In summary, these results demonstrated how our tool was useful to diversify the tested application's behaviours. Consequently, helping to have more confidence regarding the generality of the obtained results.

2) PHOEBE's results: In terms of the changes required to adjust our tool to this alternative use case, they were minimal, only requiring to modify the configured JVM in our XML configuration files. This is because the chosen diagnosis tool (i.e., WAIT) relies on Javacores (as explained in Section IV-A) which are better supported by the IBM JVM (in comparison with Oracle's version). Therefore, it was more appropriate to use an IBM JVM in this scenario. This also exemplifies how our tool can be reused across evaluation strategies. Another difference of this evaluation strategy (compared to TRINI's) was the way that the tool was used: As the aim was to mimic a complex application (so that it has multiples potential performance bottlenecks), all the different programs were executed in a single JMeter test script. Here, all the programs were iteratively executed during the total performance test run, using the highest available test workload (where applicable).

An example of the obtained results is depicted in Fig. 10 which compares the results obtained by PHOEBE against the set of eight rival static sampling intervals. There, it can be noticed the trade-off that is typically experienced when using a diagnosis tool: On one hand, if one samples very frequently, one will be able to collect more data from the system-under-test. Consequently, as one is able to better feed the diagnosis tool, the bug accuracy (i.e., the identification of performance bugs) can be drastically improved. On the other hand, if one samples too frequently, one risks introducing a high overhead into the system-under-test. If it ever happens, the test results might get compromised. In the figure, this is illustrated by the test run using the smallest SI (i.e., 0.125 minutes): It obtained the highest bug accuracy, but at the expense of introducing a considerably high performance impact. On the contrary, PHOEBE was able to achieve a good level of bug accuracy, within the tolerable level of overhead (i.e., the blue line shown in the figure).

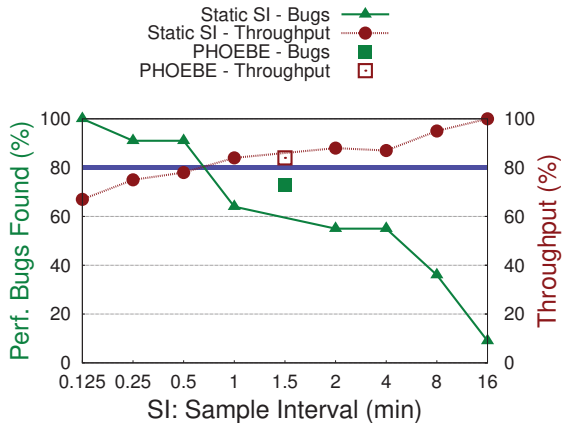

Fig. 10. Example of PHOEBE's results: Bug Accuracy vs. Throughput

To complement the results of this analysis, Fig. 11 shows an example of one of the obtained WAIT reports (using a static SI of 30 seconds). It can be observed how a rich WAIT report was successfully generated: Its top area summarises the usage of resources (i.e., CPU or memory) and the types of threads collected and analysed. Meanwhile, the bottom section shows all the performance issues that have been identified, ranked by frequency and impact. Each problem category is indicated with a different colour. For example, in Fig. 11 the top issue appeared in 68% of the samples and affected 18 threads on average. Additionally, it was a multi-factorial problem, mainly caused by a CPU-intensive method (as the blue colour is used to identify CPU problems), and in less degree by an inappropriate usage of locks (as the brown colour identifies lock-related issues).

In conclusion, these results proved how our tool can be easily adjusted to support more than one use case. For instance, in this case, we used it to demonstrate the improvements in bug accuracy (i.e., identified performance issues). This was done by combining the full set of 23 application behaviours, so that it behaves as a complex web application (which is reasonable, as we have previously shown - as part of TRINI's results - how the 23 programs belonging to the DaCapo and SPECJVM benchmarks are considerably diverse in their memory/GC behaviours).

V. Conclusions and Future Work

Conducting an extensive analysis on the efficiency and effectiveness of any proposed technique is key to understand its limits and under which conditions it might be appropriate to use. To help tackle this problem within the clustering domain, this paper presented a tool that has been designed to enable researchers to effortlessly leverage well-known Java benchmarks in the testing of cluster-related advancements. Additionally, we experimentally evaluated the tool through two case studies where it was successfully applied to the testing of a load balancing strategy and a performance testing framework. These case studies demonstrated how the tool works well, as it was possible to easily use two of the most widely-used Java benchmarks to strengthen the testing of those cluster advancements. As future work, we plan to increase the set of Java benchmarks that are supported, as well as to extend the tool to work with other technologies (e.g., .NET). Also, we plan to broaden our experimental validation. For instance, by diversifying the sizes of the clusters and assessing the suitability of the tool to other distributed architectures (e.g., grids). Finally, we plan to improve the quality of our source code in order to release our work as an open-source project.

Fig. 11. Example of WAIT Report

ACKNOWLEDGMENTS

This work was supported, in part, by Science Foundation Ireland grant 13/RC/2094.

REFERENCES

[1] A. O. Portillo-Dominguez, J. Murphy, and P. O'Sullivan, "Leverage of extended information to enhance the performance of jee systems," in *ITT*, 2012, pp. 137–138.

[2] E. G. Jin, L. Song, "Understanding and detecting real-world performance bugs," *ACM SIGPLAN conference on Programming Language Design and Implementation*, 2012.

[3] W. Y. Lee, S. J. Hong, J. Kim, and S. Lee, "Dynamic load balancing for switch-based networks," *Journal of Parallel and Distributed Computing*, vol. 63, no. 3, pp. 286–298, 2003.

[4] V. Ayala-Rivera, D. Nowak, and P. McDonagh, "Protecting organizational data confidentiality in the cloud using a high-performance anoymization engine," in *ITT*, 2013.

[5] N. Aghdaie and Y. Tamir, "Coral: A transparent fault-tolerant web service," *Journal of Systems and Software*, vol. 82, no. 1, pp. 131–143, 2009.

[6] V. Ayala-Rivera, P. McDonagh, T. Cerqueus, and L. Murphy, "Synthetic Data Generation using Benerator Tool," *CoRR*, vol. abs/1311.3, pp. 1–12, 2013.

[7] M. Delahaye and L. Bousquet, "Selecting a software engineering tool: lessons learnt from mutation analysis," *Softw. Pract. Exp.*, vol. 45, no. 7, pp. 875–891, 2015.

[8] A. O. Portillo-Dominguez, P. Perry, D. Magoni, M. Wang, and J. Murphy, "TRINI: an adaptive load balancing strategy based on garbage collection for clustered java systems," *Softw. Pract. Exp.*, 2016.

[9] V. Ayala-Rivera, A. O. Portillo-Dominguez, L. Murphy, and C. Thorpe, "COCOA: A synthetic data generator for testing anonymization techniques," *PSD*, 2016.

[10] International Data Corporation (IDC), "Java : Two and a Half Years After the Acquisition," Tech. Rep. August, 2012.

[11] H. Kasture and D. Sanchez, "Tailbench: a benchmark suite and evaluation methodology for latency-critical applications," in *IISWC*, 2016.

[12] "The DaCapo Benchmark Suite," Last accessed: 2017-07-15. [Online]. Available: http://dacapobench.org/

[13] "SPECjvm 2008," Last accessed: 2017-07-15. [Online]. Available: http://www.spec.org/jvm2008/

[14] M. Anzures-García, L. A. Sánchez-Gálvez, M. J. Hornos, and P. Paderewski-Rodríguez, "MVC design pattern based-development of groupware," in *CONISOFT*, 2016, pp. 71–80.

[15] A. Rodríguez-Mota, P. Escamilla-Ambrosio, E. Aguirre-Anaya, R. Acosta-Bermejo, and L. Villa-Vargas, "Improving android mobile application development by dissecting malware analysis data," in *CONISOFT*, 2016, pp. 81–86.

[16] A. S. Núñez-Varela, H. G. Pérez-González, F. E. Martínez-Pérez, and J. Cuevas-Tello, "Building a user oriented application for generic source code metrics extraction from a metrics framework," in *CONISOFT*, 2016, pp. 27–32.

[17] M. J. Simonette, M. E. Magalhães, and E. Spina, "Pmbok five process groups and essence standard: Perfect partners?" in *CONISOFT*, 2016, pp. 53–58.

[18] W. Shang, A. E. Hassan, M. Nasser, and P. Flora, "Automated detection of performance regressions using regression models on clustered performance counters," *ICPE*, 2015.

[19] A. Van Hoorn, J. Waller, and W. Hasselbring, "Kieker: A framework for application performance monitoring and dynamic software analysis," in *ICPE*, 2012, pp. 247–248.

[20] 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," *Softw. Pract. Exp.*, 2017.

[21] M. Kaczmarski, P. Perry, J. Murphy, and A. O. Portillo-Dominguez, "In-test adaptation of workload in enterprise application performance testing," *ICPE Companion*, 2017.

[22] M. Fowler, *Patterns of enterprise application architecture*, 2002.

[23] "Eclipse Jetty," Last accessed: 2017-07-15. [Online]. Available: http://www.eclipse.org/jetty

[24] "Apache Tomcat," Last accessed: 2017-07-15. [Online]. Available: http://tomcat.apache.org/

[25] E. Altman, M. Arnold, S. Fink, and N. Mitchell, "Performance analysis of idle programs," *ACM SIGPLAN Notices*, vol. 45, no. 10, Oct. 2010.

[26] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer *et al.*, "The DaCapo benchmarks: Java benchmarking development and analysis," in *ACM Sigplan Notices*, vol. 41, no. 10, 2006, pp. 169–190.

[27] "Apache JMeter," Last accessed: 2017-07-15. [Online]. Available: http://jmeter.apache.org/

[28] A. O. Portillo-Dominguez, M. Wang, J. Murphy, and D. Magoni, "Adaptive GC-aware load balancing strategy for high-assurance java distributed systems," in *HASE*, 2015.

[29] "KVM Hypervisor," Last accessed: 2017-07-15. [Online]. Available: http://www.linux-kvm.org/

[30] "Apache Camel," Last accessed: 2017-07-15. [Online]. Available: http://camel.apache.org/

[31] H. Wu, A. N. Tantawi, and T. Yu, "A self-optimizing workload management solution for cloud applications," in *International Conference on Web Services*, 2013, pp. 483–490.