VCR: Virtual Capture and Replay for Performance Testing

Conference Paper · September 2008

DOI: 10.1109/ASE.2008.58 · Source: DBLP

Antonia Bertolino
Italian National Research Council
238 PUBLICATIONS **3,578** CITATIONS
SEE PROFILE

Antonino Sabetta
SAP Research
64 PUBLICATIONS **679** CITATIONS
SEE PROFILE

VCR: Virtual Capture and Replay for Performance Testing

Antonia Bertolino, Guglielmo De Angelis, Antonino Sabetta
Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo"
Consiglio Nazionale delle Ricerche
Pisa, Italy
{antonia.bertolino, guglielmo.deangelis, antonino.sabetta}@isti.cnr.it

***Abstract*—This paper proposes a novel approach to performance testing, called Virtual Capture and Replay (VCR), that couples capture and replay techniques with the checkpointing capabilities provided by the latest virtualization technologies. VCR enables software performance testers to automatically take a snapshot of a running system when certain critical conditions are verified, and to later replay the scenario that led to those conditions. Several in-depth analyses can be separately carried out in the laboratory just by rewinding the captured scenario and replaying it using different probes and analysis tools.**

I. Introduction

Modern pervasive software-intensive systems need effective and efficient methods for guaranteeing and assessing their functional and non-functional properties. Our research targets the testing of non-functional properties of networked, service-oriented applications. We propose an approach combining on-line monitoring of real executions to timely detect performance problems, and off-line testing in a controlled environment to carry out the appropriate root-cause and what-if analyses.

The approach presupposes the following capabilities: 1) to keep track of the execution environment at the instant in which a problem arises, so that we can collect all relevant system state and data, and later reproduce the conditions under which the malfunctions occurred; 2) to capture all interactions, user inputs and exchanged messages that led the system to the performance situation under analysis.

We fulfill both needs by adopting existing methodologies: concerning 1), the approach we take is the well-known technique of system checkpointing, coupled with virtualization; concerning 2), we use capture & replay techniques. Thus, the approach we present is called Virtual Capture & Replay (VCR[1]).

The approach assumes virtualization technology, but this assumption does not hinder applicability. In fact, virtualization is today experiencing a "renaissance" [1], and all major companies seem to migrate to this at the same time "break-through and old" technology. Thanks to the efficient checkpointing capabilities provided by the latest virtualization technologies, rather than devoting effort and attention to carving an execution context (as e.g. in [2], [3]), it is feasible to take a snapshot of a whole machine and replay it altogether with negligible resources overhead.

Capture & replay is a long established technique, which essentially consists of recording a sequence of a system's executions and then reproducing them, mostly for test generation or for debugging purposes [4]. As said, we adopt here capture & replay for a novel objective, i.e., performance analysis.

To make the test more efficient, during the replay we use dynamic inspection analysis, which allows testers to focus on relevant parts of the captured state.

Summarizing, the contribution of this work stays in using in a novel way a combination of quite established technologies for an unusual objective, which is *performance testing*. More precisely, this paper's contributions include:

- on a broader view, the definition of a novel approach to QoS assessment that is obtained by combining virtualization, system checkpointing, capture & replay at system level, and dynamic instrumentation;
- in particular, an opportunistic realization of the above approach, based on continuous monitoring of selected QoS parameters and on checkpointing only at the triggering of specified conditions.

The paper is structured as follows: Sec. II briefly introduces background notions on the adopted technologies. Sec. III presents the VCR approach and its general components. Then, Sec. IV provides a short summary of an initial evaluation, while Sec. V proposes potential applications of VCR. Finally, Sec. VI presents our conclusions.

II. Background

In this section we introduce the technologies that we integrate in the VCR approach. Due to space limitations, the overview is necessarily abstract.

Capture and Replay. Foundational to VCR is the capability to reproduce exactly the same sequence of messages and interactions occurred in a monitored run. This is achieved by means of the well-known capture & replay technology. Automated capture & replay tools were the first commercial test products, and are still quite popular. A recorder captures all relevant events and user actions (keyboard hits, mouse movements or

[1]Coincidence of this acronym with the usual meaning of "Video Cassette Recording" is intentional, as both share the notion of recording and replaying.

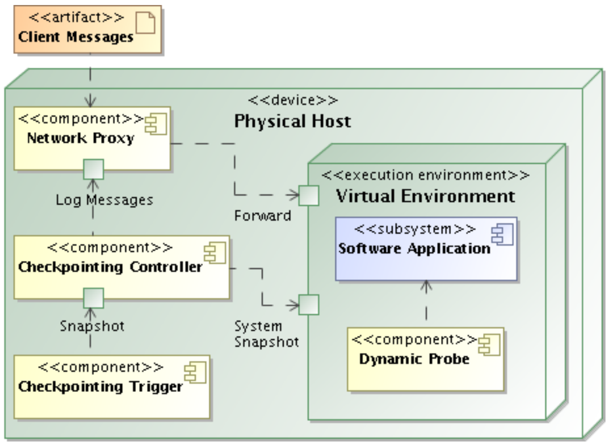

Fig. 1. The VCR architecture

clicks) during a test session. This recorded input is later used to rerun the testing session in an automated way, for instance for regression testing purposes. The technique was conceived for off-line testing, and has been most commonly applied to test Graphical User Interface systems. Joshi and Orso [4] provide a nice introduction to traditional capture & replay technology, to which we refer for space limitation. They also widely discuss its limitations with regard to on-line testing of dynamic applications.

Virtualization and Checkpointing. Virtualization offers a wealth of benefits for testing and debugging purposes. As noticed by King and colleagues [5], virtualization provides developers with the capability to navigate backward and forward the execution history as if driving a "time machine". To start replaying, the complete state of the virtual machine must be reconstructed from the intermediate point at which the execution was stopped: the snapshot of the system state is called a *checkpoint*. Originally conceived for fault-tolerance purposes, checkpointing basically consists in storing a system or application state, so that it may be reconstructed later in time, either on the same machine, or on another machine. The latter case is referred to as *migration*.

Most common implementations of virtualization provide the capability to freeze the state of the controlled virtual environment and to dump an image of it to a file. The features of checkpointing vary across technologies, both in terms of the kind of the frozen information and in performance. Nevertheless the reference implementations of virtualization solutions can save the complete run-time state of the virtual host on-line, with little overhead [6], [7].

III. Approach

The logical architecture of VCR is illustrated in Fig. 1. A physical host (PH) runs one or more virtual environments (VEs), in which the subject applications are hosted.

In addition to the VE, the PH runs at least three other components: the Network Proxy, the Checkpointing Controller, and the Checkpointing Trigger.

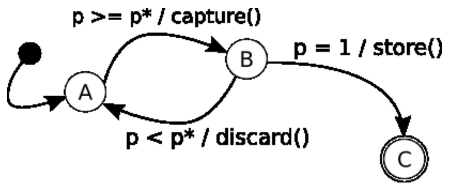

Fig. 2. Opportunistic checkpointing

The purpose of the Network Proxy is twofold: 1) to forward to the VE each client request arriving to the PH, and 2) to collect and store the forwarded messages so that they can be played back in the replay phase. The network proxy exposes a control interface whereby its operation can be dynamically switched between a "save-and-forward" mode and a "forward-only" mode.

The Checkpointing Controller is responsible for dumping the state of the VE. This is done by means of the checkpointing mechanisms that are specific to the adopted virtualization technology[2]. The Checkpointing Controller is also responsible for controlling the capturing of messages by the Network Proxy.

The Checkpointing Trigger encapsulates the policy that is used to determine when a snapshot should be taken. Based on such policy, the Checkpointing Trigger instructs the Checkpointing Controller to take a new snapshot or to discard the one previously taken.

In our experimentation, the Checkpointing Trigger implements an *opportunistic* policy [8], according to which the state of the VE is captured when the QoS of the monitored system violates a given safety threshold.

Capture. Fig. 2 depicts an instantiation of the opportunistic checkpointing policy. During normal system operation (state A in Fig. 2), the Network Proxy is in forward-only mode, while the Checkpointing Trigger continuously computes a measure of the likelihood (p) that the QoS constraints imposed on the subject application will be violated (i.e., p is a measure of how critical the QoS situation is).

When the probability p is such that $p \geq p^*$, the Network Proxy is set to save-and-forward, thus all the requests flowing through it are saved. At the same time, a snapshot of the current state of the system is taken (state B in Fig. 2).

The Checkpointing Trigger remains in B until either p decreases and falls below p^* again, or p increases until it reaches the value of 1.

In the former case Checkpointing Trigger goes back to state A, the Network Proxy is set to "forward-only", and the messages that had been saved are discarded. The VE snapshot is discarded too.

In the latter case ($p = 1$), a predicted QoS constraint has actually been violated. The messages that have been captured up to this point are stored together with the VE snapshot. The snapshot, together with the captured sequence of messages,

[2]The instantiation of VCR described in this paper implements the virtualization by means of OpenVZ [6].

constitutes a test case that can be run again later in the replay phase.

Replay. The replay phase of VCR requires a snapshot of the state of the VE and a temporal sequence of client messages. Both are the result of the capture phase, as described earlier.

When analyzing performance, it is crucial to minimize the alteration of the temporal behavior of the observed system. This alteration, which is inevitably introduced by the measurement, is usually referred to as the *probe effect*.

In order to reduce the probe effect, VCR imposes the following constraints on the analysis tool that is used in the replay phase: 1) the measurement must be done in an efficient and scalable manner; 2) the analysis tool must provide mechanisms allowing to activate and deactivate it at runtime; and 3) the target of the analysis must be adjustable in a dynamic way, allowing to monitor focussed parts of the subject application (e.g., individual classes or methods, in a Java program).

The characteristics of such a tool, which we call Dynamic Probe (Fig. 1), are combined in VCR with the capability of replaying a captured scenario multiple times. As a result, several less invasive observations can be performed by changing the target of the analysis at each repetition, as opposed to extensively monitoring all parts of the application (which would be necessary if the execution was not repeatable).

As an example, in a Java application, a dynamic probe based on JVM TI can be used (as we did) to dynamically instrument one single class (or method) at a time, in order to investigate the activity of the garbage collector, the amount of memory used by objects of a given class, and so forth [9].

In conclusion, the replay phase is performed off-line after the relevant information has been saved in the capture phase. The replay is done by resuming a snapshot of the virtual environment and by activating the Dynamic Probe on a certain part of the application at a time. Then, the subject application is exercised with a given workload according to the goal of the analysis. The replay can be repeated at will, by changing the target of the Dynamic Probe.

IV. Evaluation

The approach presented in this paper has undergone an initial evaluation whose outcome, for space limitations, is only summarized in this section.

Our studies were conducted using OpenVZ[3] [6], installed on a single-processor machine, with an Intel P4 2400MHz CPU and 1.5 gigabytes of RAM. A Debian Linux (Etch) was installed in a VE (its memory was limited to one gigabyte). In the VE, we deployed a subject application, which is a server offering image manipulation operations.

The subject application is a filter applying a "watershed transformation" to client-supplied images. It was implemented in Java, using the Java Advanced Imaging [4] (JAI) library. The application (including the library) counts more than 330 classes. To exercise it, we implemented a workload generator that we used to emulate clients' behavior and to generate a sequence of server invocations with certain timing characteristics and service demand parameters.

Firstly, we examined the effectiveness of the replay mechanism, i.e., how well VCR could reproduce - in the lab - the conditions that were observed during live execution.

The basic idea of this study was to measure CPU utilization and memory occupation in the VE for a certain live scenario and to compare such measures with those taken during the replay of the same scenario.

The results obtained with this study are encouraging, as illustrated in Fig. 3.A, which plots the measures of CPU utilization of a typical run. The measures taken in the capture phase (upper plot) can be easily compared with those taken in the replay phase (lower plot). Similarly, memory occupation measures are presented in Fig. 3.B.

Secondly, we studied the capability of VCR to replay previously captured scenarios without introducing a significant performance penalty. Considering the same application as in the previous study, we implemented an example dynamic probe, based on bytecode instrumentation. Specifically, in order to realize a mechanism whereby the instrumentation could be controlled by an external driver (in the hands of the tester), we implemented a JVM agent based on JVM TI. The agent was attached to the JVM from the beginning, in the normal operation phase, so that it would be available in the replay phase. In the replay phase the agent was used to dynamically instrument a certain part of the application.

In this experience we found that VCR does not introduce a noticeable overhead; however we remark that great care should be taken when choosing the analysis technique to apply in the replay phase, since the instrumentation/probes inserted in the subject application at replay-time could affect performance significantly.

V. Possible Applications

There are several potential uses of VCR. This section briefly discusses some of the most interesting possibilities.

Root-cause analysis. One of the natural uses of VCR is to find the cause of performance problems that were observed during normal (monitored) execution. By saving the state of the VE as a whole, VCR allows to reproduce phenomena that one cannot (or is not willing to) model explicitly. Failures that manifest themselves rarely can be investigated, even when it is not clear what are the inputs and the conditions that are necessary to cause them to happen again.

As an example scenario, let us consider an application deployed on a web server and let us assume that other processes as well are running on the same host. One or more of these processes may be spending most of their time sitting idle in the background, but then from time to time they activate themselves at random, interfering with the web application and ultimately causing the QoS as perceived by the application's end-users to degrade. The difficulty to reproduce and diagnose

[3]Kernel version: 2.6.18-ovz028stab053.5.

[4]`http://java.sun.com/javase/technologies/desktop/media/jai/`

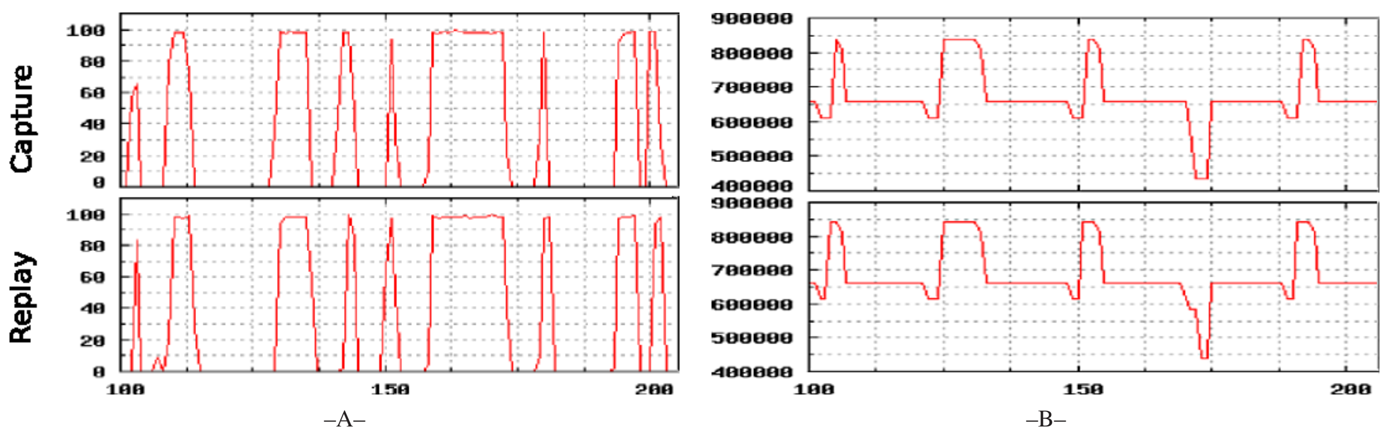

Fig. 3. CPU Usage and Free Memory: Captured vs. Replayed

this type of problems is even greater if they do not happen frequently.

VCR can be useful in different ways: Firstly, it can replay the critical scenario again and again, allowing different techniques to be exploited in the running (i.e. replaying) VE. Secondly, the opportunistic capture approach of VCR can reduce significantly the amount of execution data that is collected at run-time, concentrating only on those scenarios that lead to a performance problem.

What-if analysis. VCR can also be used to do what-if studies, by capturing a scenario as observed in the field and taking it as a baseline for comparative studies.

As an example, one may want to tamper with the clients requests captured during normal operation to obtain a variation of the same workload, in order to understand the impact of different workload parameters on the performance of the subject application. This permits to answer questions such as the following: Could the system handle 15% more users concurrently? Would the service provider obtain a gain by privileging clients (i.e. requests) of a certain type under certain circumstances?

Also, VCR can be used to study the impact on performance of application-specific configuration parameters. As an example, let us suppose we are testing a complex networked application that incorporates load-balancing policies. As system managers, we may want to compare how such policies behave in certain critical situations.

Finally, since OpenVZ allows testers to resume a snapshot on a VE with a different resource configuration, VCR can be used to understand the effect on the application of factors such as available memory, CPU share, maximum number of processes, paging parameters.

VI. Conclusions and Future Work

We have introduced the VCR approach for performance testing that allows for in-house realistic and well-focused root-cause and what-if analyses, by replaying a captured execution run within a virtual environment.

With this paper we have defined the VCR conceptual framework and have also described one possible implementation over OpenVZ that realizes an opportunistic checkpointing policy.

The goal of this work was to develop the approach and assess its feasibility via a working instantiation of an experimental framework for virtual performance testing. This framework, which has been already used for the studies shown in this paper, can be also exploited for further empirical investigation.

In the near future, we intend to compare and evaluate different tools and technologies, in order to find the most efficient combination. This work has just scratched the surface of a wealth of potential utilizations that virtualization technology opens to performance testing. For next work, we plan to perform a more systematic evaluation of the feasibility and efficiency of the approach for performance regression testing; also, we plan to demonstrate empirically the efficiency of some of the many potential applications.

Acknowledgment

This paper has been supported by the European FP6 STREP 26955 PLASTIC and by the FP7 IP 216287 TAS^3. The work has been inspired by discussion with Wolfgang Emmerich.

References

[1] R. Figueiredo, P. A. Dinda, and J. Fortes, “Guest editors’ introduction: Resource virtualization renaissance,” *Computer*, vol. 38, no. 5, pp. 28–31, 2005.

[2] S. Elbaum, H. N. Chin, M. B. Dwyer, and J. Dokulil, “Carving differential unit test cases from system test cases,” in *Proc. of FSE ’06*. ACM, 2006.

[3] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst, “Automatic test factoring for Java,” in *Proc. of ASE ’05*, 2005, pp. 114–123.

[4] S. Joshi and A. Orso, “SCARPE: A Technique and Tool for Selective Record and Replay of Program Executions,” in *Proc. of ICSM ’07*, 2007.

[5] S. T. King, G. W. Dunlap, and P. M. Chen, “Debugging operating systems with time-traveling virtual machines,” in *Proc. of ATEC ’05*. USENIX Association, 2005, pp. 1–15.

[6] “The OpenVZ Project,” `http://openvz.org`.

[7] “Xensource,” `http://www.citrixxenserver.com`.

[8] A. Bertolino, G. De Angelis, S. Elbaum, and A. Sabetta, “Scaling up SLA monitoring in pervasive environments,” in *Proc. of ESSPE ’07*. ACM, 2007.

[9] M. Dmitriev, “Selective profiling of Java applications using dynamic bytecode instrumentation,” in *Proc. of ISPASS ’04*. IEEE Computer Society, 2004, pp. 141–150.