Generic Environment for Full Automation of Benchmarking

Tomáš Kalibera[1], Lubomír Bulej[1,2], Petr Tůma[1]

[1]Distributed Systems Research Group, Department of Software Engineering
Faculty of Mathematics and Physics, Charles University
Malostranské nám. 25, 118 00 Prague, Czech Republic
phone +420-221914267, fax +420-221914323

[2]Institute of Computer Science, Czech Academy of Sciences
Pod Vodárenskou věží 2, 182 07 Prague, Czech Republic
phone +420-266053831

{tomas.kalibera, lubomir.bulej, petr.tuma}@mff.cuni.cz

Abstract. Regression testing is an important part of software quality assurance. We work to extend regression testing to include regression benchmarking, which applies benchmarking to detect regressions in performance. Given the specific requirements of regression benchmarking, many contemporary benchmarks are not directly usable in regression benchmarking. To overcome this, we present a case for designing a generic benchmarking environment that will facilitate the use of contemporary benchmarks in regression benchmarking, analyze the requirements and propose architecture of such an environment.

1 Introduction

The growing complexity of software and the need for distributed development has brought an increased demand for quality control in the software development process. As witnessed in numerous open source projects, automated regression testing of the software under development plays an important role in the quality assurance process. Regression testing, however, mostly covers only the correct functionality of the tested implementation. Regression benchmarking [BKT04a, BKT04b] extends regression testing by also covering the performance of the tested implementation.

Regression benchmarking uses benchmarks to evaluate various performance attributes of the software under development in consecutive snapshots, and analyzes the differences in these snapshots to detect performance regressions. The performance regressions can have many forms, from a slow degradation of performance to various scalability issues or inevitable decrease of performance through added functionality.

To provide useful results, the entire regression benchmarking process must be automatic, so that human attention is needed only when a suspect performance regression has been detected. The requirement of full automation means that a machine, rather than a human, has to deal with obtaining, compiling and deploying both the software under development and the benchmarks, executing the benchmarks on the software under development, monitoring of the execution, and storing and analyzing the results. Most contemporary benchmarks are not suitable for regression benchmarking simply because they do not meet some of these requirements.

As a remedy to the above issues related to full automation, we propose a generic benchmarking environment that supports automated deployment, execution and monitoring of benchmarks and related software, and a repository for storing data in common format that will serve as a data source for analysis and visualization tools. Although some of the contemporary benchmarks will need to be modified or augmented to support the benchmarking environment, we take care to keep the modifications small and typically not intrusive.

In previous work [BKT04a, BKT04b] we have focused on issues of automatic data acquisition and result analysis. The work presented in this paper complements our previous work by elaborating on the issues of automation of the benchmarking process. The rest of the paper is organized as follows: Section 2 analyzes the requirements for designing a generic benchmarking environment for regression benchmarking and points out where this extends the related work, Section 3 proposes the architecture of the environment that meets the requirements set out in Section 2, and Section 4 concludes the paper.

2 Design Requirements for Generic Benchmarking Environment

Our design is primarily driven by generalization of requirements for regression benchmarking, which can be divided intro three groups: installing and configuring the environment, executing and monitoring of benchmarks, and storing of results. We describe these requirements in detail and contrast our approach with related work in other projects that deal with the concept of systematic benchmarking. These projects include the TAO distributed scoreboard [Do04a], the continuous performance metrics for ACE+TAO+CIAO [Do04b], the Skoll continuous distributed quality assurance [Me04], the Lockheed Martin ATL benchmarking tools [ATL04], and the NIST benchmarking tools [Co02].

2.1 Installation and Configuration

Speaking in broad terms, we require the benchmarking environment to be platform-independent, self-contained, extensible, scalable and easy to install.

For reasonable platform independence, the environment must run at least on recent versions of the Linux, Solaris and Windows platforms. The benchmarking environment running on these platforms must interoperate as some benchmarks take place in a heterogeneous distributed environment. Naturally, the benchmarking environment should be open to platform-specific extensions such as monitoring.

The benchmarking environment must be self-contained and easy to install, enough to support automated remote installation and configuration where possible. The automated installation must not require prior or additional installation or configuration of third-party software that is not readily available on the installation platform. While reasonably platform independent, neither of the related projects supports fully automated installation.

The benchmarking environment should support a wide scale of benchmarking sites, ranging from a small developer or research site with few computers that are only occasionally available for benchmarking, to a dedicated benchmarking cluster with hundreds of computers. The scale of the benchmarking site should remain invisible to the benchmarks. Neither of the related projects supports such a scale of benchmarking sites. The target of [Do04a, Do04b, Me04] are mostly individual computers provided by volunteers, while [Co02] is more focused on clusters.

2.2 Executing and Monitoring

The requirements related to executing and monitoring benchmarks are concerned with the robustness of the benchmarking environment in face of failures, which minimizes the required amount of human attention.

Besides the obvious requirement of the benchmarking environment being resilient to failures of any of its components, it must also cope with failures of the benchmarks and related software it executes. Crashes and deadlocks are the most common failures that occur during benchmarking and are easy to detect and resolve. More complicated in that respect are benchmark-specific failures that do not cause the benchmark to crash or deadlock. Regardless of the type of a failure, its impact should be limited to the benchmark where the failure occurred. To our knowledge, neither of the related projects has tackled this issue, except for [Co02], which allows setting of resource limits on executed tasks.

A key requirement associated with regression benchmarking is that except for the software under development, the setup of the benchmarks may not change. The benchmarking environment should therefore support a flexible host scheduling and assignment policy, and ideally detect changes in the setup of the benchmarks. Given the nature of the related projects, this issue only needed attention in [Co02], which supports host assignment with respect to task requirements.

2.3 Storing of Results

The last group of requirements we consider stems from the need for common data format for storing and processing of benchmark results. Most benchmarks produce data in a proprietary format, which prevents using a common set of tools for analysis and visualization.

The common data format must support storing raw benchmark results in the best possible precision, along with a detailed description of the benchmark setup. Each measured attribute should carry an annotation identifying its source and meaning, to allow tracing the results back to their causes. In most projects, identification of the results is responsibility of the user of a benchmark, except for [Co02], where a free-form description of the experiment is associated with the results. In [Do04b], the results come in different formats from multiple benchmarks and are often pre-processed.

Along with the raw benchmark results, the data format should allow attaching secondary information that captures the conditions such as resource utilization under which a benchmark was run, as well as its impact on the conditions. This information helps ensure the validity of benchmark results in presence of constraints on the conditions under which the benchmark should run.

The benchmark results should be kept in a repository that will allow for efficient storage and retrieval. To conserve resources during analysis, the repository should support attaching preprocessed or partially analyzed data to the benchmark results. A result repository is only implemented in [Co02], using a relational database to store the results. The fixed data model limits the flexibility of the repository.

3 Architecture of the Generic Benchmarking Environment

The requirements outlined in Section 2 suggest splitting the architecture of the benchmarking environment into well-defined components with simple and well-specified interaction. The workflow nature of regression benchmarking, with repetitive cycles of deployment, execution, monitoring and analysis, further suggests designing the benchmarking environment as a task processing system. The task processing system will run on each host of a benchmarking site and implement all the benchmarking environment does as specific tasks.

Implementing the task processing system as a Java application and the specific tasks as Java classes helps achieve the requirements of platform independence and extensibility from Section 2.1.

3.1 Task Processing System

Running a benchmark in a heterogeneous distributed environment involves deploying, executing and monitoring the benchmark and related software. These actions may differ in implementation for a specific benchmark or platform, but often share common features such as the implementation of monitoring, the description of requirements, or the process of deployment. This allows encapsulating the common features as simple tasks, used to construct gradually more complex tasks for the actions involved in running the benchmark.

The task processing system will distinguish two types of tasks – *jobs,* which accept input, produce corresponding output and stop, and *services,* which are similar to jobs, except they keep listening for next input. Both types of tasks will consist of their description and implementation, the description defining requirements on the host to launch the task, the list of states of the task, the conditions for launching and terminating the task based on the state of other tasks, and the failure detection and resolution strategy of the task.

The task processing system will schedule the tasks and track the dependencies between the tasks defined by the conditions for launching and terminating the tasks. Depending on the failure detection and resolution strategy of each task, the task processing system will monitor the tasks and handle task failures by ignoring the failed task, restarting the failed task from a checkpoint, or restarting the failed task with a limited number of retries, as appropriate. The task processing system will also facilitate passing of information between the tasks.

Separating the task processing system from the specific tasks helps keep the scale of the benchmarking site invisible to the benchmarks, in line with the requirements of scalability from Section 2.1. The introduction of the failure detection and resolution strategy assists in achieving the requirements of robustness from Section 2.2.

3.2 Benchmarking Tasks

Benchmarking specific tasks will take care of downloading, compiling and executing benchmarks and related software, as well as potential conversion of the results and their storing in the result repository.

Most of these tasks will be common to various benchmarks and platforms, and only tailored for a specific benchmark or platform in their configuration. Checking out source code from CVS is an example of such a task, the configuration will specify the URL of the CVS repository and the target directory. Some tasks, however, will be tailored to a specific benchmark and platform, to make the benchmark fit the benchmarking environment without modification. An example of such a task is populating a database used by a benchmark with data specific to the benchmark.

The existence of tasks tailored to a specific benchmark and platform helps meet the requirements of extensibility from Section 2.1.

The benchmarking tasks will be instantiated by other tasks acting as task generators, starting with a bootstrap task generator. The task generators will rely on a configuration listing the benchmarks to run and the platforms to use, as well as the tasks to schedule for a specific benchmark and platform. Examples of task generators include a generator that instantiates tasks for downloading source code of each configured benchmark, or a generator that instantiates tasks for analyzing and visualizing the benchmark results of each executed benchmark.

Two prominent tasks of the benchmarking environment will be the result repository and the resource manager, both running as services. The result repository is a service used by all tasks that produce benchmark results to store the results. The resource manager is a service used by all task generators that instantiate the benchmarking tasks to allocate exclusive resources such as computers used by the benchmarking tasks. The implementation of the services will accommodate the requirements on executing and monitoring from Section 2.2, as well as the requirements on storing of results from Section 2.3.

3.3 Example Configuration

Figure 1 shows a part of an example configuration of the benchmarking environment running the RUBiS benchmark [Ce04]. Shown are the control host, the client host and the server host as three computers running the task processing system, as well as some tasks.

The control host is central to the configuration, running the result repository and resource manager as two prominent services instantiated by the bootstrap task generator. The client host runs the client emulator job of the RUBiS benchmark, responsible for generating the service load. The job is associated with the actual client emulator process, used without modification from the RUBiS benchmark. The server host runs the database and container of the RUBiS benchmark as two prominent services, again associated with the actual database and container processes from the RUBiS benchmark. The services are used by the initialization, compilation and deployment jobs.

Figure 1 also shows states of the tasks and dependencies between the tasks, depicted by state names in parentheses and wait conditions over arrows. Most services are in the *up* state, most jobs are in the *done* state, except for the client emulator job, which waits for the deployment job to reach the *done* state, and the deployment job, which is in the *running* state. Other dependencies denote already completed waiting of jobs on services.

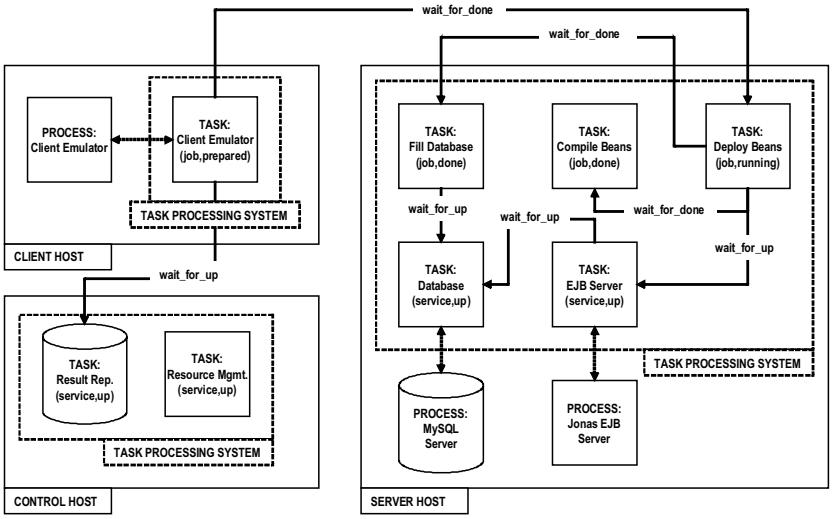

Figure 1: Example configuration of the benchmarking environment running the RUBiS benchmark.

4 Conclusion

The paper is a part of our work to extend regression testing to include regression benchmarking, which applies benchmarking to detect regressions in performance. We have illustrated that the requirement of full automation, inherent to regression benchmarking, is difficult to meet, as it includes automation of tasks such as downloading source code of the benchmarks and related software, compiling the source code, or monitoring of the benchmarks and related software, all of which normally requires human attention. Indeed, most contemporary benchmarks do not meet this requirement, as is the case for example with ECperf [Su04], RUBiS [Ce04], SPECjAppServer2004 [Sp04] or Trade3 [IBM04], which require manual configuration and deployment and can deadlock without terminating.

We have pointed out the difficulties on the related work in other projects that deal with the concept of systematic benchmarking, and proceeded by proposing a generic benchmarking environment based on a task processing system. We have explained why we believe that our design of the benchmarking environment will allow us to overcome the difficulties associated with regression benchmarking.

The paper has been styled more as an overview of the issues associated with full automation, inherent to regression benchmarking, than as a description of the generic benchmarking environment. This is partly because of space considerations, partly because the environment is still a work in progress. For more details, please refer to http://nenya.ms.mff.cuni.cz/been.

Acknowledgements. The work was partially supported by the Grant Agency of the Czech Republic project number 201/03/0911.

References

[ATL04] Advanced Technology Labs, Lockheed Martin Corp.: Agent and Distributed Objects Quality of Service, http://www.atl.external.lmco.com/projects/QoS, 2004.

[BKT04a] Bulej L., Kalibera T., Tůma P.: Regression Benchmarking with Simple Middleware Benchmarks. Proceedings of IPCCC 2004, Phoenix, USA, IEEE CS, 2004.

[BKT04b] Bulej L., Kalibera T., Tůma P.: Repeated Results Analysis for Middleware Regression Benchmarking. Special Issue on Performance Modeling and Evaluation of High-Performance Parallel and Distributed Systems, Performance Evaluation: An International Journal, Elsevier B.V., 2004.

[Ce04] Cecchet E., Chanda A., Elnikety S., Marguerite J., Zwaenepoel W.: Performance Comparison of Middleware Architectures for Generating Dynamic Web Content. Proceedings of Middleware 2004, Rio de Janeiro, Brazil, ACM, 2003.

[Co02] Courson M., Mink A., Marçais G., Traverse B.: An Automated Benchmarking Toolset. Proceedings of HPCN 2000, Amsterdam, The Netherlands, LNCS 1823, Springer Verlag, 2000.

[Do04a] Distributed Object Computing Group: ACE+TAO Distributed Scoreboard, http://www.dre.vanderbilt.edu/scoreboard, 2004.

[Do04b] Distributed Object Computing Group: Continuous Metrics for ACE+TAO+CIAO, http://www.dre.vanderbilt.edu/Stats, 2004.

[IBM04] IBM Corp.: Trade3, http://www.ibm.com/software/webservers/appserv/benchmark3.html, 2004.

[Me04] Memon A., Porter A., Yilmaz C., Nagarajan A., Schmidt D.C., Natarajan B.: Skoll: Distributed Continuous Quality Assurance. Proceedings of ICSE 2004, Edinburgh, Scotland, IEEE CS, 2004.

[Sp04] Standard Performance Evaluation Corporation: SPECjAppServer2004, http://www.specbench.org/jAppServer2004, 2004.

[Su04] Sun Microsystems Inc.: ECperf Specification, http://java.sun.com/j2ee/ecperf, 2004.