Automated Benchmarking and Analysis Tool

Tomas Kalibera, Jakub Lehotsky, David Majda, Branislav Repcek, Michal Tomcanyi, Antonin Tomecek, Petr Tuma, Jaroslav Urban [*]

Distributed Systems Research Group, Department of Software Engineering
Faculty of Mathematics and Physics, Charles University
Malostranske nam. 25, 118 00 Prague, Czech Republic
phone +420-221914232, fax +420-221914323

been@nenya.ms.mff.cuni.cz

ABSTRACT

Benchmarking is an important performance evaluation technique that provides performance data representative of real systems. Such data can be used to verify the results of performance modeling and simulation, or to detect performance changes. Automated benchmarking is an increasingly popular approach to tracking performance changes during software development, which gives developers a timely feedback on their work. In contrast with the advances in modeling and simulation tools, the tools for automated benchmarking are usually being implemented ad–hoc for each project, wasting resources and limiting functionality.

We present the result of project BEEN, a generic tool for automated benchmarking in a heterogeneous distributed environment. BEEN automates all steps of a benchmark experiment from software building and deployment through measurement and load monitoring to the evaluation of results. The notable features include separation of measurement from the evaluation and ability to adaptively scale the benchmark experiment based on the evaluation. BEEN has been designed to facilitate automated detection of performance changes during software development (regression benchmarking).

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: measurement techniques

General Terms

Performance, Measurement

Keywords

Automated benchmarking, Regression benchmarking

[*]The authors are listed in alphabetic order.

1. INTRODUCTION

Coupled with modeling and simulation, benchmarking is an essential technique for performance evaluation. Based on running model applications (benchmarks) in a real system, benchmarking provides performance data representative of a real system. This data is useful for detection of changes in the system, as well as a feedback for modeling and simulation of the system. Recently, regular automated benchmarking has been gaining popularity as a technique for detection of changes in performance during software development [7, 9, 15]. This technique, also known as regression benchmarking [3], is based on benchmarking of daily software versions on the same system.

Benchmarking of a complex system, and automated benchmarking in particular, is a complex task. Still, in contrast with advances in performance modeling and simulation tools, benchmarking is being implemented ad–hoc for each project [7,9,15], wasting resources and loosing generality.

The waste of resources includes not only re–implementing the execution environment for every software to be evaluated. Most current benchmarks are designed to report averages or similar statistics, discarding the raw data. When another statistic, such as median or variance, or a more advanced evaluation, such as clustering of the data, is needed, the benchmark has to be re–implemented and run again, additionally wasting the CPU time. Note that the CPU time can be very expensive when large or enterprise applications are evaluated, because the benchmarks have to be run on a real system. To address these problems, BEEN provides an infrastructure for benchmarks that report raw data. The infrastructure defines a benchmark– and application–independent format of the data, implements a repository of results that can handle the data and supports extensions for benchmark–independent statistical evaluation of the data. BEEN thus allows the re–use of the data for different types of statistical evaluation.

The limited generality of current ad–hoc benchmarking tools includes both the measurement and the evaluation of results. The measurement in general has to cope with random effects at various levels, such as in compilation, in benchmark execution and in individual observations [12, 13]. The random effects are present in real systems, have impact on performance and cannot be filtered out [12]. As a result, in some benchmarks, compilations, executions and observations have to be repeated. The number of necessary repetitions at each level, however, depends on the benchmark and the required result precision. Current ad–hoc benchmarking tools usually support only a predefined number of repetitions at the level of executions and observations, possibly loosing precision due to random effects in compilation, or wasting CPU time by suboptimal choice of the numbers of repetitions [13].

In ad–hoc benchmarking tools, the analysis of performance results is frequently limited in that it does not allow planning of new measurements based on the results. This feature is important for statistical evaluation of the results, because of the natural variation in performance of a benchmark that is present even when there is no change in the system. When the variation is too large, additional measurements using the same benchmarks are needed to filter it out. The automated detection of changes in regression benchmarking is an example of a benchmarking application where such an analysis is important. As a generic benchmarking tool, BEEN supports repeating compilation, execution and observations, as well as statistical evaluation methods that are independent of the measurement and may adaptively plan additional repetitions. BEEN is designed to support automated regression benchmarking, covering all its steps from automated downloading through measurement and automated detection of changes to visualization of results.

The project definition of BEEN has been presented in [11]. Currently, BEEN is available in a beta version, described in this paper in more detail. The related projects are described in Section 2. The architecture of the tool is outlined in Section 3 and the functionality is detailed in Sections 4 and 5. The current implementation is evaluated on a distributed remote procedure call benchmark in Section 6. The paper is concluded in Section 7.

2. RELATED WORK

Among the related tools are tools for automated performance monitoring during software development, generic tools for automated distributed testing and generic tools for automated distributed benchmarking.

The tools for automated performance monitoring during software development include TAO Performance Scoreboard [9], A Real-Time Java Benchmarking Framework [15], Lockheed Martin ATL Benchmarking Tools [1] and Mono Regression Benchmarking Project [7]. These tools were all created for use in a particular software project. [9] and [1] are focused at CORBA, [15] is for Java applications and [7] is for Mono/C# applications. Only [7] features automated detection of changes. Porting the tools for use in other software projects would require a significant additional effort.

The Skoll Project [14] started as a tool for distributed software testing that used computing resources provided by outside volunteers. One of many challenges of the project is finding a minimal set of tested software configurations that would still discover potential problems in any configuration. Currently, the project also covers regression benchmarking [18], focusing on finding benchmarks and configurations that are most sensible to performance problems present in any configuration. Such benchmarks and configurations are first found using the computing resources provided by outside volunteers, and then precisely evaluated on dedicated computers. Within this context, BEEN is a tool for the precise performance evaluation.

The CLIF Tool [6] is a load injection framework targeted primarily at Java middleware. It covers deployment, monitoring and storing of results. The tool is capable of a highly configurable distributed load injection, emulating for example clients accessing a web site. BEEN does not aspire to provide the load injection support for general benchmarks, but is able to run benchmarks that use [6] for load injection, adding runtime monitoring, results repository and automated evaluation of results. The results repository of [6] is limited in comparison.

The NIST Automated Benchmarking Toolset [4] is a generic tool for automated benchmarking in a grid environment. The tool uses a common format for storing results in a relational database. It relies on the Distributed Queueing System [16] as its execution environment and shell scripts as its task implementation language, therefore, the support for Windows platforms is limited. The tool is no longer being developed and the source code is not available.

3. ARCHITECTURE

The main design goal of BEEN is to support automated benchmarking in a distributed heterogeneous environment. The automated benchmarking involves compilation of software to be benchmarked, compilation of benchmarks, deployment, running the benchmarks and collecting, evaluating and visualizing the results. Many of these issues are general for any automated execution of software in a distributed heterogeneous environment – these are covered by the *execution framework*. The benchmarking specific issues are covered by the *benchmarking framework*, built on top of the execution framework. Both frameworks can be administered and controlled from a unified web based user environment. The BEEN architecture is illustrated in Figure 1.

3.1 Execution Framework

The execution framework is designed to execute tasks in a distributed system, supporting different host platforms without requiring system configuration changes to the host computers. The main components of the execution framework are *Host Runtime*, *Host Manager*, *Task Manager* and *Software Repository*. To allow a unified view on different hosts in the system, each host has to run the Host Runtime, which is used by other BEEN components to communicate with that host. The Host Runtime is capable of running tasks locally on each host, providing logging and monitoring facilities.

The availability of the hosts can change over time, both intentionally when the administrator adds or removes hosts, or as a result of a network or hardware failure. The Host Manager maintains a list of the currently available hosts as well as their current hardware and software configuration. It allows addition and removal of hosts by the administrator and lookup of hosts based on their configuration.

The execution of tasks in the distributed environment is coordinated by the Task Manager. The Task Manager allocates hosts to tasks based on the tasks requirements, monitors the running tasks and resolves task failures. The executable code and static data of the tasks are stored in the Software Repository.

The execution framework is designed with benchmarking in mind. This requires that the framework is capable of running a task exclusively on a particular host, because otherwise the task performance could be affected by concurrently running tasks. By employing the Software Repository, the framework avoids the dependence on a file system shared by multiple hosts, which can also potentially distort performance of running tasks.

3.2 Benchmarking Framework

The benchmarking framework is designed to support two different kinds of performance analysis – a traditional evaluation of performance and a repetitive evaluation of performance for regression benchmarking. The main components of the benchmarking framework are the *Benchmark Manager* and the *Results Repository*.

The Benchmark Manager submits tasks needed for execution of a particular benchmark to the Task Manager. These include tasks for compilation of the evaluated software, compilation of the benchmark, execution of the benchmark and collecting of its results. The benchmark results are stored in the Results Repository in a raw format that contains individual benchmark measurements. The results can be statistically processed and visualized by the repository itself or by configurable repository extensions.

3.3 User Interface

The distributed components of BEEN can be monitored and controlled from a single web user interface. The interface provides both high–level operations, such as starting a benchmark and viewing its results, as well as low–level operations, such as executing a task that downloads particular software into the Software Repository. For the system administrator, the interface provides a detailed information on all the BEEN components. The user interface runs independently and can be shut down while other BEEN components are running.

4. EXECUTION FRAMEWORK

The execution framework of BEEN provides a simple interface for running *tasks* in a distributed environment, hiding differences between various operating systems and platforms. Platform independence is accomplished through using Java: each task has to be wrapped within a Java class and be executable from the Java Virtual Machine (JVM). The distribution is implemented using Java Remote Method Invocation (RMI) as the communication protocol. The code, static data and, optionally, platform dependent binaries needed to run the tasks are stored in task *packages*.

Depending on its mode of execution, each task is either a *job* or a *service*. A job is a batch task created for a particular action defined by job parameters, code and input data. A job finishes as soon as the action it was created for is performed. A service is an interactive task that waits for requests from other tasks and performs actions upon those requests. A service has to be stopped explicitly, either by

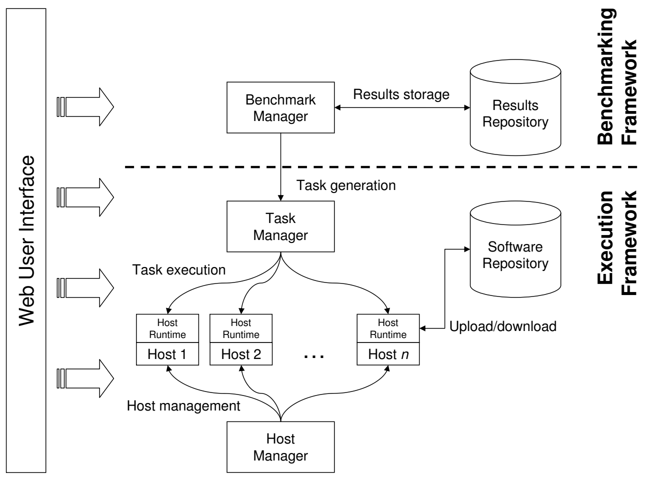

Figure 1: BEEN architecture.

other tasks, by the environment, or by the administrator. Most of BEEN components are themselves implemented as services: Software Repository, Host Manager, Benchmark Manager, and Results Repository.

The execution of a task is started by submitting a *task descriptor* to the Task Manager. Based on the hardware and software requirements described in the task descriptor and on the current utilization of available hosts, the Task Manager allocates a suitable host for running the task and instructs the Host Runtime of that host to run the task. The Host Runtime then downloads the task package from the Software Repository and executes it with parameters specified in the task descriptor. The hardware and software configuration requirements are matched against a host database maintained by the Host Manager. The database is updated automatically to match the current state of available hosts.

Cooperating tasks can synchronize via *checkpoints*. By creating a checkpoint, a task declares it has reached a particular stage of its execution, possibly attaching additional information to the checkpoint. Predefined checkpoints indicating that a task has started or finished are created implicitly. The execution of a task can be suspended until another task reaches a particular checkpoint, either by describing the condition in the task descriptor or by waiting for the checkpoint at run time. A simple use of checkpoints is creating a sequence of tasks, where each task waits for its predecessor in the sequence to finish. Each group of cooperating tasks is enclosed in a *context*. An example of such a group is a compilation context, formed by a sequence of three tasks that download the software sources from a public versioning system, compile the sources and upload the binaries to the Software Repository, respectively. A task can only synchronize with tasks from the same context. The base context where the BEEN components are run is an exception to this rule, as any task can synchronize on checkpoints of tasks in the base context.

A more complex example of the use of checkpoints is execution of a client–server benchmark, consisting of a client task and a server task. By creating a checkpoint, the server task declares that it is ready to receive requests. It attaches its serialized reference to the checkpoint. The execution of the client task depends on the server task reaching this checkpoint. When the client executes, it uses the serialized reference from the checkpoint to connect to the server.

The execution framework is designed to detect and handle failures of the running tasks. These failures can be caused by unhandled language exceptions, which are detected in a straightforward manner, or by infinite loops and deadlocks, which can be detected when the running tasks exhibit too high or too low processor utilization. Processor utilization information is a part of the host utilization information, monitored by the Host Runtime of each host and stored in the host database of the Host Manager.

4.1 Host Runtime

Each instance of the Host Runtime represents a single host in the execution framework. The runtime instances provide an interface for running tasks on the associated host and monitor the host utilization. To the running tasks, the Host Runtime provides a control interface for synchronization through checkpoints, an interface for registration and lookup of services, and interfaces for logging and adjustment of the utilization monitoring.

When requested to execute a task, the Host Runtime downloads the task package with the necessary code from the Software Repository. To save network communication, the Host Runtime caches the packages on each host. The maintenance of the package cache is simple, because once a package is uploaded to the Software repository, it cannot be modified.

Each task is run in a separate JVM, with the environment variables and command line arguments set as described in the task descriptor. The default arguments for the JVM, the name of the JAR archive to use and the name of the application class to execute are included in the metadata of the task package. Once running, the task communicates with the Host Runtime on the same host by RMI. Although launching a new JVM for each task brings certain overhead, it ensures reliable isolation of the running tasks and the Host Runtime. In our experience, errors in software under development can crash a JVM. By running the Host Runtime in a separate JVM, we ensure that the Host Runtime does not crash as a result of an error in a task.

The Host Runtime allocates three different directories in the local file system for each running task: the *task directory*, the *working directory* and the *temporary directory*. The task directory contains the extracted task package, from which the running task can read its static data. The temporary directory is intended for temporary files of the task. It is empty at task startup and deleted after the task terminates. The working directory of the task, also empty at task startup, allows the task to leave its output for the other tasks it cooperates with, provided both tasks are run on the same host. The directory is deleted when the context of the cooperating tasks is removed. A task can also use its working directory to store its state so that it can recover when restarted after a crash. The majority of BEEN components is designed to have this capability.

The Host Runtime is responsible for monitoring tasks and reporting the task failures to the Task Manager, as well as cleaning up the temporary directories of the failed tasks and shutting down of all processes started by the failed tasks. A task failure is detected when its JVM process exits with an error, when its *execution timeout* is exceeded, or when its processor utilization falls outside predefined limits. Both the timeout value and the utilization limits can be set in the task descriptor. The utilization limits are useful for detecting deadlocks and infinite loops in *exclusive* tasks. When running, an exclusive task is the only task on a host. All benchmarks are run as exclusive tasks to avoid distortion of the reported performance.

The host utilization is monitored by the *Utilization Monitor*, which is a part of the Host Runtime. Two modes of acquiring utilization information are supported: the *brief utilization* mode provides an overview of the host activity, the *detailed utilization* mode provides detailed utilization information about individual operating system processes. The brief utilization mode is used whenever the Host Runtime is running, collecting information on processor utilization, memory usage, disk usage and network activity. The information is periodically uploaded to the *Utilization Server*, which is a part of Host Manager, and used to detect failed tasks and failed hosts. The detailed utilization mode is used only when requested by a running task. The information is stored locally and used to supplement the performance results. A benchmark task that requests monitoring in the detailed utilization mode can be followed by another task within the same context that will upload the data to the Results Repository for further processing.

Since Java does not provide facilities to acquire the utilization information, native libraries are provided for supported platforms, currently Windows and Linux. The native libraries use operating system specific calls. On unsupported platforms, the Utilization Monitor does not support monitoring in the detailed utilization mode, and instead of monitoring in the brief utilization mode, it only informs the Host Manager periodically that the particular host is alive.

The Host Runtime is designed to recover from possible failures of the host it runs on or of the BEEN components it communicates with. The Host Runtime maintains a log of its state, from which it can recover when restarted after a failure. It also intercepts all communication of the tasks with the BEEN components and delays it when the components cannot be reached. The running tasks are therefore resilient against temporary failures of the Task Manager, which would otherwise become a single point of failure for the entire distributed execution system.

4.2 Task Manager

The Task Manager is responsible for scheduling, monitoring and controlling of tasks in the distributed environment. The scheduling decision is based on the hardware and software requirements of the tasks and on their dependencies on checkpoints reached by other tasks. The monitoring covers checkpoints reached by running tasks, failures of running tasks and failures of hosts on which the tasks are running.

The controlling includes restarting of failed tasks and stopping of tasks on demand.

A task is created by submitting a task descriptor to the Task Manager. The task descriptor can be submitted either by another running task, such as the Benchmark Manager, or manually through the user environment. The task descriptor tells the Task Manager how to allocate a host to the task, when to run the task, how to run the task and what to do if the task fails.

The host allocation is based on software and hardware requirements of the task, which are interpreted by the Host Manager. The specification of the software and hardware requirements is described in Section 4.3 in more detail. The host allocation algorithm balances the load among available hosts and ensures that exclusive tasks are run alone on the assigned host. As a special case of a host requirement, the specification can refer directly to an individual host.

Unless specified otherwise, a task is scheduled for execution immediately after its task descriptor is submitted to the Task Manager. Sequentially submitted tasks can run in parallel. Conditions that determine when to run a task can be specified using checkpoints, which allow a task to either wait for a particular checkpoint of a particular task to be reached, or to wait for a particular value of such a checkpoint. The task identification scheme makes it is possible to wait for a checkpoint of a task that has not been submitted to the Task Manager yet, thus increasing flexibility of both task descriptor submission and task synchronization. In addition to synchronization, the checkpoints also help the user to track the progress of the tasks.

The package with the task code is specified by a list of its features, such as name of the software, range of versions, supported platforms and build options. The specification is processed by the Software Repository, which stores the packages. Based on the packages currently available in the Software Repository, multiple packages can match the specification in the task descriptor. A single matching package is then chosen at random.

The Task Manager is informed on the checkpoints of the running tasks by the Host Runtime instances running the tasks. This includes information about task failures, which can be either reported explicitly, or inferred implicitly when a task *execution timeout* exceeds or the host a task is running on, or the associated instance of the Host Runtime, fail. The task execution timeout is specified in the task descriptor. In case of task failure, the Task Manager attempts to repeat the failed task until the maximum number of retries specified in the task descriptor is reached. The same host is used when the respective host and Host Runtime are alive; another host matching the host requirements is chosen otherwise.

In addition to starting a task, the termination of a task can also be tied to another task reaching a checkpoint. This feature is important for services, which can be stopped when all cooperating tasks that use the service terminate. Examples of services include database servers in a database benchmark or directory services in a distributed client–server benchmark. Finally, cooperating tasks can also be stopped by destroying their context.

The Task Manager keeps track of important information about running tasks, which makes it a single point of failure of the entire system. To minimize the impact of possible failure, the Task Manager maintains a log of its state and can recover when restarted after a crash. Additionally, the Host Runtime is designed to withstand a temporary failure of the Task Manager. A temporary failure of the Task Manager therefore does not cause the running tasks to fail.

4.3 Host Manager

The Host Manager is a service responsible for maintaining the list of accessible hosts in the execution framework and for managing the database with a hardware and software specification of each host. The Host Manager provides means for administration of hosts, including tools for adding and removing hosts and support for lookup of hosts based on their specification. The Host Manager is also responsible for monitoring the availability and utilization of the hosts. The host database can *group* hosts based on various criteria, simplifying the host management in large networks.

The *host database* stores a list of hosts, as well as the specification of their hardware and software. Since Java does not allow direct interaction with the underlying operating system needed to detect the installed hardware and software, native *detector* libraries are provided for the task. Currently, Linux and Windows platforms are supported by native detectors that query processor features, hard disks, disk partitions, installed software and operating system features. Detectors can update the information about a host in periodic intervals or on user demand. Each configuration update adds a new entry to the configuration history of the host. The configuration history can later be browsed through the user interface, allowing for an easy review of hardware and software changes and relating of these changes to the benchmark results.

Each host in the host database is represented by a tree of *objects* described by *properties*. The structure of the tree is based on the way hardware and software components relate to each other, with each child node adding more detail about its parent node. As an example of this arrangement, an object representing a hard drive is a parent of objects representing partitions of that hard drive. The properties are typed and identified by a path from the root of the tree.

The Host Manager provides two methods for querying the host database. In the first method, the user specifies *restrictions* on the hosts configuration. The second method requires the user to implement a more general *query interface* for matching the host configuration.

Restrictions provide the user with means to specify a set of conditions, which are essentially logical expressions over properties in conjunctive normal form. Several types of relations on properties are supported, including exact and interval match as well as regular expression match.

As an alternative to restrictions, the implementation of a query interface can be passed to the Host Manager using RMI. The implementation can access the entire host database and express complex queries that cannot be expressed as restrictions.

Not all platforms support remote execution by default, and thus the Host Manager has no means to start a Host Runtime instance on a remote host automatically. On such platforms, the Host Runtime can be started manually by the administrator. The Host Manager is still designed to be extensible to support automated starting of Host Runtime on particular platforms with a particular remote execution system, such as Secure Shell (SSH) on Unix or Windows.

4.4 Software Repository

The Software Repository is a service for storage and retrieval of all software run by the execution framework. It stores the software binaries for different platforms, as well as the static data and the sources. By using the Software Repository, the execution framework avoids relying on a distributed file system, which might be difficult to set up in a heterogeneous environment. When executing benchmarks, the presence of a distributed file system could also distort the benchmark results.

The basic storage unit in the Software Repository is a *package*. A package is a ZIP archive with *metadata* file and any additional files and directories. The metadata is stored in an XML format defined by the Software Repository and include package name, package version, type and textual description of the package. Presence of additional metadata depends on the particular package type:

- *Source package* – contains software source code, such as source of an application to be used for benchmarking. Additional metadata include supported compilers and platforms.
- *Binary package* – contains compiled software. Additional metadata include supported platforms and a description of how the software was compiled.
- *Task package* – contains a task for the execution framework in the form of Java bytecode. Additional metadata include the default JVM arguments and the name of the class to execute.
- *Data package* – contains any static data files, such as an initial database snapshot for a database benchmark.

The operations supported by the Software Repository are uploading a new package, downloading a package, deleting a package and looking up a package based on its metadata. Notably, instead of modifying a package, a new version of the package has to be created. This restriction helps to maintain coherency of package caches. The lookup of packages uses lookup code provided by the client over RMI and executed by the Software Repository. The code can analyze the package metadata in an arbitrary way.

The Software Repository is designed for transfer of large packages and for a fast lookup of packages. The transfer optimizations include asynchronous communication and a special interface provided for monitoring of the transfer progress. To improve the lookup performance, package metadata are cached at package upload and all lookup operations are processed using the cache.

The user interface to the Software Repository allows manual browsing and lookup of packages, viewing package metadata, as well as package upload and deletion. The lookup provided by the user interface is based on logical expressions over package metadata.

4.5 User Interface

The web user interface of BEEN provides an unified access to the components of the execution environment.

Via the Host Manager, it allows adding hosts to and removing from the environment, checking their availability, checking their detailed configuration, checking their current utilization, viewing their configuration history, and looking-up hosts matching given configuration requirements.

Via the Software Repository, the user interface allows browsing available software packages, looking up packages matching given requirements, and uploading to and deleting packages from the repository.

Via the Task Manager, the user interface allows monitoring currently running tasks, viewing their checkpoints, viewing their log information, and manually executing and stopping the tasks.

5. BENCHMARKING FRAMEWORK

The benchmarking framework supports fully automated benchmarking on top of the distributed execution framework described in Section 4. A single *benchmarking experiment* covers downloading of software sources, compilation, deployment, execution, measurement, statistical evaluation and visualization of results. The benchmarking framework supports a range of features, such as a separation of the measurement from the evaluation or planning of additional measurements based on evaluation, with only a minimum set of requirements on the benchmarks.

The benchmarking environment supports two distinct purposes of benchmark experiments, *comparison analysis* and *regression analysis*. The comparison analysis determines the impact of configuration change on the performance of the benchmarked software, using experiments where the benchmarked software does not change and the configuration varies in a small set of features. Examples of comparison analysis include determining a communication library implementation or configuration that maximizes the performance of the benchmark. In contrast, the regression analysis determines the impact of version change on the performance of the benchmarked software, using experiments where the benchmarked software changes and the configuration is the same. In the execution framework, both types of experiments are implemented as sets of cooperating tasks performing the necessary actions.

The benchmark experiments are subject to random effects in compilation, execution and measurement [12]. The random effects can have impact on performance, prompting the need for repeating the individual steps of the experiments. A benchmark experiment can be designed to adaptively optimize the numbers of compilations, executions and measurements to get the best precision in a given time for the experiment [13].

The benchmarking framework consists of the Benchmark Manager, which is responsible for managing benchmark analyses and running benchmark experiments, and of the Results Repository, which is responsible for data storage, statistical evaluation and visualization. Both the Benchmark Manager and the Results Repository are designed to be extensible to support different benchmarks.

Although emphasis is put on having only a minimum set of requirements on the benchmarks, some parts of a benchmark experiment are necessarily specific to a given benchmark. These parts must be provided by the benchmark in the form of benchmark plugins, packaged in a single *benchmark module*. Examples of plugins include task packages required for software download, benchmark compilation, execution and for conversion of results into a common format used by the Results Repository. Since many benchmarks use standardized software repositories and compilation tools, many of the plugins do not need to be implemented individually for each benchmark.

The benchmarking framework is designed to allow transparent and repeatable benchmarking. By employing the logging and monitoring facilities of the execution framework, a log of all potentially relevant events, as well as a listing of the current software and hardware configuration of all the involved hosts and the system utilization information of the hosts is attached to the benchmark results. The logs are stored in the Results Repository for later inspection in case an inconsistency in the results is encountered. In addition to manual inspection, the logs are used by the framework to recreate identical conditions when a repetition of a benchmark experiment is requested.

5.1 Benchmark Manager

Given a description of a benchmark analysis to perform, the Benchmark Manager is responsible for planning the corresponding benchmark experiments. A benchmark experiment consists of tasks for compilation of the benchmarked software, compilation of the benchmark itself, deployment and execution of the benchmark, and collection and evaluation of the results. Each of the tasks has specific requirements on the hosts it can use, which are particular to the benchmark analysis, the benchmarked software, and the benchmark itself. The Benchmark Manager is responsible for gathering these requirements and creating the tasks with maximum utilization of the available hosts in mind.

The creation of the *compilation* tasks is driven by the requirements of the platforms on which the benchmark will execute. The compilation tasks do not require running on the same host as the benchmark, but they must be performed on hosts that can compile binaries for the platform on which the benchmark will execute. Likely, there will be multiple hosts meeting this requirement, and the decision which of them to use should be based on the host utilization. The final decision is therefore left upon the Task Manager, which gathers the utilization data and selects the hosts based on the requirements supplied with the compilation tasks. A distributed benchmark may require a different platform for each of its parts. The planning of compilation tasks has to support this option, possibly also using multiple hosts for compilation. Multiple benchmarks of the same software can reuse the software binaries, provided that the same compile time configuration of the benchmarked software is used.

Although similar in principle, the host allocation process for the *benchmark execution* tasks is more complex. A distributed benchmark can require allocating tasks on several hosts, each host potentially having a unique *role* in the benchmark. As an example, a client–server benchmark can require one host in the role of the server and several hosts in the roles of clients. The server role may require a particular web server or component container implementation, the client hosts may similarly require a specific communication middleware. These many constraints on the host platforms are supplied by the benchmark plugins and have to be accommodated together with other requirements of the benchmarking analysis.

Finally, the host allocation for the *result collection* tasks is quite simple once the hosts for the benchmark are known. The benchmark plugins supply the information on which hosts the benchmark produces results, these are the hosts that will run the result collection tasks. At experiment creation time, the Benchmark Manager also informs the Results Repository about the results expected from the experiment. Using this information, the Results Repository can tell when it has received all the results and start the evaluation. Alongside the results, the description of the experiment and the logs of all the hosts involved in the benchmark experiment are uploaded to the Results Repository. This makes it possible to reproduce the experiment.

Planning a comparison analysis

When planning for a comparison analysis, the benchmark experiment evaluates several systems that differ in a small set of features using the same benchmark. The benchmark and the features that vary are selected by the user at the analysis creation time. The analysis proceeds in two semi–automatic steps. In the first step, the allocation of hosts that match the restrictions given by user, the benchmark requirements, the benchmarked software requirements, and the compilation requirements is performed as described above, yielding a set of benchmark experiments iterating over all the available settings for the features that were selected to vary in the analysis. Considering an example analysis of a web server benchmark performance under varying amount of system memory, one experiment will be created for each amount of system memory available among the BEEN hosts that meet the requirements to execute the web server in the benchmark.

In the second step of the analysis, the user is given the option of manually pruning the set of experiments and modifying the hosts selected for each role in the experiment. The user can also customize the benchmark using benchmark parameters, such as specifying the length of the warm-up phase or, in the web server benchmark example, the number of web users simulated by each client. The tunable parameters are specific to each benchmark and specified by the benchmark plugins.

The benchmarking experiments start immediately after the user commits the changes to the analysis. The task descriptors, interlinked by checkpoints, are generated and submitted to the Task Manager. In order to avoid possible distortion of results by tasks unrelated to the analysis, exclusive tasks are used to execute the benchmarks.

Planning a regression analysis

The regression analysis compares the performance of newly produced software versions to the previous software versions using the same benchmark. The main goal of regression analysis is locating changes from version to version that impact the observed performance. To make regression analysis possible, the results of the benchmark must reflect only the changes from version to version of the benchmarked software, there must be no other changes in the benchmarked system. Even a small modification of the benchmarked system, such as a routine security update, can impact performance and thus distort the regression analysis. When planning for a regression analysis, the benchmark tasks are therefore executed on the very same hosts for each software version. The compilation can still execute on any system that meets the requirements.

The host allocation in regression analysis is also a two–step semi–automatic process, similar to the host allocation in the comparison analysis. In the first step, the user selects the particular software and benchmark for the regression analysis and, optionally, restricts the host allocation for each role in the benchmark. It is advisable to restrict the allocation to reliable hosts that will remain available over a long period of time and will not be subject to upgrades. At the same time, if there is only a limited number of hosts that have the required compilers, they should not be blocked by the exclusive benchmarking tasks to maximize throughput of the whole system.

Based on the user selection, a benchmark experiment template is created. The template describes a single benchmark experiment with host allocation for benchmark execution and hosts requirements for the other tasks making up the experiment. The template, however, does not specify the version of the benchmarked software. In the second step of the analysis, the user can again manually modify the experiment template. Once the user commits the template, the individual experiments are created automatically based on the existence of the benchmarked software versions.

The process of planning a benchmark experiment is necessarily bound to the specifics of software download and compilation, benchmark parameter adjustment and other details. These differ from experiment to experiment and therefore cannot be handled in a generic manner by the benchmarking framework without the help of benchmark plugins. Many of the plugins, however, can be shared by more benchmarks. Examples of shared plugins include tasks for downloading software from common repositories such as CVS or SVN, or tasks for compiling software through the autoconf tool. Ideally, software vendors would distribute other necessary plugins to support their software in the benchmarking environment in the form of *software modules*, thus saving the work of the benchmark developers and simplifying reuse of the software in different benchmarks. The software modules can be stored in a centralized repository, similar to packages for various Linux distributions. Until this ideal scenario comes to pass, however, the software modules need to be packaged into the benchmark modules.

5.2 Results Repository

The Results Repository is responsible for persistent storage of benchmark results, logs related to the execution of benchmarks, and for statistical evaluation and visualization of the results. The results are stored in a benchmark–independent format that preserves information on individual measurements, and thus can be used for various types of evaluation, both benchmark–independent and benchmark–specific.

The Results Repository uses two distinct data formats, a textual format with emphasis on portability and a binary format with emphasis on efficiency. In the textual format, measurements from each benchmark execution are stored in a separate text file. The text file is a Comma–Separated–Values file representing a table with measurements in rows and metrics in columns. Each measurement consists of the same metrics, which are benchmark–specific. Often, there is only one metric per measurement, namely the current time in processor clock ticks. The textual format is highly portable and can easily be supported by any benchmark on any platform without dependencies on external libraries. The format, however, is not suitable for the results evaluation, because it is space–inefficient and does not allow random access to individual measurements.

The binary format used by the Results Repository stores the measurements from each benchmark execution in a separate NetCDF file. The NetCDF format [17] supports platform–independent storage of multidimensional arrays with one extensible dimension and allows random access to array elements in constant time. Although the format is suitable for storing measurements from a single benchmark execution, it is not suitable for storing measurements from multiple benchmark executions or even multiple *benchmark binaries*, because it does not support non–rectangular arrays. Measurements from multiple benchmark binaries of the same benchmark and benchmarked software are needed due to random effects in compilation [12, 13]. The benchmark results can be non–rectangular, because different benchmark executions in the same benchmark experiment can have different numbers of measurements. Similarly, different benchmark binaries can be benchmarked by different numbers of benchmark executions. The Results Repository therefore uses a file system directory tree to group results based on their relation to benchmark experiments and benchmark binaries.

The Results Repository is designed to be an easy–to–use platform for statistical evaluation and visualization of benchmark results. To simplify the implementation of specific statistical evaluation and visualization plugins, the Results Repository supports plugins written in the R language. The R language is a part of the R Project [10], a tool for statistical computing and visualization with a number of freely available extensions for new statistical methods. The R language itself is freely available and supports most current platforms. The R runtime environment can be easily linked to the JVM and accessed via Java Native Interface

(JNI). The R plugins can thus use the Java part of the Results Repository for communication with the benchmarking environment, such as for planning additional measurements when the variation of the results is too high. The NetCDF format is supported by R, hence the plugins can also directly access the benchmark results.

The statistical evaluation supported by the Results Repository includes built–in calculation of basic statistics and pluggable evaluation and visualization, which can be benchmark–specific, analysis–specific or fully generic. The individual plugins can store their intermediate and final results in the repository and can access results stored by other plugins. In particular, any plugin can access the basic statistics calculated by the built–in code. The list of plugins to use for the evaluation of a specific benchmark analysis, as well as the parameters of the plugins, are set at the benchmark analysis creation time.

The built–in evaluation includes calculation of sample average, variance, median and quartiles of measurements within each benchmark execution, and higher–level statistics for all executions from a single benchmark binary, as well as for all executions from a benchmarking experiment. In the regression analysis, a benchmarking experiment corresponds to a single software version, and thus the basic statistics include mean and median of all measurements of each software version. Additional generic evaluation, such as calculation of confidence intervals, analysis of variance or calculation of impact factors of random effects [12, 13], can be implemented by generic plugins.

The Results Repository contains one analysis–specific plugin for each supported benchmark analysis type. The plugin for the comparison analysis uses the basic statistics to create graphs that allow visual comparison of performance of the benchmarked systems. The plugin for the regression analysis automatically detects mean performance changes between consecutive software versions using statistical methods described in [13]. The plugin generates graphs with marked performance changes and a list of the changes. The plugin can be set to send notification messages on newly detected changes.

The benchmark–specific plugins are intended for additional evaluation of a benchmark analysis performed by a specific benchmark. A benchmark–specific evaluation is required when interrelation of different metrics measured by the benchmark is of interest, or when the metrics are not of numeric types.

All results stored in the Results Repository are presented to the user through the integrated BEEN user interface. The visualization plugins can generate any images, tables and graphs, which are then displayed by the user interface. Adding a new plugin to the Results Repository therefore does not require modification of the user interface implementation.

5.3 User Interface

The web user interface provides an unified access to the benchmarks known to the environment. The interface is integrated with the interface of the execution environment. Via the Benchmark Manager, the user interface supports defining a new benchmark analysis using a given benchmark, starting, suspending and resuming the analysis, and monitoring the running benchmarks. The monitoring interface for the benchmark is inter-connected with the interface of the execution environment, and thus individual tasks that form the running benchmarks can be monitored as well.

Via the Results Repository, the user interface supports viewing the benchmark results and the configuration information of the hosts the benchmarks were run on. The results are presented in tables of basic statistics, by comparison graphs, and also as raw data. The results of a regression analysis shall be annotated with a list of detected performance changes and specialized graphs of changes.

6. EVALUATION

The ambition of BEEN is to provide a generic distributed and multi–platform execution framework with a generic benchmarking framework built upon it. It is designed to be easy to install, easy to use and run automatically without user intervention. These qualities, however, can only be evaluated by using BEEN for a diverse set of benchmarks, platforms and software. The implementation of BEEN is in beta stage, currently capable of handling a comparison analysis of a nontrivial distributed benchmark. We therefore base the evaluation on handling a comparison analysis with the Xampler benchmark from the CORBA Benchmarking Project [8], which not only shows that BEEN is indeed usable, but also that BEEN saves coding effort when performing benchmark experiments.

Xampler is a distributed client–server CORBA benchmark consisting of a server process and a client process. The client repeatedly invokes a method on the server using the CORBA remote procedure call and measures the method invocation time. In the evaluation, we have created benchmark module that supports comparison analysis using the Xampler benchmark running on the omniORB broker [5]. The analysis of the results has been performed using benchmark–independent plugins that produce basic statistics and a comparison graph. The benchmark plugins specific to the Xampler benchmark and plugins specific to the omniORB broker had slightly over 1500 lines of code in total. Both plugins are generic and can be used in other experiments involving the same benchmark or broker. This represents a significant improvement over the ad–hoc scripts used to execute the Xampler benchmark, which do not support monitoring and analysis, cannot be used readily in other experiments, but currently already have 2000 lines of code.

The beta implementation of BEEN, including the Xampler module used for the evaluation, can be downloaded from the web [2]. Detailed step–by–step instructions to perform the evaluation, as well as screenshots of the user interface running the evaluation, are also available [2]. A more thorough evaluation will be in order as the BEEN implementation matures and is used for more benchmarks on more platforms.

7. CONCLUSIONS

Benchmarking is an essential performance evaluation technique, whose importance rests in that it provides performance data representative of a real system, which can be used either directly to assess performance or indirectly to calibrate and verify simulation and modeling results. Regular automated benchmarking is particularly popular because it gives software developers an important feedback on potential performance problems introduced during development. Automated benchmarking is, however, a complex process that comprises automated download, compilation, distributed deployment, monitoring, results collection, storage of results, evaluation and visualization. Ad–hoc benchmarking tools are commonly used for these tasks, even though most of them can be automated using a generic benchmarking tool.

BEEN is a generic benchmarking tool for automated benchmarking in a distributed heterogeneous environment, which supports comparison of system performance as well as regression benchmarking. The tool is designed to run any benchmark that provides raw performance data, with a minimal support required from the benchmark. BEEN currently focuses on the Linux, Windows and Solaris platforms, but can also work, in a limited mode, with other platforms that can run the Java Virtual Machine.

When comparing system performance, the whole benchmarking process is controlled by BEEN. This includes automated resolution of deadlocks and infinite loops in running benchmarks, storing a common form of performance results in a benchmark–independent results repository, and allowing both benchmark–independent and benchmark–specific evaluation of data. When comparing performance of consecutive software versions in regression benchmarking, BEEN automatically checks for new software versions, runs the benchmark, performs the automated detection of changes and schedules additional measurements as necessary to achieve the desired result precision.

In this paper, we describe the design and core functionality of BEEN. Although BEEN is still under development, the current beta version allows the comparison of system performance as shown on the example of the Xampler CORBA benchmark and the omniORB broker. The beta version is available on the web [2]. Future work will focus on finishing the implementation so that more types of analysis and more benchmarks on more platforms are readily supported.

8. ACKNOWLEDGMENTS

This work was partially supported by the Czech Academy of Sciences project 1ET400300504 and by the ITEA Osiris project.

9. REFERENCES

[1] Advanced Technology Labs. Agent and distributed objects quality of service. `http://www.atl.external.lmco.com/projects/QoS`, 2006.

[2] BEEN Developers. Benchmarking environment (BEEN). `http://nenya.ms.mff.cuni.cz/been`, 2006.

[3] L. Bulej, T. Kalibera, and P. Tuma. Repeated results analysis for middleware regression benchmarking. *Performance Evaluation*, 60(1–4):345–358, May 2005.

[4] M. Courson, A. Mink, G. Marçais, and B. Traverse. An automated benchmarking toolset. In *HPCN Europe*, volume 1823 of *LNCS*, pages 497–506. Springer, 2000.

[5] D. Grisby et al. Free high performance orb. `http://omniorb.sourceforge.net`, 2006.

[6] B. Dillenseger and E. Cecchet. CLIF is a Load Injection Framework. In *Workshop on Middleware Benchmarking: Approaches, Results, Experiences, OOPSLA 2003*, Oct. 2003.

[7] Distributed Systems Research Group. Mono regression benchmarking. `http://nenya.ms.mff.cuni.cz/projects/mono`, 2005.

[8] Distributed Systems Research Group. Comprehensive CORBA benchmarking. `http://nenya.ms.mff.cuni.cz/projects/corba/xampler.html`, 2006.

[9] DOC Group. TAO perf. scoreboard. `http://www.dre.vanderbilt.edu/stats/performance.shtml`, 2006.

[10] Free Software Foundation. The R project for statistical computing. `http://www.r-project.org`, 2006.

[11] T. Kalibera, L. Bulej, and P. Tuma. Generic environment for full automation of benchmarking. In *SOQUA/TECOS*, volume 58 of *LNI*, pages 125–132. GI, 2004.

[12] T. Kalibera, L. Bulej, and P. Tuma. Benchmark precision and random initial state. In *SPECTS 2005*, pages 853–862, San Diego, CA, USA, July 2005. SCS.

[13] T. Kalibera and P. Tuma. Precise regression benchmarking with random effects: Improving Mono benchmark results. In *Formal Methods and Stochastic Models for Performance Evaluation*, volume 4054 of *LNCS*, pages 63–77. Springer, June 2006.

[14] A. M. Memon, A. A. Porter, C. Yilmaz, A. Nagarajan, D. C. Schmidt, and B. Natarajan. Skoll: Distributed continuous quality assurance. In *ICSE*, pages 459–468. IEEE Computer Society, 2004.

[15] M. Prochazka, A. Madan, J. Vitek, and W. Liu. RTJBench: A Real-Time Java Benchmarking Framework. In *Component And Middleware Performance Workshop, OOPSLA 2004*, Oct. 2004.

[16] Supercomputer Computations Research Institute, Florida State University. Distributed queueing system. `http://packages.qa.debian.org/d/dqs.html`, 1998.

[17] University Corporation for Atmospheric Research. Network Common Data Form. `http://www.unidata.ucar.edu/software/netcdf`, 2006.

[18] C. Yilmaz, A. S. Krishna, A. M. Memon, A. A. Porter, D. C. Schmidt, A. S. Gokhale, and B. Natarajan. Main effects screening: a distributed continuous quality assurance process for monitoring performance degradation in evolving software systems. In *ICSE*, pages 293–302. ACM, 2005.