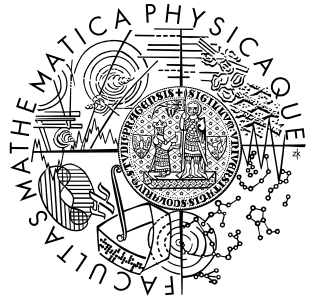
Charles University in Prague
Faculty of Mathematics and Physics

DOCTORAL THESIS



Tomáš Kalibera

Performance in Software Development Cycle: Regression Benchmarking

Department of Software Engineering
Advisor: Doc. Ing. Petr Tůma, Dr.

Abstract

Title: Performance in Software Development Cycle: Regression Benchmarking

Author: RNDr. Tomáš Kalibera
email: kalibera@nenya.ms.mff.cuni.cz
phone: +420 2 2191 4232

Department: Department of Software Engineering
Faculty of Mathematics and Physics
Charles University in Prague, Czech Republic

Advisor: Doc. Ing. Petr Tůma, Dr.
email: tuma@nenya.ms.mff.cuni.cz
phone: +420 2 2191 4267

Mailing address (both Author and Advisor):
Dept. of SW Engineering, Charles University in Prague
Malostranské nám. 25
118 00 Prague, Czech Republic

WWW: http://nenya.ms.mff.cuni.cz

This thesis: http://nenya.ms.mff.cuni.cz/~kalibera/phd-thesis

Abstract:

The development cycle of large software is necessarily prone to introducing software errors that are hard to find and fix. Automated regular testing (regression testing) is a popular method used to reduce the cost of finding and fixing functionality errors, but it neglects software performance.

The thesis focuses on performance errors, enabling automated detection of performance changes during software development (regression benchmarking). The key investigated problem is non-determinism in computer systems, which causes performance fluctuations. The problem is addressed by a novel benchmarking methodology based on statistical methods. The methodology is evaluated on a large open-source project Mono, detecting daily performance changes since August 2004, and on open-source CORBA implementations omniORB and TAO. The benchmark automation is a complex task in itself. As suggested by experience with compilation of weather forecast model Arpege/Aladin and implementation of component model SOFA, large systems place distinguishing demands on tasks such as automated compilation or execution. Complemented by experience from Mono benchmarking, the thesis proposes an architecture of a generic environment for automated regression benchmarking. The environment is being implemented by master students under supervision of the author of the thesis.

Keywords: regression benchmarking, non-determinism in performance

Acknowledgments

It is my pleasant obligation to thank all the people who have helped me with my research. I appreciate the help and counseling I received from my advisor Petr Tůma. My thanks also go to the head of the Distributed Systems Research Group, František Plášil, for his support and encouragement.

The thesis assumes the form of a collection of published papers. I would like to thank the co-authors of the included papers, my colleague Lubomír Bulej and my advisor Petr Tůma, for their collaboration, for sharing their experience from CORBA benchmarking projects, and for their consent with including the papers in the thesis. I am especially thankful to Lubomír Bulej for helping me at the beginning of my study, before his research focus shifted to component deployment.

I would like to thank Jaromír Antoch, Alena Koubková and Alena Koubková, jr., for their enduring advice on mathematical statistics and on methods potentially applicable to regression benchmarking. My special thanks go to my friend Tomáš Ostatnický, who has helped me with solving an integral that for long had been the missing bit in one of the proofs. I would also like to thank Michal Beneš for his help with mathematical analysis.

The thesis includes a design and architecture of a generic benchmarking environment. Such a design would be of a limited value without an implementation. I am grateful to master students Jakub Lehotský, David Majda, Branislav Repček, Michal Tomčányi, Antonín Tomeček, and Jaroslav Urban for implementing the environment under my demanding supervision. I would also like to thank them for their significant help with writing a paper about the environment design and for their consent with including the paper in my thesis.

I would like to thank Vladimír Mencl, Jiří Adámek and Tomáš Bureš for carefully proofreading some of the papers. My thanks also go to Martin Janoušek, Filip Váňa, Maria Derková and my other colleagues from Czech Hydrometeorological Institute for the unique opportunity: to get in touch with a huge real application, which is both developed and used on daily basis – the numerical weather forecast model Arpege/Aladin. I would like to thank Miguel de Icaza, Radek Doulík, Ben Mauer, and other Mono developers for their interest in the Mono Regression Benchmarking Project, their valuable comments, and support.

Contents

Chapter 1

Introduction

performance goals

Performance is an important factor of software quality. Performance requirements drive the design, implementation, and use of computer systems, where the most common objective is to get the highest possible performance for a given cost [16]. A closely related objective is to design a system with a given performance for a minimal cost (*capacity planning*).

perf. techniques

Typical tasks solved to achieve these objectives are: comparing performance of multiple systems, finding an optimal value of a system parameter (*system tuning*), finding a system component which is the performance bottleneck, characterizing the realistic workload of a production system, predicting performance for a given workload or predicting performance of a system built of components with known performance (*performance prediction*) [16].

perf. metrics

Software systems differ in the performance metrics of interest. Interactive systems are designed to minimize duration of a single operation or a user request (minimize the *latency* or *response time*). Batch systems are designed to process a maximum number of operations or user requests per second (maximize *throughput*) or to use the most expensive devices to the limit (maximize *utilization*). A complex system may consist of both an interactive and a batch sub-system. Additional low-level performance metrics, such as memory consumption, binary code size, network utilization, utilization of the garbage collector, or the duration of an application start-up, are of interest in specialized systems with limited system resources.

With the current trend of growing software size, complexity, and use [55], it is increasingly more important and more challenging to fulfill the performance requirements.

1.1 Performance in Software Development Cycle

The possible approaches to meeting software performance requirements depend on how the software is being developed. In this section, we discuss existing performance evaluation techniques in the context of the *software development process* (*software development cycle*).

There are many different models of the software development process. While the models differ in the ordering of the steps leading to a software product, they generally agree that the steps include design and implementation. Some models identify finer-grained steps, such as requirements analysis, verification and testing, but these steps can be understood as parts of the design and implementation. The software performance requirements influence both design and implementation.

Performance in Software Design

Design decisions that influence performance include the distribution of the system on different computers, remote communication, parallelization, locking granularity, and selection of algorithms and data structures. The performance evaluation methods appropriate for the design are *performance modeling* and *simulation*.

modeling

Performance modeling analytically evaluates parameters of a mathematical model of the software system of interest. The key analytical modeling technique is the *queueing theory* [16]. Its application to performance modeling is based on the common pattern in computer systems, where multiple requests are waiting in lines (*queues*) to be served or to get access to a system resource.

simulation

Simulation methods can be used for substituting not-yet-available parts of the system by stubs that have performance characteristics similar to the real system parts. Different algorithms can be prototyped and evaluated using the simulated system parts, and the best algorithm with respect to the performance requirements can be chosen. The simulation methods rely heavily on good knowledge of the simulated system. Realistic simulations of complex systems can be by orders of magnitude slower than the real systems.

Performance in Software Implementation

There is no clear distinction between design and implementation. The decisions that influence performance at design level gradually merge into technical

decisions at the implementation level, such as selecting types for variables, selecting compiler optimizations or advising the compiler how to optimize different parts of the code. The optimizations, both manual and automated by the compiler, can be targeted at better processor data- or instruction-cache utilization, better performance of the branch prediction unit, better performance of simultaneous multi-threading, or on other aspects, depending on a given platform. Both today's hardware and software are very complex and their internal specifications on the level of detail that would enable precise optimization are often not available to the developers. Predicting the performance implications of the listed decisions on optimizations is therefore hard or even impossible. The decisions are thus based on experiments with different implementations. The basic experimental methods are *performance profiling* and *benchmarking*.

benchmarking

Benchmarking is an experimental performance evaluation technique that attempts to estimate performance of a system under a realistic workload. The workload is usually generated by a probabilistic model, but a real workload snapshot can be replayed if available. The evaluated system is exercised by a specialized application called a *benchmark*, which should be designed to closely mimic a real usage scenario of the system. Benchmarking can answer the basic question, whether the system fulfills its performance requirements. The answer has to be backed statistically, because of frequent performance fluctuations in current systems.

When the system does not fulfill the performance requirements, its parameters or implementation have to be modified. The hard part is to find the optimal parameter changes (system tuning) or implementation modifications that would improve performance. The optimal values of parameters can be found experimentally by running a benchmark for each combination of the possible values. If the number of combinations is too large, and the parameters are not inter-dependent, the optimum can be found without testing all possible combinations. Selecting the combinations of parameters and their values to test by a benchmark can be based on *experimental design theory* [16].

profiling

Profiling helps to find out which parts of the implementation should be modified to improve system performance. Profiling is similar to benchmarking, except that it monitors the behavior of the running system and collects as much performance data as possible. By collecting big amounts of data, the profiler significantly slows down the execution of the system, and thus the performance results are incorrect in absolute values. The behavior information, such as call trace or lock contention, attached with approximate performance data, however, helps the developers to find out which parts of

the implementation use most resources. Improving performance of the identified parts (if possible) is an efficient way to improve the performance of the whole system. If the identified parts cannot be improved, the investigation moves on system parts that are using less system resources.

Performance in Development Models

Although the design and implementation steps are typically present in the development of a software system, the design does not always precede the implementation.

common models

The typical development process models where the design does precede the implementation include derivatives of the *sequential model*, described and criticized in [21, 22]. In the sequential model, each step is done only once, in the following order: requirements analysis, specification, design, and implementation. This model is believed to be feasible for mission critical applications with design well known in advance, and it is believed to allow realistic planning of software product milestones in advance [59, 58]. The model is used for U.S. Air Force and NASA software development [58].

In [21], the sequential model is criticized for neglecting that some design problems can be discovered only during implementation or testing, as well as for neglecting inconsistencies in requirements found at design time. A modified model is proposed that allows this feedback [21][1]. The approach of designing as much as possible before implementing is also advocated and supported by a case study in [56]. The model is further extended to the *spiral model*, which allows an incremental creation of parts of the application [22]. Each iteration is modeled by the sequential model with feedback propagation.

The performance evaluation of the design can be especially useful in these models, where the implementation is not available at the design time. Similar to fixing functionality problems, fixing a performance problem at design time is cheaper than fixing it after the implementation is finished. Unfortunately, some design performance problems may be very hard to find at the design time. This is the case of performance problems caused by hardware and software with too complex or unknown internal behavior. Such a behavior obviously cannot be modeled or simulated by the developers, but may have design implications. For instance, the performance of applications that are memory and computation intensive heavily depends on CPU memory cache performance [8]. Structuring the code of such applications into functions or methods can affect the cache performance in a way impossible to predict. The

[1] Both the proposed model and the original sequential model are sometimes called "waterfall" models. We do not use the term "waterfall" to avoid confusion.

performance requirements, more than the functional ones, therefore suggest the use of a development process model that allows incorporating the feedback from the implementation into the design.

agile models

The *agile development process models* [14, 20] are built on this feedback from the outset. In these models, the design is created incrementally based on the feedback from the implementation. The most widely known of these models is the *extreme programming model* [14]. The extreme programming model is iterative, where each iteration is formed by coding, testing and design. In each iteration, only minimal changes are made to the code. A key technique of the model is automated regular testing, which not only helps to discover errors, but also partially supplies the design – a test implements what the tested code part should do. Despite the model being relatively new, there are several published case studies [23, 24, 25]. More case studies are referenced in [23]. In extreme programming, the major performance evaluation techniques would be benchmarking and profiling. The simulation and modeling techniques are of lower importance, unless performance prediction or system dimensioning are required.

1.2 Continuity: Development Cycle in Broader View

Despite the increasing popularity of agile programming, prevailing development practices still largely assume designing before implementing. The automated regular testing, propagated by the agile models, is, however, widely used, especially in large and continuously developed systems. In this section, we emphasize the role of regular testing in continuous software development, and argue for extending the testing to cover performance.

live systems

In live and large software systems (e.g. a with million or more lines of code), requirements change frequently. The changes of requirements are caused by new scientific results, new potential of the hardware, social changes, or discontinued support for hardware or software the system depends on.

The incorporation of such new requirements into a system can be achieved either through a complete re-design and re-implementation of the system, or through adapting both the design and the implementation of the existing system. The first approach is very expensive and time-consuming, but due to its nature allows even very large changes, and is believed to be less prone to errors. This approach is used for mission-critical applications, for example in NASA [59]. The approach of adapting the existing system is more common, yielding a continuous development of a single system for tens of years. The

continuous development

continuous development is, however, technically challenging and potentially prone to introducing errors.

The technical challenges are caused by the extent of the code base, a large and geographically distributed team of developers, and by dependencies on dated technologies from earlier development of the project. There are numerous specialized tools that help to face these challenges, such as incremental or modular build systems for large code, versioning systems, hyper-linked source browsers, and debuggers.

source dependencies

In this context, an incremental and modular source dependency tool was designed and implemented for the Arpege/Aladin numerical forecast model. This tool is used by Arpege/Aladin developers in Czech Hydrometeorological Institute, Prague, and is integrated into its versioning and build system. The tool is described in [5], included in Section 10. The main advantage of the tool is that it is scalable enough to support a project with a million lines of code (which is the case of Arpege/Aladin) and complex inter-dependencies among source files.

incorporating errors

In the continuous development, gradual implementation changes in ways not foreseen by the original design, as well as the mentioned technical challenges, necessarily end up importing errors into the code. This trend includes not only functional errors, which result in the application failing to do what it was designed for, but also *performance regressions*, which are (avoidable) degradations of performance. This problem can be approached both by prevention – improving design methods to be more flexible, and by automated testing – detecting the errors as soon as possible and thus making them cheaper to fix.

Designing for Continuity

component design

One of the design methods that aim at making software more flexible and reusable is component design. In component design, systems are fully or partially built of inter-connected components that have to explicitly declare how they communicate. The components not only declare the functionality they provide, but also declare the functionality they require. The component design makes an application easier to adapt, because of the explicit declaration of inter-dependencies and explicit architecture. Moreover, replacing a component can be supported at run-time, components can be distributed transparently on different computers, and the behavior of a component can be described and checked statically or at run-time [82, 83, 84]. References to different component models can be found in [82].

There is no strict definition of a single generally accepted component

model. Thus many component models exist that do not have all of the features described above. Current systems can still benefit from the component design ideas, being implemented either ad-hoc or using an infrastructure of some of the available component models.

SOFA comp. model

In this context, support for SOFA component model in the C++ environment was designed and implemented. The prototype implementation is available on-line [11], and the experience was published in [10], included in Section 11. The experience report focuses in detail on the limitations of static component architectures and of run-time CORBA interoperability.

Testing for Continuity

Neither the component design nor other design techniques may fully prevent incorporating errors into a continuously developed system. It is well known that early locating of errors makes their fixing cheaper. Manual testing after each source code modification would however not be feasible.

regression testing

The functionality tests are therefore fully automated [53, 15]. The tests are either run daily, usually at night when the developers do not commit new changes, or immediately after each commit. The tests are also called *regression tests*, since their purpose is to detect regressions in software quality. They include *unit tests*, which test functionality and robustness of small and isolated parts of code on synthetic data, and *system tests*, which test functionality of the whole system. The regression tests are written by the source code developers and/or by dedicated developers responsible for software quality. The operation of the testing is fully automated, up to notification of the authors of the failing code. The automation necessarily includes regular builds, the developers are thus also informed when they break the very ability to build the system. The regular builds themselves are considered a useful technique that simplifies testing and locating of errors [57].

performance testing

Incorporating detection of performance problems into regression testing seems to be a logical step, but it is surprisingly difficult. The basic problem is the definition of a performance problem. In terms of functionality, the objective is to have no error for any workload or input data. The *hard real-time systems* may have strict performance requirements on any workload or input, and then any performance test failing to meet these requirements is an evidence of a performance problem.

regression benchmarking

The performance objectives are not always that clear in *soft real-time systems* or *non real-time systems*. Often, getting the highest possible performance for a given cost is the objective [16]. In these situations, any avoidable degradation of performance (*performance regression*) is considered a perfor-

mance error. It is unlikely that any automated testing could detect that a performance degradation is avoidable, thus the testing can only focus on detecting all performance changes. Such performance testing is based on regular automated benchmarking and automated detection of changes in the benchmark results (*regression benchmarking*). The regression benchmarking is used or planned to be used by many systems and vendors, as described in detail in Section 13. However, the automation support of running the benchmarks is usually created ad-hoc for each system, and the performance changes are not detected automatically.

automation of benchmarking

The automated running of benchmarks, which is necessary for regression benchmarking, is more challenging than automated running of functional tests. There are two distinguishing problems. In order to minimize interferences, the benchmarks have to be run in an isolated environment and on a stable platform. Moreover, the random nature of current systems makes it necessary to repeat each benchmark execution, based on statistical characteristics of the results.

The automated detection of performance changes is hard because of the random fluctuations in current systems. These fluctuations might easily be mistaken for performance changes. Last but not least, an important task is to match performance regressions in the system to modifications of the system's source code. For a wide adoption of regression benchmarking into software development processes, tools that would help the developers with these tasks are required.

1.3 Goal of the Thesis

The goal of the thesis is to realize the concept of regression benchmarking and to evaluate its applicability on a real, live and large software project. The main problem is to devise a methodology for running benchmarks and to automatically detect performance changes based on benchmark results. The methodology should be reliable – it should not produce too many false notifications of changes, it should be benchmark- and system-independent, and the whole regression detection process should be automated as much as possible.

1.4 Structure of the Thesis

The thesis, being a collection of published papers with an unifying text, is structured as follows. Section 2 gives an introduction to regression bench-

marking with references to the included papers; the papers are included in Sections 3–11. Section 13 provides a detailed overview of the projects related to regression benchmarking and to the problems solved in the thesis. The contribution of the thesis is summarized in detail in Section 12. The thesis is briefly concluded and evaluated in Section 14, with an outlook to potential future work.

Chapter 2

Regression Benchmarking

Compared to functionality, performance is a slightly overlooked aspect of software quality. Functionality can be defined and checked more easily, it can be even specified formally and the code proven automatically to follow the specification, programming models are built on functionality testing [14, 15], and automated functionality testing frameworks are available [53]. However, especially large and continuously-developed projects are prone to incorporation of not only functional errors, but also performance regressions. The performance regressions can have the form of slowing down the system or increasing the use of other resources, such as memory or network capacity, and may be very hard to find.

regression benchmarking

As a response to this problem, the Distributed Systems Research Group introduced the concept of regression benchmarking [79]. The main objective of regression benchmarking is to automatically detect performance regressions during software development. The additional objective is to help developers with finding modifications of the source code that have caused the regressions. The basic idea of regression benchmarking is very simple – a stable set of benchmarks is run regularly, i.e. on daily versions of the tested software system. The benchmark results from consecutive software versions are then compared and a significant increase in response time (or in use of a system resource) is reported as a performance regression. As a consequence, regression benchmarking can easily be used to detect performance improvements, and to monitor software performance during development.

In the last two years, a demand for monitoring performance during software development has been formulated by many software vendors or project managers, for systems such as the Linux kernel, GNU Compiler Collection, Open Office (Sun Microsystems), and Solaris (Sun Microsystems). The currently performed regression benchmarking, however, lacks the automated de-

tection of changes. [1]

This section follows by an analysis of the automated detection of performance changes, automated running of benchmarks, and locating source code modifications that are causing the performance changes.

2.1 Automated Analysis of Benchmark Results

The objective of the automated analysis of benchmark results for regression benchmarking is to reliably detect performance changes. An exact detection of performance changes is not possible due to non-determinism in current computer systems, both in hardware and software. The objective therefore is a detection of changes with a small number of *false alarms* (reported changes when there is no change), and with a small number of undetected changes. To meet the objective, the benchmarks have to be well designed and their results analyzed by statistically sound methods.

Benchmark Design

Designing a good benchmark is itself a technically challenging task. The Distributed Systems Research Group has been involved in several CORBA benchmarking projects [80, 81], and based on this experience, typical problems of benchmark design were published [78]. These problems are general in nature, and thus also applicable to systems other than CORBA.

benchmark warm-up

One of the problems that is not surprising, but still is broadly overlooked, is the warming-up of software systems. It is common that performance of a software system changes abruptly at some moment after the benchmark is started.

The problem is caused by various initializations (*warm-up*). The technical reasons for and the duration of the warm-up period are system- and benchmark-dependent. Although performance of an already-warmed-up system is usually of interest, benchmarks often measure the performance during the warm-up period, either estimating the duration of the warm-up period incorrectly or completely ignoring the warm-up period.

Estimating the duration of the warm-up period is a complex task. Although there are many relevant statistical methods (a good list of references can be found in [26]), they are not readily applicable on general computer

[1] A detailed overview of existing regression benchmarking projects is provided in Section 13.

systems. The duration of a benchmark warm-up period on a given computer system can be estimated experimentally: the benchmark is executed several times to collect many observations, the results are visualized, and the upper bound for the duration of the warm-up period is determined manually.

separating measurement and analysis

Benchmarks should be designed not to interfere with the system they measure, and therefore the measurement code should be simple and non-intrusive. In particular, the code should not analyze the results, but it should report them in a raw format. Such a design also allows re-using the results for different types of analysis. In particular, it allows the described experimental detection of the duration of the warm-up period to be performed off-line.

workload complexity

Any benchmark, be it used for regression benchmarking or not, should be designed to measure performance that is relevant to a real system. The relevancy of results largely depends on how realistic is the applied system workload. Generating a realistic workload for a nontrivial system may be a complex task and may require a complex infrastructure, both for the large number of requests per second and for concurrency of the requests. [2] In respect to workload, we identify two distinct benchmark designs in [3], included in Section 3: a *complex benchmark*, which uses the realistic workload, and *a simple benchmark*, which uses a synthetic workload with minimized interferences.

A complex benchmark uses a realistic workload to provide results very close to a real use of the evaluated system. It tests most of the evaluated system, but its execution is expensive. A simple benchmark, on the other hand, measures performance of a small part of the system under a trivial synthetic workload, in which a single system operation is invoked repeatedly. Simple benchmarks are cheaper to execute than complex benchmarks, provide performance information on selected parts of the system, but do not test performance of the interplay of the system parts, such as parallelization, lock contention, etc. In [3], we show that the results from simple and complex benchmarks require different evaluation methods.

non-determinism in performance

The simple benchmarks suffer less from random fluctuations in results than the complex benchmarks, because they test a small code part (a single operation) in an isolated environment. The low result fluctuation allows detecting even small changes in performance. The problem described first in [3] is non-determinism in benchmark execution that influences the measured operation performance, in particular the operation response time. The

[2] For example, according to [64], up to 5 client machines may be needed to generate a realistic workload for an application server.

non-determinism can result in much larger variation in response times of operations measured in different benchmark executions, compared to relatively small variation in response times of operations measured in the same execution. This effect largely depends on the benchmark: in some benchmarks, there is a difference in several orders of magnitude, in some there is no difference. Ignoring this effect would easily lead to false alarms when detecting changes, because performance of the same software is, due to this effect, different in each benchmark execution.

performance over-shadowing

The complex benchmarks measure performance of a heterogeneous set of operations forming a realistic system workload. As a consequence, performance changes in some parts of the system may easily be over-shadowed by absolute performance of the parts of the system that dominate the workload. The over-shadowed changes may still be important, but their discovery by simple benchmarks may be too expensive in large systems, because too many benchmarks would have to be run; the time available for regular benchmarking is usually limited. In addition, identifying all parts of the system that would require a dedicated benchmark would be a hard task. The performance changes that appear only in interplay with other parts of the system cannot be detected by simple benchmarks at all. In [3], we show that the over-shadowed performance changes can sometimes be distilled from the results of complex benchmarks using cluster analysis of the results.

Change Detection in Face of Non-determinism

non-determinism in execution

The problem of non-determinism in benchmark execution is further evaluated in [8], included in Section 4. It is shown that the problem exists on different hardware platforms, different operating systems and in different benchmarks, and it affects performance to a varying degree. The paper defines a metric which allows quantification of this effect and compares different systems using this metric. The cause of this effect is tracked down by analyzing the information collected from hardware performance monitoring counters. Among other things, the non-determinism in performance is caused by random selection of physical memory addresses for code and data allocated by the operating system at process start-up. The actual selection of physical memory addresses predetermines the number of memory cache misses during benchmark execution, and thus the response time of the measured operation.

non-determinism in compilation

The paper identifies yet another source of non-determinism: the compilation. It is shown that compilation may be intentionally randomized by the compiler, and that different binaries created from the same source code may have different performance. The reason of the impact on performance is sim-

ilar to the case of benchmark execution: the different binaries have different memory layout, and thus are subject to different numbers of cache misses during execution. Therefore, not only different executions, but also different binaries of the same software may differ in performance.

living with non-determinism

Both sources of non-determinism are inherent to current computer systems, may have significant effect on performance, and thus cannot be overlooked in regression benchmarking. It is often technically impossible to remove this non-determinism. Moreover, performance results with artificially removed non-determinism would not be realistic. The problem can be solved by repeating compilations and executions, and statistically evaluating the data. The non-determinism can be statistically modeled by random effects, and parameters of such model can be estimated based on experiments. The challenge is to devise a model that will cover the widest possible range of benchmarks, but will still allow reliable detection of performance changes.

statistical model of non-determinism

We did not find any statistical model that could readily be used for modeling the non-determinism in performance of computer systems. The closest model is the random effects model in one way classification [13]. [3] In [1], included in Section 6, we describe a new statistical model of benchmark performance that incorporates the non-determinism in benchmark execution. The model allows detection of changes in the mean of a performance metric, typically a response time. It also allows determining the optimum number of benchmark executions and operations in each execution so that the best results precision is reached within a given time. The more precise are the results, the more reliable and powerful is the change detection.

Mono benchmarking

The model is evaluated in the Mono Regression Benchmarking Project [6], which automatically performs regression benchmarking of Mono [70], from downloading sources to reporting detected performance changes. The results are available on the web [6], covering daily Mono versions since August 2004. Selected performance changes from the discovered ones are experimentally verified as not being false alarms – the true cause of the changes is located in the sources. Verifying all the detected changes would, however, require too much effort and very good knowledge of the sources.

Mono

The Mono project was chosen as it is open-source, has a large code base (over 2 millions lines of code), over 70 developers and is being actively worked on. The project implements the standard .NET platform [65], including the C# language compiler, the virtual machine (Common Language Runtime) and the standard class libraries.

[3] The related statistical models are mentioned in Section 13.

non-determinism in compilation

The problem of non-determinism in compilation, not covered by the statistical model in [1], is further evaluated in [9], included in Section 7. The experiments with the Mono benchmarks used in the Mono Regression Benchmarking Project, and the results from CORBA benchmarks, show that ignoring the non-determinism in compilation can significantly increase the number of false alarms in regression benchmarking.

The paper includes a new statistical model, which incorporates the non-determinism both in benchmark execution and in compilation. The model assumes that both benchmark compilation and benchmark execution is repeated, and it allows statistical detection of changes. The model makes it possible to determine optimum numbers of compilations, executions and operations that maximize the precision of results reached in a given time.

The evaluation of the model is based on the same Mono and CORBA benchmarks on which the problem was demonstrated. The evaluation shows that when used for regression benchmarking, the model and the repetitions of the compilations help to dramatically cut down the number of false alarms. The model is also applicable to benchmarking in general, when precise results are needed. Regression benchmarking is, with respect to the model, only an application of benchmarking, which is indeed very sensitive to the precision of results.

2.2 Locating Causes of Performance Changes

For regular use of regression benchmarking in software development, it is very important not only to minimize the chances of false alarms, but also to help developers with finding the modifications of source code that have caused the detected performance regressions.

locating in general

The modifications can be located partially with the support from a versioning system. The versioning system can reveal a list of modified source files between two software versions, and sometimes can group these modifications into logical units. The logical grouping may help the developers to faster identify the modifications causing the performance changes. Still, most of the executed benchmarks would likely not test all the code of the evaluated system, and thus not all code from the modifications, either. The code actually covered by a specific benchmark can be discovered by run-time behavior analysis of the benchmark and/or by off-line static analysis of source code. The various options, in the context of Mono benchmarking, are analyzed in [2], included in Section 5.

locating in Mono

The Mono Regression Benchmarking Project links each detected performance change to a list of source files modified between the compared Mono versions. The list of modified files is further restricted to the files possibly used by a specific benchmark. These files are found approximately, using a run-time analysis of method invocations supported by Mono, and by a simplified static analysis of source files. The run-time analysis is performed in a dedicated execution of the benchmark, because it significantly slows down the benchmark execution. The analysis only covers modifications in the Mono libraries. Automated mapping of performance changes to modifications of the C# compiler itself, or to modifications of the virtual machine implementation, does not seem to be realistically achievable.

2.3 Automated Running of Benchmarks

automation requirements

Regression benchmarking is based on fully automated running of benchmarks. The automation has to include all the necessary steps from code checkout and download through software compilation, benchmark compilation, deployment, execution and monitoring, and data collection to analysis and presentation. Although many of these steps are already present in functional regression testing, automation of benchmarking brings several unique problems. Benchmarks should be run without any interfering applications, benchmarks of consecutive software versions should be run with identical configurations, and non-deterministically failing benchmarks should be restarted.

In our experience with the Mono project, benchmarks of software under development may crash non-deterministically – some executions of the same benchmark are successful and some fail. When the failures are rare, the execution environment should automatically restart the failing benchmarks several times, to get a required number of successful executions. There are even some cases when a part of a benchmark fails, but the results are still relevant. For example a client emulator of the Rubis benchmark [54, 63] sometimes hangs during shutdown. The problem is caused by limited Java interoperability with the underlying operating system, and cannot be fixed easily. Any benchmark execution environment should be extensible enough to overcome different types of failures, including those that do not have impact on the results.

A benchmark execution environment should be distributed, because it needs to be able to test distributed systems, and because multiple hosts may be required to generate workload that would saturate a tested server. In addition, the distribution may be required for parallelizing the software compilation and benchmark execution. In regression benchmarking of large software

systems with many benchmarks, the parallelization may become necessary to get results from each system version on-time. With the requirement of running benchmarks on consecutive software versions on identical configurations, the assignment of different tasks to hosts may be complicated: a typical requirement would be to adaptively assign available hosts for compilation of a benchmark, but run the benchmark on a fixed set of hosts.

More requirements on a benchmark-independent environment for automated benchmarking are analyzed in [4], included in Section 8. The paper confronts these requirements with the existing projects that use automated benchmarking, and proposes a high-level architecture of a generic environment for full automation of benchmarking, with support for regression benchmarking.

automation design

The environment is being implemented in a student project BEEN [7] under the supervision of the author of the thesis. Based on experience with an early beta version of the environment, the architecture was detailed and published in [12], included in Section 9. The beta version of the environment is mature enough to validate the usefulness of the proposed architecture. In particular, it supports automated benchmarking with a distributed CORBA benchmark [80] targeted at comparing performance of different versions of omniORB [71]. The environment currently has over 65,000 lines of code.

Chapter 3

Repeated Results Analysis for Middleware Regression Benchmarking

Lubomír Bulej,
Tomáš Kalibera,
Petr Tůma

Contributed paper in **Performance Evaluation** [3].

In *Performance Evaluation: An International Journal, Performance Modeling and Evaluation of High-Performance Parallel and Distributed Systems*,
published by Elsevier B.V.,
vol. 60,
pages 345–358,
ISSN 0166-5316,
May 2005.

The original version is available electronically from the publisher's site at http://dx.doi.org/10.1016/j.peva.2004.10.013.

Repeated Results Analysis for Middleware Regression Benchmarking

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

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

Abstract: The paper outlines the concept of regression benchmarking as a variant of regression testing focused at detecting performance regressions. Applying the regression benchmarking in the area of middleware development, the paper explains how the regression benchmarking differs from middleware benchmarking in general, and shows on real-world examples why the existing benchmarks do not give results sufficient for regression benchmarking. Considering two broad groups of benchmarks based on their complexity, novel techniques are proposed for the repeated analysis of results for the purpose of detecting performance regressions.

Keywords: middleware benchmarking, regression benchmarking, regression testing

1. Introduction

The development and release process of a software system is typically subject to a demand for certain level of quality assurance. One of the approaches to meet this demand is regression testing, where a suite of tests is built into the software system so that it can be regularly tested and potential regressions in its functionality detected and fixed. Regression testing has many potential uses. Applied to the source code of a software system, it verifies the syntactical correctness of the tested code across the range of supported platforms. Applied to the running software system or its parts, it helps guarantee correct functionality of the tested code.

Regression testing is becoming an integral part of the development and release process of communication and application middleware. The complexity of the middleware has led many middleware projects to adopt some form of regression testing, as evidenced by open source middleware projects such as CAROL [19], OpenORB [17], or TAO [16] with its distributed

scoreboard [15], which collects results of daily build and test runs on various platforms from across the world. Focusing on functionality, however, the regression testing of middleware tends to neglect the performance aspect of quality assurance, which is typically orthogonal to correct functionality and thus seen as a minor factor in quality assurance. This contrasts with the otherwise common use of middleware performance evaluation to satisfy the obvious need to evaluate and compare performance of numerous implementations of communication and application middleware standards, such as CORBA [18], EJB [22], or RMI [23].

To remedy the existing neglect of the performance aspect in regression testing, we focus on incorporating middleware performance evaluation into regression testing. Our experience from a series of middleware performance evaluation and comparison projects [13][14][10][11] shows that systematic benchmarking of middleware can reveal performance bottlenecks and design problems as well as implementation errors. This leads us to believe that detailed, extensive and repetitive benchmarking can be used for finding performance regressions in middleware, thus improving the overall process of quality assurance. For obvious reasons, we refer to such middleware performance evaluation as regression benchmarking.

In section 2 of the paper, we investigate in more depth the concept of regression benchmarking, explaining why and how it differs from benchmarking in general. Dividing the existing benchmarks into two broad groups based on their complexity, sections 3 and 4 discuss the suitability of the two groups for regression benchmarking and propose techniques for the repeated analysis of results for the purpose of detecting performance regressions. Section 5 concludes the paper.

Throughout the paper, we use TAO [16] and omniORB [12] as real world examples of a complex and mature open-source middleware to illustrate the individual points and proposed techniques.[1] Illustrating the points and techniques on commercial middleware coming from a closed-source vendor would be difficult because the development practices of such a vendor are not public and we would be limited to a user experience with the middleware. Nevertheless, our past middleware performance evaluation and comparison projects have revealed several performance problems in commercial middleware, ranging from minor performance flaws to major scalability issues. These problems would probably have been found had the middleware been subjected to regression benchmarking. From this, we conclude that regression benchmarking also has a valid application in the commercial sphere.

2. Regression Benchmarking

Regression benchmarking is a special application of benchmarking for software regression testing. As such, it has to be tightly integrated with the development process, fully automated, comprehensive and repetitive. Architecture of an environment for regression benchmarking is outlined in figure 1.

[1] The TAO examples use TAO 1.3.1 to 1.3.5 on Pentium M 1.3GHz, 512MB RAM, Linux 2.4.22, GCC 3.3.2. The omniORB examples use omniORB 4.0.3 on Dual Pentium Xeon 2.2GHz, 512MB RAM, Linux 2.4.22, GCC 3.3.2.

In the outlined architecture, the regression benchmarking is started by the control module, responsible for executing all the configured benchmark modules. To keep the benchmark modules as small and simple as possible, the common functionality of the benchmarks is factored into the benchmark framework that supports the modules. The results collected by the benchmark modules are stored in the results repository, forming a history of results. The analysis module examines the history of results and detects performance regressions. To avoid repetitive or redundant notifications, the analysis module consults the notification repository for a history of notifications.

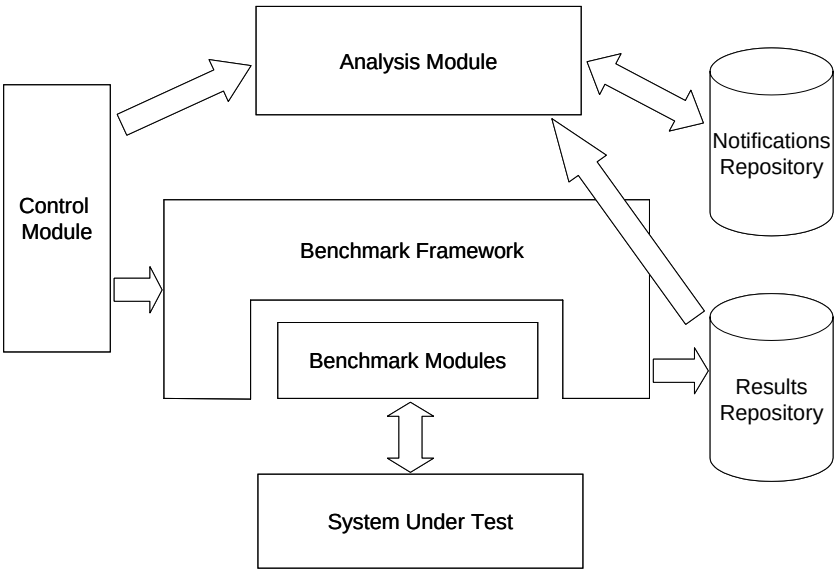


Figure 1: The architecture of an environment for regression benchmarking.

The nature of regression benchmarking makes it different from middleware benchmarking in general in the areas of benchmark integration, benchmark automation, result precision and result interpretation.

2.1. Benchmark Integration

The regression benchmarks must be comprehensive in how they cover the functionality provided by the middleware. This is best achieved by integrating the benchmark framework with the middleware so that new benchmark modules can be added alongside new middleware features. The integration minimizes the cost of creating and maintaining benchmark modules.[2]

[2] It is not without interest that the integration fits well with the very similar guidelines for unit testing and acceptance testing from the extreme programming methodology [1].

The integration of the benchmark framework with the middleware has the added benefit of the benchmarks supporting the same platforms as the middleware. Unlike middleware benchmarking in general, the portability of the benchmarks between middleware platforms is less of an issue with regression benchmarking. For a real-world example of the difference, consider the benchmark suite [13], which has 2500 platform-independent and 13500 platform-dependent lines of code to support 12 middleware implementations on 3 operating systems, or the benchmark suite [14], which has 6000 platform-independent and 8000 platform-dependent lines of code to support 4 middleware implementations on 3 operating systems.

2.2. Benchmark Automation

The regression benchmarks must be fully automated so that they can run unattended. The requirement of automation concerns not only the execution of the benchmarks, but also the data acquisition and the results analysis. Of these three tasks, automated execution is the simplest, with the existing remote access and scripting mechanisms being well up to the task.

The automated data acquisition must be able to recognize when the regression benchmark outputs stable data as opposed to data distorted during the warm up period of the benchmark. Middleware benchmarking in general either uses long warm up periods or expects the warm up periods to be set by trial and error, neither of which is acceptable for regression benchmarking.

Another problem associated with the automated data acquisition is the need to collect and store large amounts of data without interference with the benchmark. For a real-world example, consider the benchmark suite [14], which generates about 80 megabytes of data in a single run. A rough estimate of an individual observation consisting of the measured operation duration accompanied by annotations such as resource usage data and relevant event lists yields tens of bytes per observation, which in turn yields tens of kilobytes per second for one observation per millisecond.

2.3. Result Precision

The regression benchmarks must detect performance regressions as early as possible. The longer the period between the occurrence and detection of a performance regression, the more difficult it is to find the source of the regression and the more costly it is to fix the regression. The requirement for early detection implies a need for benchmarks that are so short they can be run daily and so precise they can detect minuscule changes in performance. This is especially a problem for creeping performance degradations, which consist of a sequence of individually negligible changes over a long period of time.

2.4. Result Interpretation

The results of the regression benchmarks are interesting in how they change rather than in what absolute values they have. This means that compared to middleware benchmarking in general,

tuning for maximum performance and comparing maximum performance across platforms is less of an issue.

The changes in the results of the regression benchmarks can have many causes, ranging from random fluctuations through effects of inadvertent configuration changes to true performance regressions. The automated interpretation must be able to distinguish these causes reliably to minimize the number of both false positive interpretations and false negative interpretations.

3. Simple Benchmarks

In the paper, we consider the suitability of the existing benchmarks for two broad groups of benchmarks based on their complexity. The group of simple benchmarks covers benchmarks such as [13][14][15][6][8], where an isolated feature of the middleware is tested under artificial workload. The intuitive justification for the simple benchmarks is that they provide little space for interference and thus yield precise results with straightforward interpretation.

A real-world example of a simple middleware benchmark is a remote method invocation benchmark that measures the duration of an isolated remote method invocation. The results of such benchmarks in [13][14][15] suggest that individual runs of a simple benchmark typically yield results that differ in units of percents. Using such results for regression benchmarking would imply a need to ignore these differences and only identify differences of tens of percents as performance regressions. Such a precision is clearly too low.

3.1. Minimizing Interference

The difference in the results of individual runs of a simple benchmark can be partially attributed to interference from the operating system, consisting especially of involuntary context switches and device interrupts.

One way of minimizing this interference is keeping the measured operation duration below the period of the interference and thus making the probability of interference during the measured operation reasonably small. In our example, this means measuring the low-level operations that form the remote method invocation, such as the marshaling and unmarshaling operations, data conversion operations and dispatching in various stages of the invocation, rather than the entire remote method invocation. The durations of the low-level operations range from tens to hundreds of microseconds, which is well below the period of the operating system interference, ranging from tens to hundreds of milliseconds.

It is also possible to mitigate the impact of the interference on the results by expressing the results using robust estimators that are not affected by a small number of exceptional observations. In our example, this means using the median of the operation duration rather than the average.

3.2. Collecting Observations

Another reason for the difference in the results of individual runs of a simple benchmark can be an insufficient number of observations collected by each individual run. When estimating the median of the operation duration, we assume the observed durations to be independent identically distributed observations and estimate the median using order statistics. To determine the minimal number of observations necessary to ensure a precise estimate of the median, we employ a quantile precision requirement proposed by Chen and Kelton in [4], which uses a dimensionless maximum proportion confidence half-width instead of the usual maximum absolute or relative confidence half-width. We then determine the required sample size n_p for fixed-sample-size procedure of estimating the p quantile of an independent identically distributed sequence using the formula proposed in [4]:

$$n_p \geq \frac{z^2_{1-\frac{\alpha}{2}} \cdot p \cdot (1-p)}{(\varepsilon')^2}$$

where $z^2_{1-\frac{\alpha}{2}}$ is the $1-\frac{\alpha}{2}$ quantile of the normal distribution, ε' is the maximum proportion half-width of the confidence interval, and $1-\alpha$ is the confidence level.

For a 95% confidence that the median estimator has no more than $\varepsilon' = 0.005$ deviation from the true but unknown median, we need to collect at least $n_p = 38416$ observations. For our experiments, we choose $n_p = 65536$, for which the confidence level borders with 99%.[3]

3.3. Comparing Results

Even after minimizing the interference and collecting the necessary number of observations, the individual runs of a simple benchmark yield different results. A real-world example of a simple middleware benchmark that minimizes the interference by measuring the duration of marshalling as a low-level operation and uses 65536 observations to estimate the median of the operation duration is shown in figure 2.

The differences in figure 2 suggest that we do not have enough control over the initial state of the system to make the results repeatable across runs, even for a very simple middleware benchmark. This prevents a direct comparison of results from individual runs. Figure 3 illustrates this problem on the results of 10 benchmark runs for two subsequent releases of TAO.

[3] We have also considered other ways of determining the required number of observations, described in detail in [3].

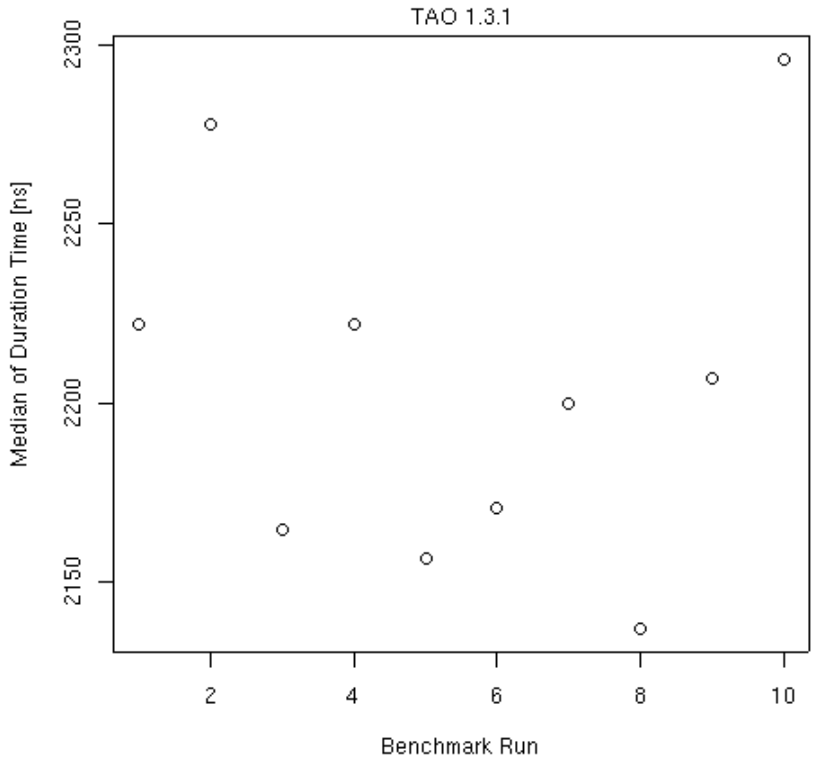


Figure 2: Results of consecutive runs of a benchmark measuring the time to marshal an input array of 1024 CORBA::ULong values on TAO 1.3.1.

Although the results in figure 3 indicate that the two subsequent releases of TAO deliver different performance, comparing results for the two releases using only a single run for each release would be potentially incorrect. To compare the results correctly, we have to take into account their random character, and treat each result of an individual benchmark run as an observation of a random variable.

We can assume the results of several consecutive benchmark runs to be a sequence of independent identically distributed observations of a random variable. The assumption of independency and identical distribution can be supported by executing each benchmark run after a system reset, making the initial state of the system for each benchmark run independent from the initial state for the other runs. Under the assumption of independency and identical distribution, the sets of results from multiple benchmark runs can be compared using the nonparametric statistical tests for comparing samples from two populations, such as Kolmogorov-Smirnov test, Wilcoxon rank sum test, and Kruskal-Wallis test.

Given the generally more pessimistic nature of nonparametric statistical tests when compared to parametric statistical tests, we find it useful to also assume the results of several consecutive benchmark runs to have a normal distribution and apply the parametric statistical tests. The assumption of normal distribution can be tested using Shapiro-Wilk test for normality

either directly on the results of consecutive benchmark runs or after applying a normalizing transformation, such as logarithm, reciprocal or reciprocal square root. Samples from two populations with normal distribution can be compared using the unmatched two-sample t-test. Generally, the t-test requires the two samples to have the same variance, but it is also fairly robust against the inequality of variances if the sample sizes are equal, which is our case.

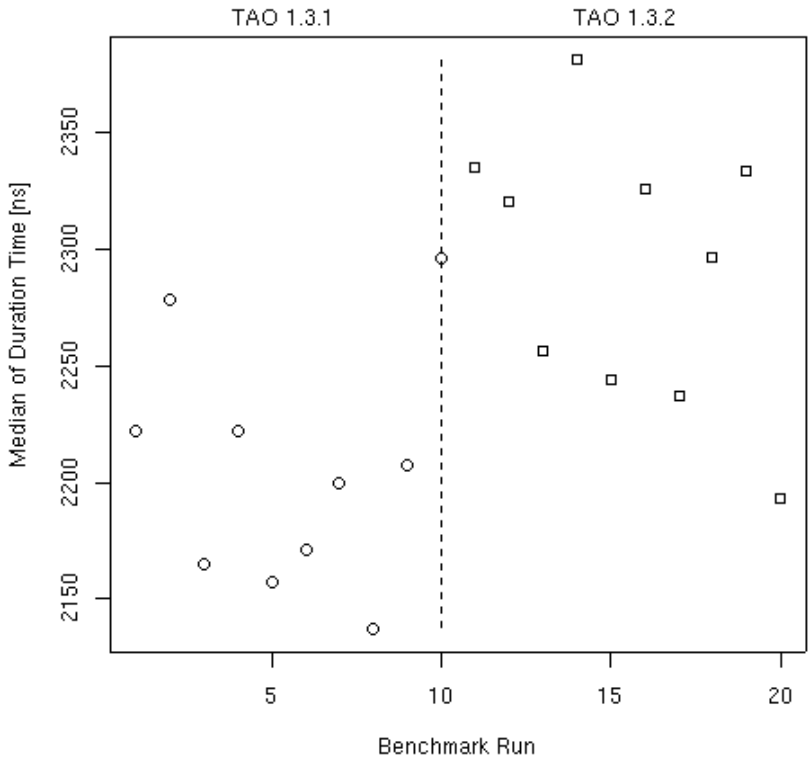


Figure 3: Results of consecutive runs of a benchmark measuring the time to marshal an input array of 1024 CORBA::ULong values on TAO 1.3.1 and 1.3.2.

The results of the technique applied to a real-world example are illustrated in figure 4. The example evaluates the development progress of the marshalling mechanism in TAO over five releases from TAO 1.3.1 to TAO 1.3.5, separated by dashed vertical lines in the figure. At the significance level of 0.05, the technique detects changes between TAO versions 1.3.1 and 1.3.2 and TAO versions 1.3.3 and 1.3.4, marked by bold vertical lines. The differences between other releases are not considered significant. The p-values of the results of several nonparametric and parametric statistical tests are tabulated in figure 5.

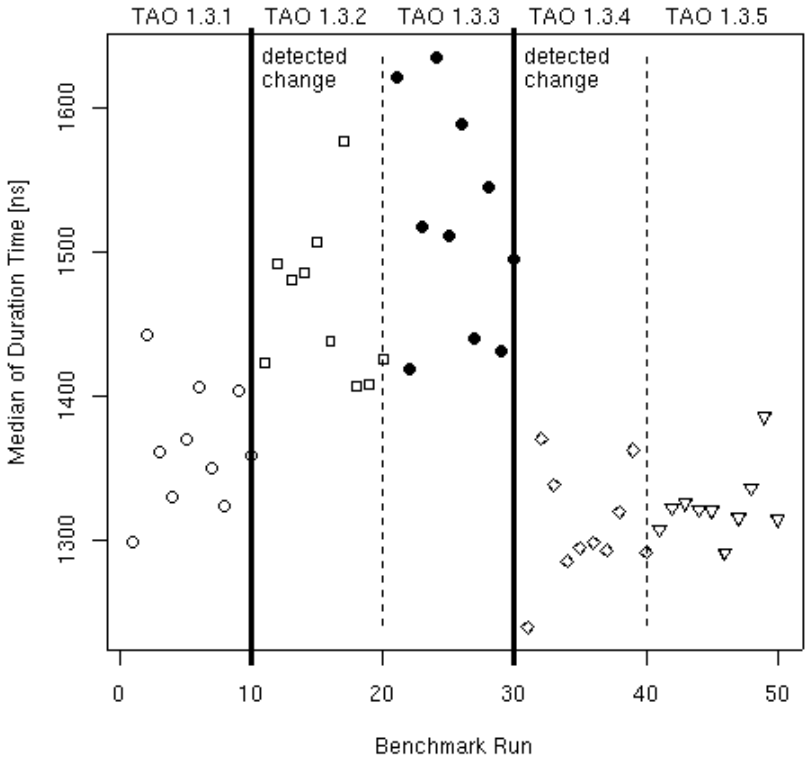


Figure 4: Results of consecutive runs of a benchmark measuring the time to marshal an input array of 1024 CORBA::Octet values on TAO 1.3.1 to 1.3.5.

Results of statistical tests (p-values)	Compared TAO versions			
	1.3.1/1.3.2	1.3.2/1.3.3	1.3.3/1.3.4	1.3.4/1.3.5
Nonparametric (raw data only)				
Kolmogorov-Smirnov	**0.003323**	0.167821	**0.000011**	0.164079
Wilcoxon	**0.000877**	0.052426	**0.000011**	0.256660
Kruskal-Wallis	**0.000765**	0.049366	**0.000157**	0.241145
Parametric (raw and transformed data)				
t-test	**0.000272**	0.078836	**0.000003**	0.328786
t-test, log(x)	**0.000225**	0.079531	**0.000001**	0.318049
t-test, 1/x	**0.000196**	0.080375	**0.000000**	0.307710
t-test, 1/sqrt(x)	**0.000209**	0.079935	**0.000001**	0.312829

Figure 5: Results of nonparametric and parametric statistical tests for consecutive runs of a benchmark measuring the time to marshal an input array of 1024 CORBA::Octet values on TAO 1.3.1 to TAO 1.3.5.

4. Complex Benchmarks

In the paper, we consider the suitability of the existing benchmarks for two broad groups of benchmarks based on their complexity. The group of complex benchmarks covers benchmarks such as [20][21][24], where a set of features of the middleware is tested under real-world workload. The intuitive justification for the complex benchmarks is that they provide results directly applicable to real-world applications.

Complex middleware benchmarks are indispensable because they exercise multiple functions of the middleware concurrently and therefore provide room for effects of complex interactions among the functions to influence the results. Unfortunately, complex benchmarks are more expensive to run than the simple benchmarks in terms of both the cost of the hardware and software setup and the time to run the benchmark. The results of a complex benchmark also have a less straightforward interpretation, especially when expressed as a single value of throughput in a number of operations per second, as in [20][21][24].

These characteristics make the complex benchmarks unsuitable for regression benchmarking. To remedy the situation, we proceed by implementing a simplified version of the TPC-W benchmark [24], which is less expensive to run and collects the duration of individual operations rather than a single value of throughput. The TPC-W benchmark simulates an online bookstore, with a hypothetical architecture in figure 6.

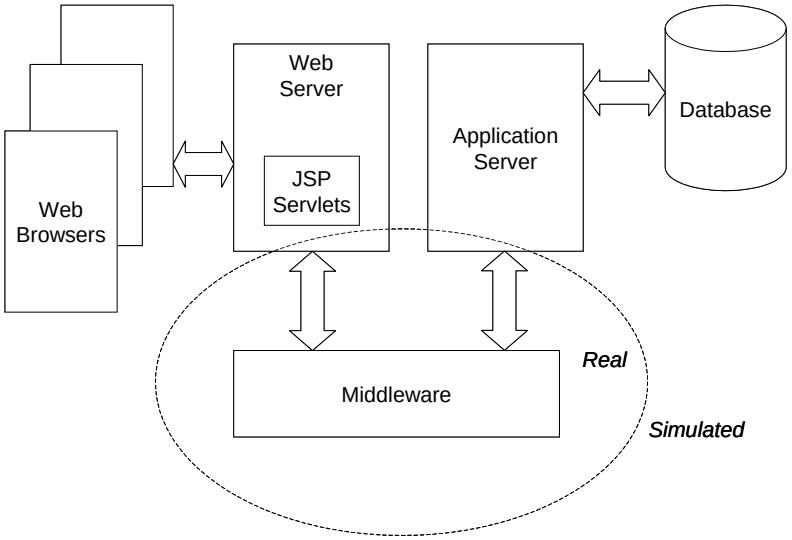


Figure 6: The architecture of an online bookstore for the TPC-W benchmark.

Our simplified version of the TPC-W benchmark reproduces the workload that the TPC-W architecture imposes on the middleware connecting the web server with the application server, without requiring the entire TPC-W architecture to be present. The only interactions that are actually executed and measured are the requests sent by the web server to the application server.

The interaction between the web browser and the web server is modeled by sending a sequence of requests to the application server for each request received by the web server. The interaction between the application server and the database is modeled as a delay for each request received by the application server that would normally access the database.

4.1. Comparing Results

Figure 7 shows the duration of one of the frequently executed operations of the simplified TPC-W benchmark, which retrieves a title and author information for a book. The figure contains results obtained by running the same benchmark twice, each time with a different version of omniORB. One version is the original omniORB, the other version is an artificially damaged omniORB which takes roughly 10% longer to complete a trivial remote method invocation. Ideally, we would want the regression benchmarking to detect the difference.

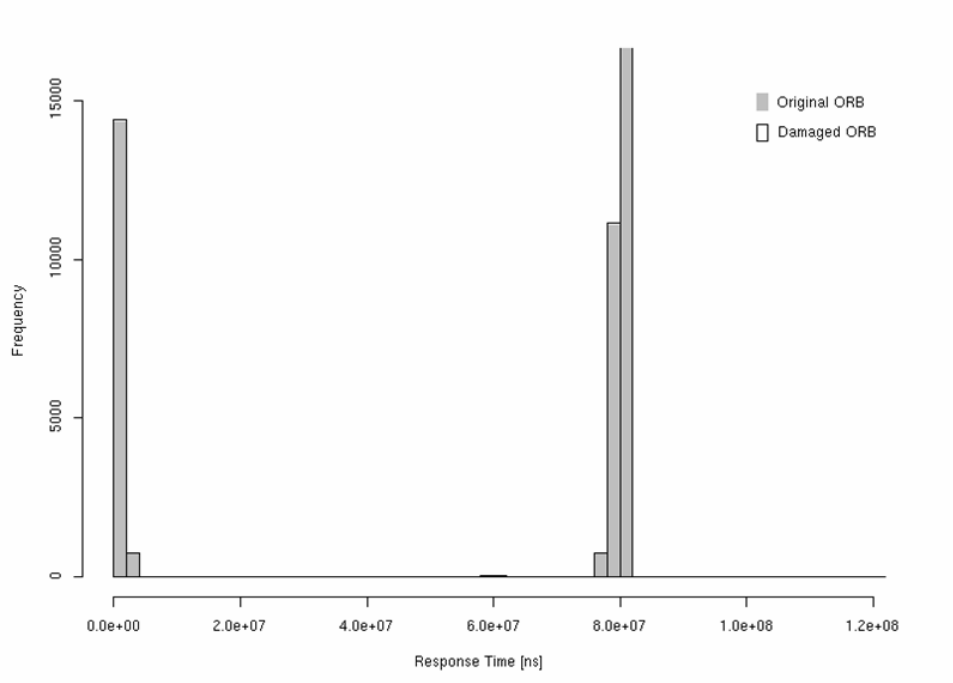


Figure 7: Results of the benchmark measuring the time to retrieve book information on original and damaged omniORB.

As is clear from the figure, the results for the original omniORB and the damaged omniORB are very similar, which means that the methods of comparing averages or medians cannot be used to detect the difference. Indeed, the averages and medians of the results are almost the same for the two versions, as shown in figure 8 for several consecutive runs of the benchmark.

Benchmark version		Response time [µs]			
		Run 1	Run 2	Run 3	Run 4
original omniORB	median	79910	79910	79910	79910
	average	55890	56070	55920	56220
damaged omniORB	median	79910	79910	79910	79910
	average	55850	55920	56330	55540

Figure 8: Medians and averages of the time to retrieve book information.

The reason why the difference between the results of the two versions is difficult to detect is apparent from figure 9, which contains a zoom in of the results. The duration of the operation differs for the cases where the operation takes about 100 microseconds, but the difference is overshadowed by the cases where the operation takes about 80 milliseconds. An analysis of the benchmark would reveal that the longer times occur because of waiting for database object instantiation.

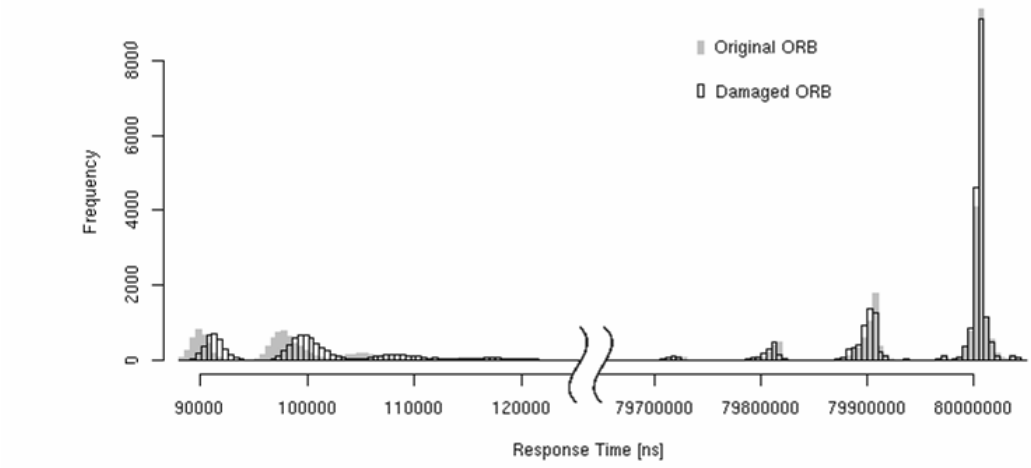


Figure 9: Results of the benchmark measuring the time to retrieve book information on original and damaged omniORB.

4.2. Clusters of Observations

To better understand the results of the two runs, we can intuitively interpret each result as a union of clusters of observations that roughly correspond to specific cases of interactions among the functions of the middleware. An example of a coarse-grained interaction is in figure 7, where the two large clusters correspond to interactions that differ in that one hits while the other misses accessing an object cache. An example of a fine-grained interaction is in figure 10, which shows how the results for three individual operations of the simplified TPC-W benchmark correspond to clusters in the complete set of results.

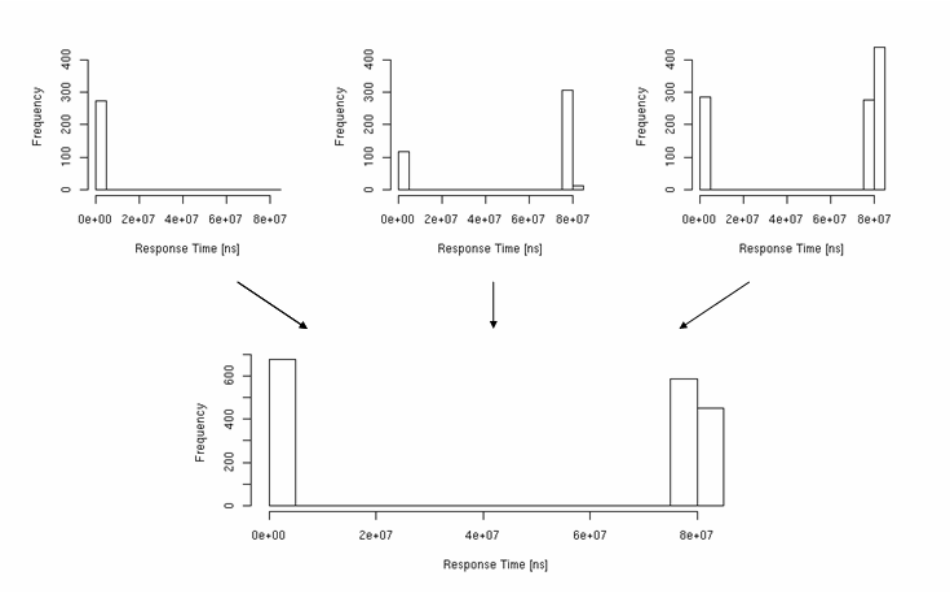


Figure 10: Contribution of three individual operations to the complete results.

While supporting the interpretation of results as a union of clusters of observations, figure 10 also emphasizes that it would be difficult to associate every single cluster with a specific interaction among the functions of the middleware. The degree of insight into the system and the amount of data collected to make such an association is technically prohibitive. Nonetheless, the differences between the results in figure 7 become visible on individual clusters in figure 9, suggesting that results can be compared cluster-by-cluster.

To compare the results cluster-by-cluster with no information about the association of every cluster with a specific interaction, the clusters need to be identified in the results based on the values of the individual observations only. In our experiments, the best results were obtained when using traditional iterative clustering algorithms that start with a set of initial centers of the clusters and then repeatedly assign data points to the nearest cluster and recompute centers of the clusters. The basic algorithm known as k-means defines the center of a cluster as a mean of the data points in the cluster. The algorithm can find a local minimum of the error measure calculated as a sum of variances of the clusters. Such a minimum corresponds to a centroidal Voronoi configuration, in which each data point is closer to the center of its cluster than to the center of any other cluster [5].

The problem of the k-means algorithm with respect to regression benchmarking is that it requires the initial set of cluster centers to be defined. The centers are often chosen randomly or heuristically. When the centers are badly chosen, the algorithm requires more iteration steps and may find worse solutions. Although improvements of the stability of the clustering results such as bagged clustering [9] exist, the problem of choosing the number of centers remains. In our example, we leave this problem open and select the initial set of cluster centers by hand. The result of the k-means algorithm is in figure 11.

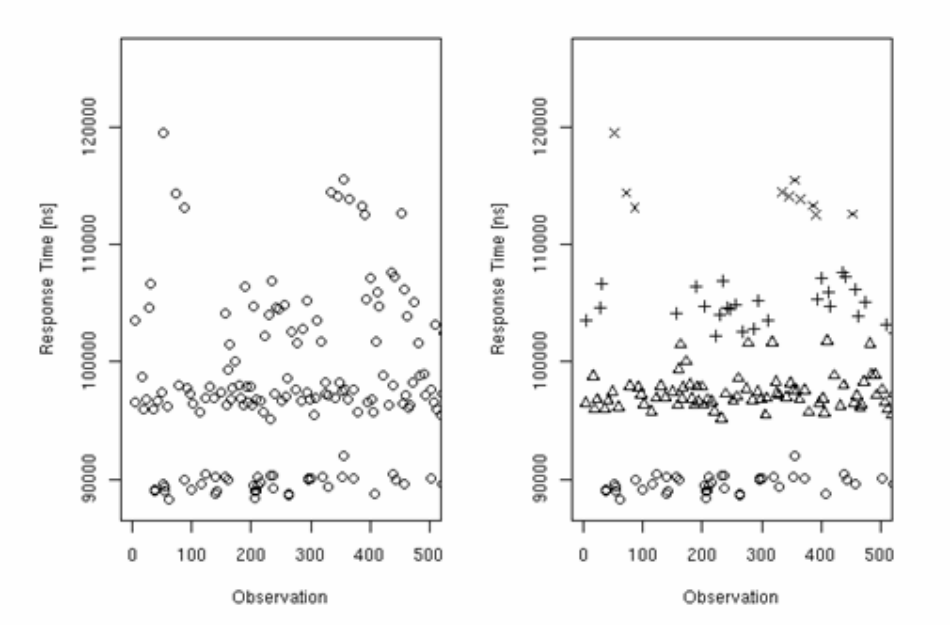


Figure 11: Clusters as identified by the k-means algorithm.

Once the clusters are identified, we can assume the observed durations to be independent identically distributed observations of a random variable and use the technique proposed in section 3. In our example, the technique classifies the identified clusters as having distributions with different means, which is the correct result.

5. Conclusion

We have outlined the idea of regression benchmarking as a variant of regression testing focused at detecting performance regressions. Applying the regression benchmarking in the area of middleware, we explain how the regression benchmarking differs from middleware benchmarking in general in its requirements on benchmark integration, benchmark automation, result precision and result analysis, and show on real-world examples why the existing benchmarks do not give results sufficient for regression benchmarking.

For the group of simple benchmarks, we propose techniques for minimizing interference of the operating system on the benchmark results, collecting the necessary number of observations, and comparing results while taking into account the fact that the results themselves are observations of a random variable. This contrasts with the current practice in middleware benchmarking, where the interference of the operating system is silently suffered, the necessary number of observations is chosen offhand, and the results themselves are incorrectly treated as precise numbers [2][7]. For the group of complex benchmarks, we propose a technique for comparing results cluster-by-cluster, again in contrast with the current practice in middleware benchmarking, where the results are expressed as a simple value of throughput [20][21][24].

With the exception of clustering, the proposed techniques are fully automated. All the techniques are demonstrated on real-world examples of middleware performance evaluation. Additional details are available at http://nenya.ms.mff.cuni.cz and in [3].

6. Acknowledgements

This work is partially sponsored by the Grant Agency of the Czech Republic grant 102/03/0672.

7. References

[1] K. Beck, Extreme Programming Explained: Embrace Change, Addison Wesley Professional, Boston, 1999.

[2] L. Boszormenyi, A. Wickener, H. Wolf, Performance Evaluation of Object Oriented Middleware - Development of a Benchmarking Toolkit, in: Proc. Euro-Par '99, Lecture Notes in Computer Science, vol. 1685, Springer, Berlin, 1999, pp. 258-261.

[3] L. Bulej, T. Kalibera, P. Tůma, Regression Benchmarking with Simple Middleware Benchmarks, in: Proc. IPCCC ’04, IEEE CS, New Jersey, 2004, pp. 771-776.

[4] E.J. Chen, W.D. Kelton, Simulation-based Estimation of Quantiles, in: Proc. WSC ’99, ACM Press, New York, 1999, pp. 428-434.

[5] V. Faber, Clustering and Continuous k-Means Algorithm, Los Alamos Science 22 (1994) 138-144.

[6] A.S. Gokhale, D.C. Schmidt, Measuring and Optimizing CORBA Latency and Scalability Over High-speed Networks, IEEE Transactions on Computers 47 (1998) 391-414.

[7] M.B. Juric, I. Rozman, M. Hericko: Performance Comparison of CORBA and RMI, Information and Software Technology Journal 42 (2000) 915-933.

[8] A.S. Krishna, J. Balasubramanian, A. Gokhale, D.C. Schmidt, D. Devilla, G. Thaker, Empirically Evaluating CORBA Component Model Implementations, OOPSLA ‘03 Middleware Benchmarking Workshop, USA, 2003, http://nenya.ms.mff.cuni.cz/projects/corba/oopsla-workshop.

[9] F. Leisch, Bagged clustering, Adaptive Information Systems and Modeling in Economics and Management Science 51 (1999).

[10] F. Plášil, P. Tůma, A. Buble, Charles University Response to the Benchmark RFI, OMG bench/98-10-04, 1998.

[11] P. Tůma, A. Buble, Open CORBA Benchmarking, in: Proc. SPECTS ’01, SCS, San Diego, 2001.

[12] AT&T Laboratories Cambridge, omniORB: Free High Performance ORB, http://omniorb.sourceforge.net.

[13] Distributed Systems Research Group, Open CORBA Benchmarking Project, http://nenya.ms.mff.cuni.cz/~bench.

[14] Distributed Systems Research Group, Vendor CORBA Benchmarking Project, http://nenya.ms.mff.cuni.cz/projects.phtml?p=cbench.

[15] Distributed Object Computing Group, TAO Performance Scoreboard, http://www.dre.vanderbilt.edu/stats/performance.shtml.

[16] Distributed Object Computing Group, The ACE Orb, http://www.dre.vanderbilt.edu/TAO.
[17] ExoLab Group, OpenORB Community, The Community OpenORB Project, http://openorb.sourceforge.net.
[18] Object Management Group, CORBA Specification 3.0.2, OMG formal/02-12-02, 2002.
[19] ObjectWeb Consortium, CAROL: Common Architecture for RMI ObjectWeb Layer, http://carol.objectweb.org.
[20] ObjectWeb Consortium, RUBiS: Rice University Bidding System, http://rubis.objectweb.org.
[21] Sun Microsystems, ECperf Specification, version 1.1, 2002, http://www.theserverside.com/ecperf.
[22] Sun Microsystems, Enterprise JavaBeans Specification, version 2.1, 2003, http://java.sun.com/products/ejb/docs.html.
[23] Sun Microsystems, Java Remote Method Invocation Specification, http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.html.
[24] Transaction Processing Performance Council, TPC Benchmark Web Commerce Specification 1.8, 2002, http://www.tpc.org.

Chapter 4

Benchmark Precision and Random Initial State

Tomáš Kalibera,
Lubomír Bulej,
Petr Tůma

Contributed paper at **2005 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS 2005)** [8].

In conference proceedings,
published by Society for Modeling and Simulation (SCS),
pages 853–862,
ISBN 1-56555-300-4,
July 2005.

Benchmark Precision and Random Initial State

Tomas Kalibera[1] Lubomir Bulej[1,2] Petr Tuma[1]

[1]Faculty of Mathematics and Physics
Charles University in Prague
Malostranske nam. 25
118 00 Prague, Czech Republic
{tomas.kalibera,petr.tuma}@mff.cuni.cz

[2]Institute of Computer Science
Czech Academy of Sciences
Pod Vodarenskou vezi 2
182 07 Prague 8, Czech Republic
bulej@cs.cas.cz

Abstract

The applications of software benchmarks place an obvious demand on the precision of the benchmark results. An intuitive and frequently employed approach to obtaining precise enough benchmark results is having the benchmark collect a large number of samples that are simply averaged or otherwise statistically processed. We show that this approach ignores an inherent and unavoidable nondeterminism in the initial state of the system that is evaluated, often leading to an implausible estimate of result precision. We proceed by outlining the sources of nondeterminism in a typical system, illustrating the impact of the nondeterminism on selected classes of benchmarks. Finally, we suggest a method for quantitatively assessing the influence of nondeterminism on a benchmark, as well as approach that provides a plausible estimate of result precision in face of the nondeterminism.

Keywords Performance Evaluation, Random Initial State, Statistical Analysis, Benchmark Precision.

1. INTRODUCTION

Software benchmarks are commonly used for empirical evaluation of performance. Typical uses of benchmarks include analysis of system behavior, evaluation of absolute system performance on selected classes of applications, or comparison of system performance on different implementations of an application [2,3]. In contrast to analytical methods or simulation, benchmarking provides results based on the behavior of a real system in a real environment. The benchmark results provide the values of various performance indicators that characterize the behavior of the system under particular workload, e.g. the typical duration of characteristic operations, memory consumption, etc.

To obtain the typical value of a performance indicator, a benchmark usually performs a number of measurements and calculates the typical value as an average, or median of the collected samples. Although this approach represents common practice, it involves making several hidden assumptions that are only rarely discussed explicitly.

The approach assumes that the values measured during a benchmark run can be considered independent and identically distributed samples of a random variable, which represents the performance indicator of interest. The justification for the assumption stems from the fact that due to complexity of contemporary hardware and software, the duration of a nontrivial operation is subject to frequently occurring but relatively small changes due to inherent nondeterminism in operation execution, as well as rarely occurring but relatively large changes due to external disruptions. The slightly changing values resulting from the nondeterminism in operation execution are considered representative for the repeated execution of the operation on a given hardware platform. The values distorted by external disruptions are, depending on the purpose of the benchmarking experiment, considered either extremal (with respect to the typical execution times of the exercised operation) or a part of the overall behavior of the system.

The typical value of a performance indicator would be then a value representative of the distribution of the random variable. Such values are usually the mean and the variance. Based on the kind of data provided by the benchmark, we can safely assume that they exist. The mean and the variance are typically estimated using average and sample variance. The typical value of a performance indicator is therefore a statistical estimate of the mean, and as such has a limited precision. Although the exact demands on the precision of the benchmark results depend on the particular use of the benchmark, we can safely state that a sufficient precision often needs to be as good as units of percents, simply because the very use of a benchmark suggests that a precise empirical evaluation, rather than a simple offhand estimate, is needed [5,12].

As follows from the Central Limit Theorem, the probability statements about the average can be approximated using the Normal distribution. The precision of the average can then be determined using the length of confidence interval for

This work was partially supported by the Grant Agency of the Czech Republic projects 201/03/0911 and 201/05/H014.

the mean, which can be estimated using a well known formula. To improve the precision of the benchmark results, a benchmark usually collects and computes the typical values using more samples. This approach can also be considered common practice, and has a theoretical foundation in the Weak Law of Large Numbers.

Consequently, a typical benchmark executes an operation multiple times and reports an average calculated from thus collected multiple samples as a result. The chance that the result is distorted by a rare sample to a degree outside the sufficient precision grows smaller with larger number of samples. In presence of extreme values caused by external disruptions, we can use robust statistic estimators such as median to mitigate the influence of the outliers, yet with the use of other estimators than average, the computations are more complex and without the theoretical backing of the Weak Law of Large Numbers and the Central Limit Theorem.

This model of a benchmark, however, fails to account for distortions caused by factors other than the inherent nondeterminism and external disruptions occurring during the operation execution. Consequently, for some benchmarks, the estimate of the mean and its precision calculated from thus obtained data do not truly represent the typical value of a performance indicator within the calculated error margin.

In this paper, we focus on improving the plausibility of the results by also accounting for distortions caused by a nondeterministic part of the initial state of the system that is evaluated. In Section 2 we point out the discrepancies between the common assumptions and the behavior observed on a simple benchmark, which suggest that the initial state is partially nondeterministic. Section 3 presents a measure of the influence of the random initial state on benchmark results and demonstrates the application of the measure on several examples of typical benchmarks. Section 4 outlines the possible sources of nondeterminism in the initial state and shows their impact on further examples of benchmarks. In Section 5, we suggest an approach to dealing with the nondeterministic part of the initial state and conclude the paper in Section 6.

2. SOMETHING IS ROTTEN

Consider the simple model of a benchmark introduced in Section 1, which empirically evaluates the duration of a specific operation execution. The duration is subject to random changes both inherent and external to the benchmarked operation. The simple model also assumes that apart from the inherent nondeterminism during execution of the operations, the duration of the operation depends only on the operation being performed. We show that this assumption is not sufficient and provide an extension to the simple model, where the duration of the operation execution also depends on the state of the system that is evaluated just before the operation execution. In itself, this extension is obvious rather than revolutionary, until the state is examined in more detail.

In a typical system, the state that can influence the duration of the operation execution consists of many parts, ranging from minute details such as the state of the processor branch prediction logic or internal memory cache or the paged state of the memory accessed during the operation execution, to significant hardware and software settings such as the bus clock speed or the virtual machine settings. Depending on the impact of the operation execution, the individual parts of the state can be broadly divided into two categories. Parts such as the state of the processor branch prediction logic or internal memory cache will most likely get changed by the operation execution, and are here termed as the *mutating* parts of the state. Parts such as the bus clock speed or the virtual machine settings will most likely stay unchanged by the operation execution, and are here termed as the *initial* parts of the state.

A benchmark typically handles the mutating and initial parts of the state in different ways. The setting of the mutating state is done by the benchmark itself, which introduces warm-up in addition to measurement. The operation execution is the same during warm-up and measurement, but samples are only collected during measurement. A long enough warm-up will set the mutating state so that the collected samples are representative for repetitive operation execution and independent of the mutating state before warm-up. The setting of the initial state is either hard coded in the benchmark or done outside the benchmark, forming a part of the system configuration to which the collected samples are representative.

A generally accepted requirement is that benchmark results should be reproducible. In terms of the mutating and initial parts of the state, reproducibility means performing long enough warm-up to make sure the mutating state is reproducible, and describing configuration precisely enough to make sure the initial state is reproducible, thus covering all parts of the state that can influence the duration of the operation execution [1, 4].

Before extending the simple model further, we can examine the assumptions made so far on an example of the FFT benchmark [10], which measures the duration of a Fast Fourier Transform of a predefined sequence.[1] Figure 1 shows individual samples collected by the benchmark during multiple runs, where a run is defined simply as a single execution of the benchmark application by the operating system. The horizontal axis is a sample index, the vertical axis is the sampled FFT computation time. Vertical lines denote new benchmark runs.

[1] The implementation of the FFT benchmark was run on Dell Precision 340 workstation with Intel Pentium 4 at 2.2 GHz and 512 MB RAM, running Fedora Core 2 with kernel 2.6.5 and gcc 3.3.3.

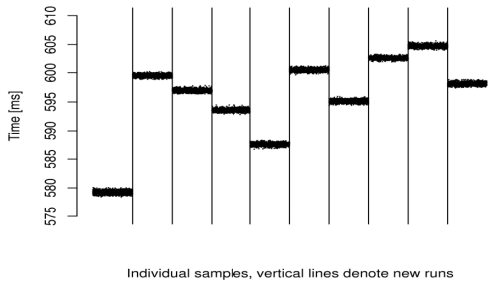


Figure 1. Impact of initial state on FFT benchmark results.

Figure 1 shows that the samples collected during a single run of the FFT benchmark are much closer to each other than the samples collected during different runs of the benchmark. The difference indicates that the initial state of the benchmark was not the same for each run, even though each run of the benchmark was executed on the same idle and isolated system, with disabled randomization of virtual memory address allocation, immediately after a reboot into a dedicated run level, using the same files and settings.

The extreme conditions of this experiment show that no reasonable effort could guarantee the initial state of the benchmark to be the same for each run. We have to cope with the deceitful reality that the initial state is partly random for each benchmark run.

As evidenced by the results of the FFT benchmark in Figure 1, the impact of the random initial state can completely overshadow the impact of the mutating state, so that taking more than several samples in a benchmark cannot improve the ability of the benchmark to describe real systems. This contrasts with the assumptions made in the simple model, where collecting more samples in a single run improves the benchmark precision. We therefore introduce a new model which recognizes the random initial state with a more realistic concept of benchmark precision discussed in Section 5.

3. QUANTIFYING THE PROBLEM

Although the influence of the random initial state can be often demonstrated graphically as shown in Figure 1, it can be difficult to notice for large data sets or in less obvious cases. Moreover, the graphical representation does not provide a simple measure of the degree to which the random initial state influences the benchmark results.

Considering the behavior of the FFT benchmark described in Section 2, a simplistic approach to quantification of the influence of random initial state would compare the standard deviation of all samples collected in all benchmark runs against the standard deviation of samples from individual benchmark runs.

However, the simplistic approach does not take into account the fact that the sample standard deviation calculated from all collected samples depends on the number of samples in each benchmark run and on the number of runs. This impairs the credibility of such a measure as it is unclear how to choose the number of samples and the number of benchmark runs.

An extreme choice would be to collect a high number of samples from a single run, thus rendering the method unusable. Instead, the choice of these numbers should be based on the measure itself, collecting a few samples from a high number of benchmark runs to reflect the impact of random initial state as well as collecting a high number of samples from a few runs to reflect the impact of the mutating state.

The dependency on the number of samples per run and the number of benchmark runs can be avoided when comparing standard deviation of samples taken from different runs against the standard deviation of samples from individual runs. Taking the above into account, we define the measure of the influence of the random initial state, termed *impact factor,* as the ratio of the standard deviation of samples from different benchmark runs to standard deviation of samples from individual benchmark runs. The value of impact factor greater than 1 indicates the influence of the random initial state on the benchmark results.

The calculation of the impact factor for a particular benchmark requires samples from multiple runs of the benchmark. To obtain plausible quantification of the influence of the random initial state on benchmark results, the calculation

Input:
1. *m* data sets corresponding to *m* benchmark runs, each containing *n* samples
2. number of samples *c* to choose randomly, $c < min(m,n)$
3. number of iterations *k*

Output:
1. estimate of the impact factor for the input data

Algorithm:
1. repeat *k* times
    1. randomly choose *c* data sets corresponding to *c* benchmark runs
        1. randomly choose 1 sample from each of the *c* selected data sets
        2. compute standard deviation SD1 from the *c* selected samples
    2. randomly choose 1 data set from the input data
        1. randomly choose *c* samples from the selected data set
        2. compute standard deviation SD2 from the *c* selected samples
    3. compute ratio SD1/SD2 of the two standard deviations
2. compute the impact factor as a median of the *k* ratios

Figure 2. Algorithm for estimating impact factor of random initial state.

of the impact factor requires relatively large amounts of data. In order to save time and resources, we calculate the impact factor using a bootstrap method described in Figure .

Table 1 shows the impact factor for several benchmarks and systems, and demonstrates that the influence of the random initial state on the benchmark results exists in a wide variety of benchmarks. The Marshaling benchmark measures the duration of marshaling of a string constant during a simple remote procedure call. The Ping benchmark measures the duration of a simple remote procedure call. The RUBiS benchmark measures the duration of an operation on an auction website [4].[2] We have used a modified version of the RUBiS benchmark which allowed us to track the response times of individual methods.

Table 1. Impact factor of random process initial state.

Benchmark	Runs	Samples per Run	Impact Factor (median)
FFT P4/FC2	150	2000	25.81
Marshaling P4/FC2	100	100000	2.61
Ping P4/FC2	100	100000	1.10
FFT P4/DOS	100	2000	1.06
FFT P4/W2K	100	2000	94.74
FFT IA64/Sarge	100	2000	35.91
RUBiS P4/FC2	15	15 min. ~ 5500	1.01

For each benchmark, we have executed a number of runs, as indicated in Table 1. In each run, we have collected between 2000 and 100000 samples, depending on the benchmark. In case of the RUBiS benchmark, the number of samples was determined by the default execution time limit. Where possible, the numbers of runs and samples were intentionally set high to provide more than enough data for the bootstrap. We have performed 10000 iterations of the bootstrap method on the input data, with the number of randomly chosen samples c set to $0.75m$ for increased robustness. The impact factor has been computed as the median of 10000 ratios of standard deviations of samples collected during different runs of the benchmark to standard deviations of samples collected during an individual run of the benchmark.

[2] The Pentium 4 Linux and Windows systems were Dell Precision 340 workstation with Intel Pentium 4 at 2.2 GHz and 512 MB RAM. The Itanium system was Dell PowerEdge 7150 server with two Intel Itanium processors at 800 MHz and 1 GB RAM.

The results indicate that the impact factor is very large in the FFT benchmark, except for DOS operating system where it is negligible, significant for the Marshalling benchmark, small for the Ping benchmark and negligible for the RUBiS benchmark. These results agree with the analysis of the sources of nondeterminism provided in Section 4.1.

4. SOURCES OF NONDETERMINISM

The influence of the random initial state on the benchmark results is somewhat contrary to the common understanding of a benchmark as, by and large, a reproducible process. To exclude the possibility of non-reproducible benchmark results discussed in Section 2, which points to the existence of the random initial state and its influence on the benchmark results, we further trace and explain some causes of the nondeterminism in the initial state. We also show that attempts to eliminate the sources of nondeterminism do not provide reliable, long term benefits.

4.1 Nondeterminism in Memory Allocation

One source of nondeterminism in the initial state is related to memory allocation. Among the many activities an operating system performs when starting a benchmark application is the allocation of memory for the code and the data of the benchmark. The allocation entails the selection of the virtual addresses for the code and the data of the benchmark and the assignment of physical pages to back the allocated virtual addresses. Even though neither the selection of virtual addresses nor the assignment of physical pages has to remain unchanged during the benchmark execution, it is likely to remain so, especially when the system that executes the benchmark has enough memory to avoid swapping. Even when the operating system assigns physical pages on demand, the assignment takes place during the warm-up phase of the benchmark and, again, is likely to remain unchanged during the data-collection phase of the benchmark.

The selection of virtual addresses and the assignment of physical pages can lead to a different distribution of cache hits and misses during the execution of the benchmark. The difference is mainly caused by the limited associativity of TLB (Translation Look-Aside Buffer) and memory caches, with the hardware mapping several physical addresses to the same cache slot. Since programs do not access their virtual pages in an uniform way, different assignment of physical pages to virtual addresses leads to different numbers of cache hits and misses, therefore influencing the benchmark results. This influence can be verified by relating the benchmark results with the values of processor-specific performance counters that keep track of TLB and memory cache hits and misses.

Figure 3 relates the counts of memory cache misses with the results of the FFT benchmark from Figure 1 for execution on two platforms, both with fixed selection of virtual address-

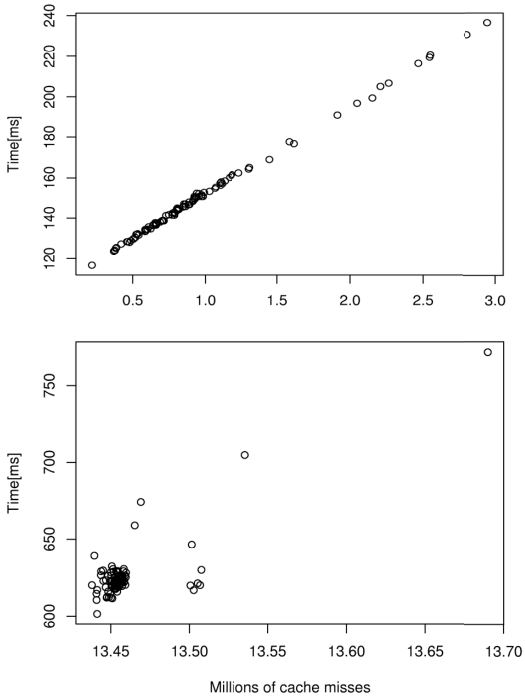


Figure 3. Correspondence of result fluctuation of FFT benchmark with memory cache misses.

es and random assignment of physical pages. The top plot in the figure is for the Intel Itanium processor and shows a strong positive linear dependency of measured times on memory cache misses. The bottom plot in the figure is for the Intel Pentium 4 processor and shows a weak positive correlation, however the results for this platform are only approximate due to a bug in the processor which prevents precise counting of memory cache events. The clear relationship visible in Figure 3, as well as the difference of impact factor for systems with different TLB and memory cache architectures in Table 1, confirms the impact of the memory cache misses on the benchmark results.

While the selection of virtual addresses by the operating system can be made deterministic and therefore reproducible at least on some operating systems, the assignment of physical pages to virtual addresses is not reproducible without nontrivial modifications of the operating system, which is obviously out of the question for many benchmarks.

4.2 Nondeterminism in Code Compilation

Another source of nondeterminism in the initial state is related to code compilation. The process of compilation and linking of a benchmark application into a binary is not necessarily reproducible even when using the same compilation and linking commands on the same benchmark sources.

Table 2 shows the effect of nondeterminism in compilation and linking on the Marshaling and Ping benchmarks from Table 1. The cause of nondeterminism in this particular example can be observed with the GNU C++ compiler [6] since version 3.3.1. The compiler uses random name mangling for symbols defined in anonymous name spaces. This can result in the linker linking the symbols in different order and placing them at different addresses in the binary. The placement of the symbols on different addresses in the binary is then reflected in placement of the symbols on different addresses in memory during dynamic linking, which in turn influences the benchmark results in a similar way as the nondeterminism in memory allocation.

Table 2. Impact factor of random initial state in compilation.

Benchmark	Runs	Impact Factor (median)
Ping	100	1.13
Marshaling	100	1.06

Although the cause of nondeterminism in this particular example can be avoided by using certain compiler options, it may not be possible for all benchmark sources. More importantly, the causes of nondeterminism described both in Section 4.1 and in Section 4.2 are difficult to avoid, which also makes it difficult to prove that they are the only causes of nondeterminism in the initial state [9]. In fact, more causes of nondeterminism in the initial state can be found depending on the benchmark and the system that executes the benchmark.

4.3 Nondeterminism is Unavoidable

The nondeterministic part of the initial state can be eliminated either by removing the individual sources of nondeterminism or by simulating the whole system. Both approaches have their drawbacks.

As outlined in previous sections, removing the sources of nondeterminism is not very feasible, mainly because we cannot ensure that all sources of nondeterminism in particular benchmark experiment have been identified and eliminated. Even though it may seem that the nondeterminism in memory allocation described in Section 4.1 could be eliminated by running a benchmark experiment immediately after system reboot, this solution is not sufficient on contemporary systems. Complex benchmarks, which require services such as databases or web servers, will never start in a deterministic fashion, even after system boot.

In case of simple benchmarks, many system services, such as page daemon, I/O daemons, or file system journaling daemon, are often provided by the operating system kernel and cannot be shut down. Moreover, these services are started concurrently with the order of their execution determined by the current hardware state, which will again result in nondeterministic initial state for a benchmark executed during system boot sequence.

The nondeterminism in code compilation described in Section 4.2 can be eliminated for some benchmarks, but may occasionally result in the compiler generating incorrect code. Also, the only way to find out whether the benchmark is still influenced by nondeterministic initial state is to run the benchmark multiple times and observe the differences in the results. Then it may actually be easier to obtain the results using the approach we describe further, which is based on analysis of the results from multiple runs of the same benchmark.

Without a sound approach to eliminating the causes of nondeterminism in the initial state, we may consider simulating the whole system, which would be a deterministic process and would allow setting identical initial conditions prior to each execution of a benchmark. While simulation is considered a useful tool in the area of performance evaluation, its application to benchmarking would be time consuming at best. Today's benchmarks, applications and system hardware are so complex that simulating even a second of the benchmark execution would consume an enormous amount of time and resources. Creating a simpler model or actually simplifying the benchmark or the application for the purpose of simulation defies the obvious goal to evaluate performance of real systems in a realistic scenario. In addition to that, specification of software and hardware that would allow designing such simulation would be hard to get for current systems mostly for licensing reasons.

This further underscores the fact that a benchmark should not be understood as a fully reproducible process, which has been the practice so far, but rather as a process that is inherently and unavoidably nondeterministic because of the existence of the random initial state and its influence on the benchmark results.

5. LIVING WITH NONDETERMINISM

The presence of nondeterminism in the initial state forces us to reconsider and ask what is the result and the precision of a benchmarking experiment.

Without considering the nondeterminism in the initial state, the answer to the question is rather straightforward. The benchmarking experiment consists of a single run of a benchmark, which collects a number of samples to determine the value of a performance indicator. These samples are representative for the repetitive execution of the measured operation as described in Section 1. The samples are considered independent and identically distributed, but the distribution is typically unknown. To estimate a single value which is a representative of the repeated execution of the measured operation, an average of the samples is typically used, for reasons outlined in Section 1. From the weak law of large numbers, even though the distribution is unknown, the average of the samples is a good estimate of the mean value of the underlying distribution, which is usually considered a representative value for the distribution. The precision of a benchmark is identical to the precision of the estimate of the mean value and can be assessed using the central limit theorem.

In presence of nondeterminism in the initial state, we can either attempt to eliminate the nondeterminism and thus achieve the deterministic initial conditions that have been so far silently assumed in common practice, or we can accept the nondeterminism as a trait of a real system and focus our effort towards obtaining meaningful results in presence of nondeterminism in initial state.

As discussed in Section 4.3, all sources of nondeterminism in a particular benchmark cannot be always reliably eliminated, not even considering the time and resources required for a detailed analysis of a system in order to identify the sources of nondeterminism. The other approach is to accept the influence of the nondeterministic initial state as a part of a real system and take it into account when benchmarking. For a performance indicator of interest, this requires determining the value and precision that are representative of both the repeated execution of the measured operation and the impact of the nondeterministic initial state.

5.1 Measuring with Nondeterminism

We have shown that the random initial state influences the benchmark results when benchmarking on real systems and that this influence cannot be easily eliminated. Because of the random initial state, multiple runs of a benchmark will yield different values of the performance indicator of interest as benchmark results. The value of the performance indicator reported by a single benchmark run is therefore a representative of the sampled values in a system with a particular realization of the random initial state, which includes the state of the operating system process with respect to assignment of physical to virtual pages as described in Section 4.1 and Section 4.2, and possibly other components.

In face of the described nondeterminism, an ideal benchmarking experiment would be set up so that the benchmark would be recompiled, restarted and warmed up before collecting each individual sample. In this setup, the collected samples would still be independent and identically distributed, therefore the average of all collected samples would still be a good estimate of the mean. This setup would ensure the repeatability of benchmarking experiments and the precision of the estimate of the mean would better reflect the behavior of the real system in a real environment.

However, there are two issues with the above scenario. First, if we only collect a single sample after warm-up phase during each benchmark run, we risk collecting a more or less distorted sample, caused by the inherent nondeterminism in operation execution or by an external disruption. To reduce the chance of invalidating the results because of a single distorted sample, we would have to collect a large number of samples. This leads us to the second issue related to the ideal scenario, which is efficiency of the benchmark experiment.

Described as it was, the ideal setup of the benchmarking experiment is very expensive to run. Imagine a Ping benchmark which uses CORBA (Common Object Request Broker Architecture) for remote communication. The compilation of the benchmark including the ORB (Object Request Broker) takes about 30 minutes, starting and warming up a benchmark takes several seconds and measuring one Ping request takes tens of microseconds.

Under such circumstances, collecting more than a few samples to obtain the sufficient precision of the estimate of the median would require an enormous amount of time. Or the other way around, the number of samples collected in a reasonable time may be far too low to obtain the sufficient precision.

Obviously, our goal is to reduce the number of runs to save time, but not so as to lower the precision by increasing the chance of collecting a distorted sample. With the knowledge of the influence of the random initial state on the benchmark results, we may save time by repeating the measured operation several times in each run to improve the precision of the result of a single run, which consequently allows us to reduce the required number of runs.

5.2 Efficient Benchmarking with Nondeterminism

Although the ideal setup of benchmarking in the face of nondeterminism outlined in Sections 4.1 and 4.2 requires rebuilding and restarting each benchmark to get precise and repeatable results, it is not always necessary to restart a benchmark run to obtain a new sample. The impact factor for a benchmark is not always so high as for the FFT benchmark and therefore there can be overlap between possible values of samples in different benchmark runs. Consequently, each benchmark run can contribute more than one sample to the estimate of the mean that is representative for both the repeated execution of the measured operation and the impact of the nondeterministic initial state.

In the following paragraphs we show that for an additive dependency of a sample on the random initial state, using this approach provides the same estimate of the mean as the one described in the model experiment from Section 5.1. Moreover, we can calculate the precision of the estimate and, given the number of samples required for benchmark warm-up, determine the optimal number of samples per benchmark run to obtain the most precise estimate of the mean. Concluding the section, we show how the additive dependency explains the influence of random initial state on benchmarks described in Section 3.

The additive model of dependency of a sample on the random initial state is defined as follows[3]:

1. the random initial state for each benchmark run is represented by a random sample a of a random variable A with finite variance and mean value
2. samples collected in a benchmark run with random initial state a (a is fixed) are independent, identically distributed samples of random variable $R(a)$, where $R(a) = a + X$, and X is a random variable with finite variance and mean value

The model applies only to samples after the warm-up phase of a benchmark. In the scope of this model, the ideal benchmarking experiment from Section 5.1 has the following interpretation:

1. for each benchmark run i, where $i = 1..k$, a single sample $r_i = a_i + x_i$ of random variable $(A+X)$ is collected
2. the average $\bar{r}_k$ of samples r_i estimates the mean value $E(A+X)$ of random variable $(A+X)$, which is representative for the repeated execution of the measured operation and the influence of the random initial state on benchmark results

The additive model of dependency is very similar to a random effect model with one-way specification from [11], which is based on normal distribution. The additive model we present here, however, does not assume the normal distribution of X and A. The distributions of X and $R(a)$ in our experiments were right-skewed, and thus could not be assumed normal.

For the additive model of dependency, we can show that the same estimate of the mean of the random variable $(A+X)$ can be obtained even when each benchmark run contributes more than one sample to the estimate. Consider a benchmarking experiment that consists of k benchmark runs, each collecting n samples of the performance indicator of interest. The i-th sample in j-th benchmark run is labeled $r_{j,i}$, $r_{j,i} = a_j + x_{j,i}$.

The average of all samples from all runs

$$\bar{r}_{k,n} = \frac{1}{k \cdot n} \cdot \sum_{j=1}^{k} \sum_{i=1}^{n} r_{j,i}$$

can be expressed as $\bar{r}_{k,n} = a_k + \bar{x}_{k,n}$.

[3] For better readability, we use a relaxed notation, where random variables are sometimes denoted by lowercase letters, which is a common approach in literature on hierarchical models, such as [11]. We also use the word *sample* in a broader sense, not distinguishing rigorously between realizations and random variables; the precise meaning is always clear from the context.

From the central limit theorem, the distributions of $\bar{a}_k$ and $\bar{x}_{k,n}$ can be approximated as

$$\bar{a}_k \approx E\,A + N\left(0, \frac{var\,A}{k}\right) \text{ and } \bar{x}_{k,n} \approx E\,X + N\left(0, \frac{var\,X}{k \cdot n}\right).$$

From the properties of the normal distribution it follows that

$$\bar{r}_{k,n} \approx (E\,A + E\,X) + N\left(0, \frac{var\,A}{k} + \frac{var\,X}{k \cdot n}\right) = \\ = (E\,A + E\,X) + N(0,1)\sqrt{\frac{n \cdot var\,A + var\,X}{k \cdot n}}. \quad (1)$$

Therefore the mean value $E\bar{r}_k$ of the average $\bar{r}_k$ is the mean value E(A+X) of random variable (A+X), which is representative for the repeated execution of the measured operation and the influence of the random initial state on benchmark results. The confidence interval for E(A+X) can be constructed as follows

$$\bar{r}_{k,n} \pm z_{1-\frac{\delta}{2}} \sqrt{\frac{n \cdot var\,A + var\,X}{k \cdot n}}, \quad (2)$$

where $z_{1-\delta/2}$ is a 1-δ/2 quantile of the normal distribution. With probability 1-δ, the interval contains the true value of E(A+X), which is the representative value of the performance indicator of interest and the correct result of a benchmark experiment. Even though the distributions of A and X are unknown, we can estimate their variances *var(A)* and *var(X)* because

$$var\,(\mathrm{R}(a)) = var\,(a+X) = var\,(X),$$

and thus by estimating variance of samples in any benchmark run we are also estimating the variance *var(X)* of the random variable X. Since we need to perform multiple runs of the benchmark, we can estimate the variance of each benchmark run and calculate the average of these estimates:

$$\tilde{S}(X)^2_{k \cdot n} = \frac{1}{k}\sum_{j=1}^{k} S_j(R)^2_k = \frac{1}{k}\frac{1}{n-1}\sum_{j=1}^{k}\sum_{i=1}^{n}(r_{j,i} - \bar{r}_{j,k})^2. \quad (3)$$

In addition, the following also holds

$$\mathrm{E}(R(a)) = \mathrm{E}(a+X) = a + \mathrm{E}(X).$$

To estimate the variance *var(A)*, let us consider a random variable M, the values of which are the mean values of the performance indicator in benchmark runs with different values of the sample a of the random variable A,

$$\mathrm{M} = \mathrm{E}_A(R(a)) = A + \mathrm{E}(X).$$

Since the following holds

$$var\,(M) = var\,(A + \mathrm{E}(X)) = var\,(A),$$

the estimate of variance *var(M)* also estimates the variance *var(A)*. As already mentioned, the samples of the random variable M are the mean values of different benchmark runs, and can be therefore estimated by the average of samples from the respective benchmark run.

We then estimate the variance *var(A)* as

$$\tilde{S}(A)^2_k = \frac{1}{k-1}\sum_{j=1}^{k}(\bar{r}_{j,n} - \bar{r}_{k,n})^2. \quad (4)$$

Substituting the estimates of variance (3) and (4) for the true but unknown variances in (2), the confidence interval for the mean value of the performance indicator of interest is then

$$\bar{r}_{k,n} \pm z_{1-\frac{\delta}{2}} \sqrt{\frac{n \cdot \tilde{S}(A)^2_k + \tilde{S}(X)^2_{k \cdot n}}{k \cdot n}}. \quad (5)$$

When the number of collected samples kn is small, the quantiles z of the normal distribution are replaced with quantiles of the t-distribution with $k(n - 1)$ degrees of freedom, because the true but unknown variances were replaced by sample variances.

Under the assumption of the additive model of dependency on the random initial state, the confidence interval (5) allows determining the precision of the benchmark result. The formula for constructing the confidence interval can also be used to determine the optimal number of samples that should be collected in each benchmark run.

To minimize the error of the estimate of the mean value of a performance indicator, we have to minimize the factor

$$\sqrt{\frac{n \cdot var\,A + var\,X}{k \cdot n}}. \quad (6)$$

We can define the cost of the benchmark experiment in terms of the total number of samples that need to be collected as

$$cost = k(w + n), \quad (7)$$

where w is the number of warm-up samples that have to be collected at startup of each benchmark run, but cannot be included into the evaluation; w also includes the price of starting a new benchmark run. By expressing k from (7), substituting it into (6), deriving by n and finding roots, we get the optimal value of n for the given *cost* and warm-up w

$$n = \left\lceil \sqrt{\frac{w \cdot \tilde{S}(X)^2}{\tilde{S}(A)^2}} \right\rceil. \quad (8)$$

Of particular interest should be the fact that the optimal number of samples does not depend on the value of *cost*, but only on the number of warm-up samples w and the values of $\tilde{S}(A)^2$ and $\tilde{S}(X)^2$. It is therefore possible to obtain benchmark results with better precision using existing results without a loss of efficiency with respect to the cost of the benchmarking experiment.

The formula also suggests that, for benchmarks where the variance of samples from different runs is much greater than the variance of samples from individual runs, increasing number of samples in runs does not help, which is the case of FFT benchmark (Figure 1). Also, we should note that the precision of the result obtained from (8) depends on the precision of the estimates of variances $\tilde{S}(X)^2_{k \cdot n}$ and $\tilde{S}(A)^2_k$.

Even though the additive model of the dependency on the random initial state allows for more efficient measurement (in contrast to the ideal benchmarking experiment described in Section 5.1) and determining the precision of the benchmark results, it remains to be decided how well can be a particular benchmark modeled under the assumption of additive dependency.

For this purpose, we can use the impact factor defined in Section 3. The additive dependency model is characteristic by shifting all the samples in a benchmark run by a constant. For each benchmark run with a different sample a of the random variable A the following holds

$$R(a) - \mathrm{E}(R(a)) = a + X - (\mathrm{E}X + a) = X - \mathrm{E}X.$$

Under the assumption of the additive dependency model, $(R(a) - \mathrm{E}(R(a))$ has the same distribution with variance *var(X)* and mean value $\mathrm{E}X = 0$ for every sample a, which means that it does not depend on a particular benchmark run.

Remember the impact factor defined in Section 3, which provides a measure of difference between samples in different runs and samples in individual runs. We can apply the impact factor calculation on transformed data, in which we have subtracted the average of samples in one run from all samples in the run. If the impact factor for the transformed data is smaller than for the original data, we can consider the influence of the random initial state on that particular benchmark to follow the additive dependency model. If the value of the new impact factor tends to 1, we can say that the additive dependency model describes the influence of the random initial state on that particular benchmark well.

Table 3 shows the result of applying the impact factor calculation on transformed data of benchmarks described in Section 3. We can see that while on certain platforms the random initial state influences the results of the FFT benchmark to a very high degree (impact factor nearly 90), the additive dependency model describes the influence very well (impact factor calculated from transformed data tends to 1).

Table 3. Difference in impact factor calculated from transformed data.

Benchmark	Impact factor (original data)	Impact Factor (transformed data)
FFT P4/FC2	25.81	1.00
FFT P4/DOS	1.06	1.00
FFT IA64/Sarge	31.78	1.01
FFT P4/W2K	87.48	1.01
Marshaling P4/FC2	2.61	1.20
Ping P4/FC2	1.09	1.00

6. CONCLUSION

We have shown that the results of software benchmarks executed on contemporary computer architectures and operating systems are likely to be influenced by a random initial state of the system. Depending on the benchmark and the system that executes the benchmark, the influence of the random initial state can lead to non-realistic and non-repeatable benchmark results and an implausible estimate of the precision of the results. Due to its somewhat counterintuitive nature, this situation can remain unnoticed, especially when the benchmark is incorrectly understood as a deterministic and therefore reproducible process.

Experiments with different benchmarks representing the classes of scientific computation benchmarks, distributed middleware benchmarks and micro-benchmarks show that in presence of the random initial state, the traditional approach of providing a representative value of a performance indicator based on a single benchmark run provides results that are representative only for that particular run. Improving the precision by collecting more samples during a single run is counterproductive, because it consumes more time and resources and only improves the precision of the result with respect to that run.

Using a bootstrap method to calculate an impact factor of the random initial state on a benchmark, we can quantify how much a particular benchmark is susceptible to the influence of the random initial state. We point out that while the influence of the random initial state on the benchmark results can be traced and explained in detail, it may be unavoidable. We show that in such a case, it is possible to obtain a value of a performance indicator that is representative for the benchmarked application and the system it runs on using samples from multiple runs of the benchmark. Balancing the number of runs and the number of samples collected in each run allows us to increase the efficiency of the benchmarking experiment and achieve sufficient precision within given time.

References

[1] Buble, A.; L. Bulej; P. Tuma. 2003. “CORBA Benchmarking: A Course With Hidden Obstacles.” In

Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS 2003) Workshop on Performance Modeling, Evaluation and Optimization of Parallel and Distributed Systems (Nice, France, April 22-26). IEEE, Piscataway, NJ.

[2] Bulej, L.; T. Kalibera; P. Tuma. 2005. "Repeated Results Analysis for Middleware Regression Benchmarking." *Performance Evaluation 60*, No. 1-4, May: 345-358.

[3] Bulej, L.; T. Kalibera; P. Tuma. 2004. "Regression Benchmarking with Simple Middleware Benchmarks." In *Proceedings of IPCCC 2004 Workshop on Middleware Performance* (Phoenix, AZ, USA, April 15-17). IEEE, Piscataway, NJ, 771-776.

[4] Cecchet, E.; A. Chanda; S. Elnikety; J. Marguerite; W. Zwaenepoel. 2003. "Performance Comparison of Middleware Architectures for Generating Dynamic Web Content." In *Proceedings of the 4th ACM/IFIP/USENIX International Middleware Conference* (Rio de Janeiro, Brazil, June 16-20).

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

[6] Free Software Foundation. *The GNU Compiler Collection.* http://gcc.gnu.org.

[7] Frigo, M.; S.G. Johnson. *BenchFFT – A Program to Benchmark FFT Software.* http://www.fftw.org/benchfft.

[8] Giladi, R. and N. Ahituv. 1995. "SPEC as a Performance Evaluation Measure." *Computer 28*, No. 8, August: 33-42.

[9] Gu, D.; D. Verbrugge; E. Gagnon. 2004. "Code Layout as a Source of Noise in JVM Performance." In *Object-Oriented Programming, Systems, Languages, and Applications(OOPSLA 2004) Workshop on Middleware Benchmarking* (Vancouver, Canada, Oct. 24-28).

[10] Mayer, R. and O. Buneman. *FFT Benchmark.* ftp://ftp.nosc.mil/pub/aburto/fft.

[11] McCulloch, C.E. and S.R. Searle. 2000. *Generalized, Linear and Mixed Models.* Wiley-Interscience, New York, NY.

[12] *OVM Predictability and Performance Benchmarking.* 2004. http://www.ovmj.org/bench.

Biography

Tomas Kalibera is a doctoral student at the Department of Software Engineering, Faculty of Mathematics and Physics, Charles University in Prague, Czech Republic. He received his master degree in computer science from the Charles University in 2002. His primary interests are design of component systems and performance evaluation of middleware. He is a member of the Distributed Systems Research Group of Charles University.

Lubomir Bulej received his master degree in electrical engineering from the Czech Technical University, Prague, Czech Republic in 2002 and is currently pursuing doctoral degree at the Department of Software Engineering, Faculty of Mathematics and Physics, Charles University in Prague. Since 2002 he also holds a research assistant position at the Institute of Computer Science, Academy of Sciences of the Czech Republic. He is a member of the Distributed Systems Research Group of Charles University and his primary research interests include performance evaluation of middleware technologies and connectors in component-based software architectures.

Petr Tuma is a senior assistant professor with the Department of Software Engineering, Faculty of Mathematics and Physics, Charles University in Prague, Czech Republic. He received his master degree in electrical engineering from the Czech Technical University in 1994 and his doctoral degree in software systems from the Charles University in 1998. From 1998 to 1999, he worked as a researcher at INRIA Rennes in France. His primary interests include operating systems, component systems and middleware, with focus especially on design and performance evaluation. He is a member of the Distributed Systems Research Group of Charles University.

Chapter 5

Quality Assurance in Performance: Evaluating Mono Benchmark Results

Tomáš Kalibera,
Lubomír Bulej,
Petr Tůma

Contributed paper at **Second International Workshop on Software Quality (SOQUA 2005)** [2].

In *Quality of Software Architectures and Software Quality*,
published by Springer-Verlag,
LNCS 3712,
pages 271–288,
ISSN 0302-9743,
September 2005.

The original version is available electronically from the publisher's site at http://dx.doi.org/10.1007/11558569_20.

Quality Assurance in Performance: Evaluating Mono Benchmark Results

Tomas Kalibera[1], Lubomir Bulej[1,2], and Petr Tuma[1]

[1] Distributed Systems Research Group, Department of Software Engineering
Faculty of Mathematics and Physics, Charles University
Malostranske nam. 25, 118 00 Prague, Czech Republic
phone +420-221914267, fax +420-2219143232
{kalibera,bulej,tuma}@nenya.ms.mff.cuni.cz

[2] Institute of Computer Science, Czech Academy of Sciences
Pod Vodarenskou vezi 2, 182 07 Prague, Czech Republic
phone +420-266053831
bulej@cs.cas.cz

Abstract. Performance is an important aspect of software quality. To prevent performance degradation during software development, performance can be monitored and software modifications that damage performance can be reverted or optimized. Regression benchmarking provides means for an automated monitoring of performance, yielding a list of software modifications potentially associated with performance changes. We focus on locating individual modifications as causes of individual performance changes and present three methods that help narrow down the list of modifications potentially associated with a performance change. We illustrate the entire process on a real world project.

1 Introduction

The ever-increasing amount of available computing power is closely followed by increasing scale and complexity of software systems, which in turn causes increase in the size of development teams producing the software. For various reasons, be it a distributed development model, which makes communication and coordination difficult, or the Extreme Programming [1] approach, which prefers rewriting code and rigorous testing to detailed analysis of all possible requirements, the development of software puts more and more emphasis on quality assurance.

Both the distributed development and the Extreme Programming approaches depend on regular testing of all components of the developed software, which is often automated and performed either on a regular basis, such as every day, every week, or as soon as a change is introduced into the source code management system. This testing process is known as regression testing.

The current practice typically limits the testing to functional correctness and robustness of the code and neglects an important aspect of quality, which

is performance. Regression benchmarking fills the gap by extending the regression testing practice with automatic benchmarking and evaluation of software performance [2, 3].

Regression benchmarking requires the testing process to be fully automatic, which includes downloading and building the software, and executing the benchmarks in a robust environment. The execution environment must be able to handle most of the typical failure scenarios to allow running without systematic supervision. These requirements alone constitute a technical challenge, if only because the executed software is under development, where deadlocks, crashes, or infinite loops can be expected, but their occurrence cannot be easily predicted. Often, the same program repeatedly executed under (to a maximum possible and reasonable extent) identical conditions finishes with success in one case and with failure in another.

Many of these problems have been already solved for the purpose of regression testing, yet in case of regression benchmarking, the goal is also to minimize the influence of undesirable effects on the benchmark results. During benchmark execution, there should be no other users of the system, minimum number of concurrently running system services, minimum or preferably no network communication unrelated to the benchmark, and no substantial changes to the configuration of the system. Also, the monitoring of benchmarks must not be too intrusive, etc.

We have elaborated the requirements and described the design of a platform independent environment for automatic execution of benchmarks in [4]. At present, the benchmarking environment is under development.

As an experimental testbed, we have created a simple system for regression benchmarking of Mono [5], which is being developed by Novell as an open source implementation of Common Language Infrastructure specification [6], known as the .Net platform. The Mono implementation of CLI comprises a C# compiler, a virtual machine interpreting the Common Intermediate Language instructions, and the implementation of runtime classes.

Since August 2004, the system monitors the performance of daily development snapshots of Mono on the Linux/IA32 platform, using four benchmarks focused at .Net remoting and numeric computing. The results are updated on daily basis and publicly available on the web of the project [7]. The currently used benchmarks do not cover all the functionality of Mono, but rather serve as a test bed for developing methods for detecting performance changes and locating their causes.

Compared to the envisioned generic benchmarking environment, the system for benchmarking Mono is less universal, depends on the Unix/Linux environment, and is not distributed. Nevertheless, its operation is fully automatic and provides experience which is used to drive the design and implementation of the environment described in [4].

Software projects that employ regular and automated benchmarking, such as [8] or [9], typically focus on tracking the performance of the particular project and lack the automatic detection of performance changes. The automatic detec-

tion of performance changes is also missing in other related projects, such as the generic framework for automated testing [10] and the framework for execution and advanced analysis of functionality tests [11]. The foundations of a generic environment for automated benchmarking in grids are implemented in a discontinued project [12]. We are not aware of any other project that would support automatic detection of performance changes and location of their causes.

The rest of the paper has the following structure. Section 2 describes the methods for automatically detecting performance changes, while Section 3 deals with identifying the source code modifications causing the performance changes and provides an analysis of regressions identified by the regression testing system. Section 4 concludes the paper and provides an outlook on future work.

2 Automatic Detection of Regressions

Regression benchmarking requires automatic analysis of benchmark results to discover performance changes, be it performance regressions or improvements. The complexity of contemporary platforms and software causes the durations of operations measured by a benchmark to differ each time the operations are executed, making it impossible to discover performance changes simply by comparing the durations of the same operations in consecutive versions of software. The differences in operation durations often exhibit random character, which can originate for example in the physical processes of hardware initialization, or in the intentionally randomized algorithms such as generators of unique identifiers and subsequent hashing.

Typically, the operations measured by a benchmark are therefore repeated multiple times and the durations are averaged. The precision of such a result can be determined if the the durations are independent identically distributed random variables. Unfortunately, the requirements of independence and identical distribution are often violated. In Section 2.2, we describe several ways of processing the benchmark data which help to satisfy the requirements without distorting the results.

Another frequently overlooked effect of the complexity of contemporary platforms and software is the influence of random initial conditions on the durations of operations measured by a benchmark [13]. It is generally impossible to discover performance changes even by comparing the averaged durations of the same operations in consecutive versions of software, because the averaged durations differ each time the benchmark is executed. The difference is not related to the number of samples and therefore cannot be overcome by increasing the number of samples.

Regression benchmarking thus requires not only repeating the operations measured by the benchmark within the benchmark execution, but also repeating the execution of the benchmark within the benchmark experiment. The precision of the averaged durations can then be calculated as outlined in Section 2.1. Each execution of a benchmark during a benchmark experiment, however, increases

the overall cost of the experiment. That is mainly due to initialization and warm-up phases of a benchmark, which are costly in terms of time but collect no data. During the benchmark warm-up, operation durations can be influenced by transitory effects such as initialization of the tested software, operating system or hardware. Ignoring this fact in analysis can lead to incorrect results, as shown in [14].

The number of samples in a benchmark run and the number of benchmark executions during a benchmark experiment both contribute to the precision of the result of a benchmark experiment. The contribution of each of the components depends on the character of the developed software. Because the most costly factor of a benchmark run is the initialization and warm-up, our objective is to minimize the number of benchmark runs and maximize the number of samples collected in each run. From this naturally follows the incentive to determine the optimal number of samples that should be collected in a single benchmark run and beyond which increasing the number of samples does not contribute to the precision of the result anymore. After that, as explained in Section 2.1, we only need to determine the number of benchmark runs required to achieve the desired precision.

Knowing the precision of the result of a benchmark experiment is important so that a comparison of results that differ by less than their respective precision is not interpreted as a change in performance. Detection of performance changes in regression benchmarking can be carried out using the approach described in Section 2.3.

Digressing somewhat from the outlined approach of executing multiple measurements and collecting multiple samples, we can also consider reducing the variance of the samples by modifying the benchmark experiment in ways that remove the sources of variance. Intuitively, things such as device interrupts, scheduler events and background processes are all potential sources of variance that could be disabled, minimized or stopped. The sources of variance, however, may be not only difficult to identify [15, 16], but also inherent to the benchmark experiment and therefore impossible to remove. Coupled with the fact that the sources of variance would have to be identified and removed individually for each benchmark experiment, this suggests that sufficiently reducing the variance of the samples by modifying the benchmark experiment is generally impossible. For illustration, a checklist of precautions to be taken to minimize variance in micro-benchmarking in FreeBSD is given in [17]. The complexity of these precautions demonstrates the infeasibility of this approach.

It should also be pointed out that to verify whether the sources of variance were removed, executing multiple measurements and collecting multiple samples is necessary anyway. Finally, the relevance of the modified benchmark experiment to practice, where the sources of variance are present, may be questionable.

2.1 Statistics Behind the Scenes

We presume that the durations of operations measured by a benchmark in a run are random, independent and identically distributed, that the distributions of

the durations from different runs can differ in parameters, and that the mean values of the distributions from different runs are independent identically distributed random variables. The benchmark result is the average of all measured operation durations from all runs. We consider the benchmark precision to be the confidence with which the result estimates the mean value of the distribution. Specifically, we define the precision as the half width of the confidence interval for the mean, for a given fixed confidence level.

For each $j = 1..m$ as a benchmark run with $i = 1..n$ measurements, the random durations of operations R_{ji}, for $i = 1..n$ and fixed j, are independent identically distributed with a distribution that is described by the two traditional parameters: the mean and the variance. The conditional mean and variance of the distribution are $E(R_{j1}|\mu_j, \sigma_j^2) = \mu_j < \infty$, $var(R_{j1}|\mu_j, \sigma_j^2) = \sigma_j^2 < \infty$.

The means μ_j are independent identically distributed random variables for each $j = 1..m$ as a benchmark run, $E(\mu_1) = \mu < \infty$, $var(\mu_1) = \rho^2 < \infty$.

The result of a benchmark experiment is

$$\overline{R}_{ji} = \frac{1}{mn} \sum_{j=1}^{m} \sum_{i=1}^{n} R_{ji}$$

as an estimate of μ. Note that μ is also mean of R_{ji}, if we do not know the specific μ_j, since

$$E\left(R_{ji}\right) = E\left(E\left(R_{ji}|\mu_j\right)\right) = \mu.$$

From CLT, the distribution of $\overline{\mu}_j$ as an estimate of μ is asymptotically normal:

$$\frac{1}{m} \sum_{j=1}^{m} \mu_j = \overline{\mu}_j \sim N\left(\mu, \frac{\rho^2}{m}\right). \tag{1}$$

From CLT, the average M_j of operation durations from run j has asymptotically the normal distribution:

$$M_j = \frac{1}{n} \sum_{i=1}^{n} R_{ji}|\mu_j \sim N\left(\mu_j, \frac{\sigma_j^2}{n}\right).$$

From the properties of the normal distribution, it follows that

$$\frac{1}{m} \sum_{j=1}^{m} M_j = \overline{M}_j; \overline{M}_j|\overline{\mu}_j \sim N\left(\overline{\mu}_j, \frac{\sum_{j=1}^{m} \sigma_j^2}{nm^2}\right). \tag{2}$$

It can be shown that from (1), (2) and the known fact that the convolution of Gaussians is again a Gaussian, it follows that:

$$\overline{M}_j \sim N\left(\mu, \frac{\rho^2}{m} + \frac{\sum_{j=1}^{m} \sigma_j^2}{nm^2}\right). \tag{3}$$

The confidence interval for the estimate of μ can be constructed from (3). The result of a benchmark experiment therefore is $\overline{R}_{ji} = \overline{M}_j$ and with the probability $1 - \alpha$, the precision pr of the result value is

$$pr = u_{1-\frac{\alpha}{2}} \cdot \sqrt{\frac{\rho^2}{m} + \frac{\sum_{j=1}^{m} \sigma_j^2}{nm^2}}, \tag{4}$$

where u are quantiles of the standard normal distribution.

This result holds asymptotically for sufficiently large n and sufficiently large m, needed for CLT to apply. The unknown variance ρ^2 of the means μ_j can be approximated by the variance of the averages of samples from individual runs, which can be estimated using the S^2 estimate:

$$S_\rho^2 = \frac{1}{m-1} \sum_{j=1}^{m} \left[\left(\frac{1}{n} \sum_{i=1}^{n} R_{ji} \right) - \overline{R}_{ji} \right]^2.$$

The variance of the samples in a run σ_j^2 is still unknown. If the variance of the individual runs was the same, $\sigma_j^2 = \sigma^2$, we could estimate it as:

$$S_\sigma^2 = \frac{1}{m(n-1)} \sum_{j=1}^{m} \sum_{i=1}^{n} \left(R_{ji} - \frac{1}{n} \sum_{i=1}^{n} R_{ji} \right)^2$$

to get the precision of the result value:

$$pr_\sigma = u_{1-\frac{\alpha}{2}} \cdot \sqrt{\frac{nS_\rho^2 + S_\sigma^2}{mn}}.$$

If we know the maximum variance σ_{max}^2 of the samples in a run, we can use a similar approach and estimate the upper bound of the precision pr (lowest possible precision).

Note also that the formula for pr does not rely on the individual values of variance, but only on the average variance. We can therefore use the formulas for the precision pr_σ and the variance estimate S_σ^2 for large m, as implied by the Weak Law of Large Numbers, except for the precision of the estimate itself, which is not included in the formula for pr_σ.

From (4) it follows that increasing the number of runs m always improves the precision of the result, while increasing the number of measurements in a run n improves the precision only to a certain limit. In practice, each benchmark run has to invoke the measured operation w times discarding the results to warm-up, prior to measuring n operation durations. Usually there is only a limited time c (cost) for each experiment, which can be expressed as the number of all invoked operations: $c = (w+n)m$. The question is how to choose m and n to achieve the best precision pr for a given c. From (4) we can derive that the optimal number of (non-warmup) measurements in a benchmark run is

$$n_{opt} = \sqrt{\frac{wS_\sigma^2}{S_\rho^2}}.$$

The optimal strategy to achieve the desired precision of the result is to first increase n up to n_{opt}, and only then increase m. However, determining n_{opt} requires the knowledge of the estimates S_σ^2 and S_ρ^2. An experiment can therefore be split into two parts – in the first part, a sufficient number of runs and number of measurements in a run are chosen to get good estimates S_σ^2 and S_ρ^2. The benchmark precision is also calculated, and when it is not sufficient, n_{opt} is calculated using the variance estimates and a second part of the experiment is performed which only invokes n_{opt} operations in each run.

2.2 Handling the Auto-Dependence and Outliers

When determining the result of a benchmark experiment and its precision as described in Section 2.1, the key assumption is that the samples of the durations of operations in a single benchmark run are independent and identically distributed. Our experience suggests that these assumptions do not generally hold. If the benchmark data is found to violate the assumptions, a simple transformation can be often found that will remedy the situation without significant impact on the results.

The assumption of independence can be easily verified using lag plots. The lag plot is generally used for visually inspecting auto-dependence in time series by plotting the original data against lagged version of themselves to check for any identifiable structure in the plot. For example, in one of the benchmarks used by the Mono regression benchmarking system, the data contained four clearly visible clusters. Inspection using lag plot suggested that the values from the individual clusters were systematically interleaved. We have numbered the clusters and transformed the original data so that each value was replaced by the number of the cluster it belonged to, which made the interleaving pattern in the data obvious and confirmed the suspected violation of the independence assumption.

We solve the auto-dependence problem by resampling the original data. For each benchmark run, we generate a reduced subset from the original data using sampling with replacement. The use of sampling with replacement is important, because it eliminates a potential dependency on the number of samples in case of systematic interleaving of values from different clusters. From the resampled data, we can calculate the estimates of mean and the variance of the distribution of the values collected by the benchmark, as outlined in Section 2.1.

The assumption that the samples come from the same distribution is typically violated in presence of outliers in the collected data. In benchmarking, the nature of the outliers is such that under certain circumstances, the duration of an operation can be as much as several orders of magnitude longer than in most other cases. The exact circumstances leading to the occurrence of outliers are

difficult to identify, but some of the outliers can have a plausible explanation [3]. Even so, explicitly removing the outliers from the data is typically impossible, because we do not have enough information to discriminate between valid data and outliers.

The outliers can significantly influence the results, especially when they are based on sample average or variance. The preferred solution is to use robust statistics such as the median or median absolute deviation in place of the traditional but fragile sample average or sample variance [14, 3]. The main drawback of using robust statistics lies with the fact that the analysis of their precision is difficult compared to sample average or variance.

To handle outliers and auto-dependence in the benchmark data from the Mono project, we use a combination of resampling and robust statistics, but still report sample average or variance as the result of a benchmark run, which makes the analysis of the precision easier.

If there are any outliers in the original data, they will likely be also in the resampled subset, which means that the sample average or variance will be influenced by the outliers. We therefore generate a high number of reduced subsets from the original data, and compute the required estimators using the generated subsets. From the resulting set of sample averages or sample variances, we select the median to obtain a single estimate of the mean or the variance from a single benchmark run. These values still have the nature of the original sample mean or variance, because they were computed using the data from one of the subsets.

2.3 Detecting and Quantifying the Changes

Detecting changes in performance between different version of the software in development requires comparing benchmark results for the two versions. As outlined in Section 2.1, processing the benchmark data in order to carry out the comparison is not so trivial a task.

Comparing performance of two versions of the same software requires comparing the mean values of the respective distributions of samples obtained by running the benchmarks. Since the true means are unknown, the comparison has to be carried out on their estimates and take into account the precision of the estimates. Benchmarks provide the estimate of the distribution mean as a grand average of average durations of an operation execution in multiple benchmark runs and the precision of the estimate as a confidence interval for the grand average.

When using confidence intervals to compare unpaired samples from the Normal distribution, Jain [18] suggests a method which considers the distribution means different when the confidence intervals of sample averages do not overlap, and equal when the center of either of the confidence intervals falls in the other. In the last case, when the confidence intervals overlap, the decision is based on the t-test.

For the purpose of regression benchmarking, the algorithm for detecting changes in performance must avoid or minimize the amount of false alarms,

otherwise the whole system would be useless. Therefore, our system only reports performance change if the confidence intervals do not overlap.

The developers, as the users of the regression benchmarking system would be mainly interested in performance regressions resulting from an inappropriate modification to the source code. Not all the changes in performance are caused by inappropriate modifications though.

Changing the name of a variable or an identifier may change the binary layout of the executable, thereby changing the layout of the executable in memory, which may result in a performance change [16]. While not necessarily true in general, the cosmetic code modifications such as the renaming of identifiers can be assumed to have smaller impact on performance than a modification introducing a real performance regression into the code.

In addition to detecting a performance change, it is therefore desirable to assess the magnitude of the change, so that the developers can focus on performance regressions they consider important. We report the magnitude of a performance change as a ratio of the distance between the centers of the confidence intervals for the older and the newer version and the center of the confidence interval for the older version.

2.4 Automatically Detected Regressions

The method for detecting performance changes has been applied on the results of TCP Ping, HTTP Ping, FFT Scimark and Scimark benchmarks compiled and executed under Mono.

The TCP Ping and HTTP Ping benchmarks measure the duration of a remote method execution using .Net remoting, the client and server being different processes running on the same machine. The remote method is invoked with a string argument, which it returns unchanged. The time is measured at the client side and includes the time spent processing the request on the server side. The TCP Ping uses the TCP Channels, while the HTTP Ping uses the HTTP Channels of the .NET remoting infrastructure.

The Scimark benchmark is the C# version of the Scimark2 numerical benchmark [19, 20]. The benchmark evaluates the performance of several numerical algorithms, such such as Fast Fourier Transform, Jacobi Successive Over-Relaxation, Monte Carlo Integration, Sparse Matrix Multiplication, and Dense Matrix Factorization.

The original benchmark does not measure the computations repeatedly, which means that the results come from the warm-up phase of a benchmark and can be influenced by initialization noise [14]. Based on the original benchmark, we have created the FFT benchmark, which only contains the code for computing the Fast Fourier Transform. The modified benchmark performs the execution repeatedly and can provide data collected after the warm-up phase has passed, which is more suitable for regression benchmarking.

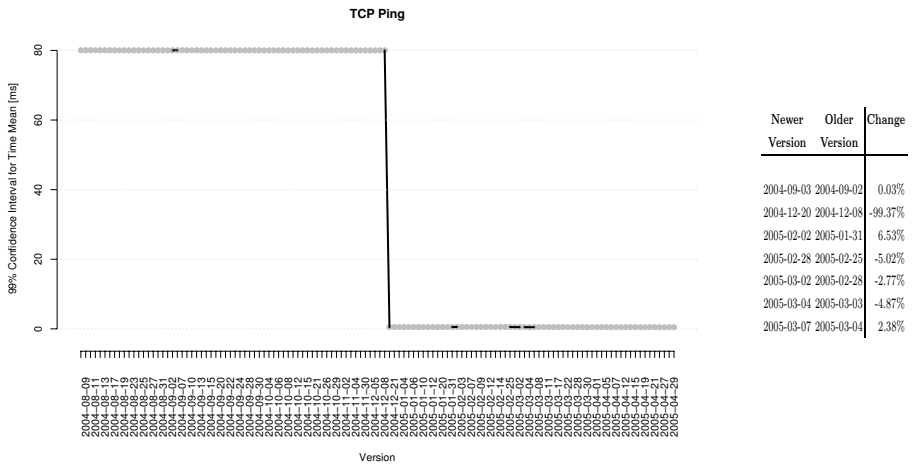
All benchmarks are run on two configurations of Mono. The first configuration has only the default optimizations turned on in the just-in-time compiler,

the other has all optimizations turned on. The performance has been monitored from August 2004 till April 2005.

Within that period, the regression benchmarking system has found a number of performance changes. The most notable change was a 99% improvement in performance of the TCP Ping benchmark, which occurred in a development version of Mono from December 20, 2004. The performance changes in the TCP Ping benchmark are shown in Figure 1 and summarized in the table beside the plot.

The detected performance changes are marked by bold black lines, confidence intervals are marked by grey lines. The table shows the magnitude of the changes, along with dates of versions between which a performance change occurred (positive number represents an increase in the duration of a method and thus a performance regression).

A number of significant performance changes has been identified using the HTTP Ping benchmark and the FFT benchmark, see Figure 2 and Figure 3 respectively. The modifications of source code, which cause some of the more significant changes are analyzed in Section 3.

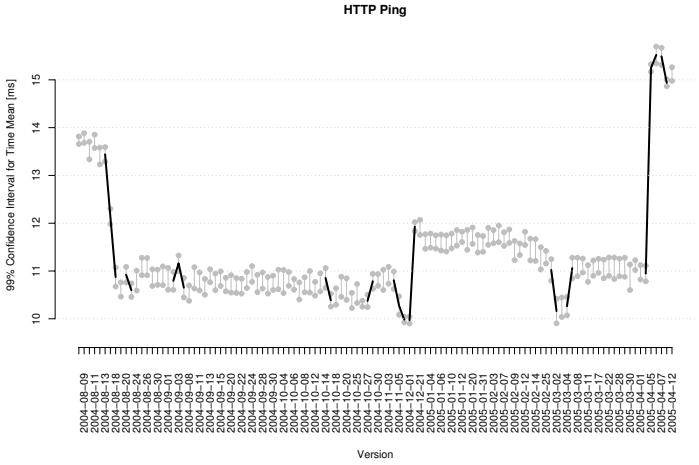


Newer Version	Older Version	Change
2004-09-03	2004-09-02	0.03%
2004-12-20	2004-12-08	-99.37%
2005-02-02	2005-01-31	6.53%
2005-02-28	2005-02-25	-5.02%
2005-03-02	2005-02-28	-2.77%
2005-03-04	2005-03-03	-4.87%
2005-03-07	2005-03-04	2.38%

Fig. 1. Changes detected in TCP Ping mean response time

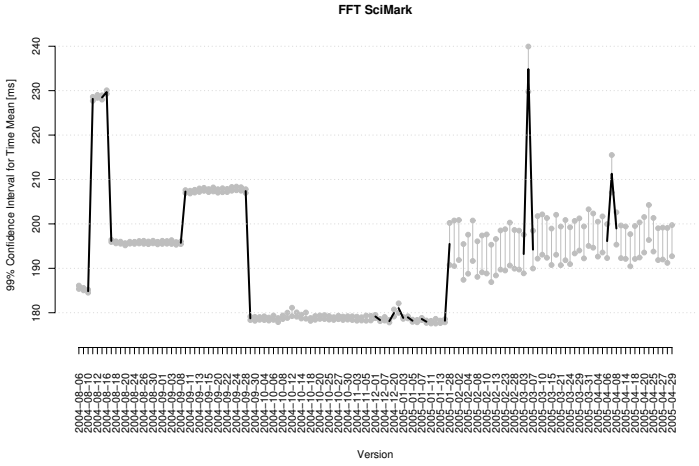
3 Analysis of Discovered Regressions

Regression benchmarking uses the method for detecting changes in performance from Section 2 on daily versions of software subject to modifications to yield a



Newer Version	Older Version	Change
2004-08-17	2004-08-13	-9.67%
2004-08-18	2004-08-17	-10.44%
2004-08-23	2004-08-20	-2.95%
2004-09-03	2004-09-02	3.4%
2004-09-07	2004-09-03	-4.57%
2004-10-15	2004-10-14	-4.29%
2004-10-29	2004-10-27	3.96%
2004-11-05	2004-11-04	-4.9%
2004-11-30	2004-11-05	-2.87%
2004-12-20	2004-12-01	19.64%
2005-03-02	2005-02-28	-7.81%
2005-03-07	2005-03-04	7.77%
2005-04-05	2005-04-04	39.29%
2005-04-06	2005-04-05	1.77%
2005-04-08	2005-04-07	-3.58%

Fig. 2. Changes detected in HTTP Ping mean response time



Newer Version	Older Version	Change
2004-08-11	2004-08-10	23.43%
2004-08-16	2004-08-13	0.53%
2004-08-17	2004-08-16	-14.59%
2004-09-10	2004-09-08	5.9%
2004-09-29	2004-09-28	-13.81%
2004-12-05	2004-12-01	-0.44%
2004-12-20	2004-12-08	1.03%
2005-01-03	2004-12-21	-1.22%
2005-01-05	2005-01-04	-0.43%
2005-01-10	2005-01-07	-0.34%
2005-01-28	2005-01-20	9.71%
2005-03-04	2005-03-03	21.55%
2005-03-07	2005-03-04	-17.31%
2005-04-07	2005-04-06	7.73%
2005-04-08	2005-04-07	-5.83%

Fig. 3. Changes detected in FFT mean response time

list of versions of software where the performance changes first exhibited themselves. Performance changes are either due to modifications that were done with the intent of improving performance, or due to modifications that were done for other reasons and changed performance inadvertently. In the former case, the modifications that caused the performance changes are known and the list can merely confirm them. In the latter case, the modifications that caused the performance changes are not known and the list can help locate them.

It is necessary to correlate the performance changes with the modifications, so that the modifications can be reviewed to determine whether to revert them, to optimize them, or to keep them. Locating a modification that caused a performance change is, however, difficult in general.

An extreme example of a modification that is difficult to locate is a change of an identifier that leads to a change of the process memory layout, which leads to a change in the number of cache collisions, causing a significant performance change [16]. Locating such a modification is not only difficult, but also useless, because the performance change will be tied to a specific platform and a specific benchmark and not reliably reversible.

Locating a modification that caused a performance change is also difficult in large and quickly evolving software projects, where the number of benchmarks that cover various features of the software will range in tens or hundreds, and where the number of modifications between consecutive versions will be high.

Achieving a fully automated correlation of the performance changes with the modifications seems unlikely. Given that the correlation is indispensable for practical usability of regression benchmarking, we focus on devising methods that aid in a partially automated correlation. Specific methods that narrow down the list of modifications potentially associated with a performance change are described in sections 3.1, 3.2 and 3.3.

3.1 Modifications as Differences in Sources

An obvious starting point for correlating a performance change with a modification is the list of modifications between the version of software where the performance change first exhibited itself and the immediately preceding version. This list is readily available as an output of version control tools such as Concurrent Versions System or Subversion or text comparison utilities such as diff.

The list of modifications between consecutive versions is often large. Figure 4 shows the size of modifications between consecutive versions of Mono, with performance changes detected using the method from Section 2 denoted by triangles pointing upwards for regressions and downwards for improvements. The size of modifications is calculated as a sum of added, deleted and changed source lines. The scale of the graph is logarithmic, namely each displayed value is the decadic logarithm of the modification size plus one.

The version control tools or text comparison utilities typically output a list of physical modifications to source files and blocks of source lines. The knowl-

edge of physical modifications is less useful as it puts together multiple logically unrelated modifications that were done in parallel. More useful is the knowledge of logical modifications as groups of physical modifications done with a specific intent, such as adding a feature or fixing a bug. The knowledge of the logical modifications, ideally with their intent, helps correlating the performance changes with the modifications.

The knowledge of logical modifications can be distilled from the output of version control tools when the versions are annotated by a change log. A change log entry typically contains a description of the intent of modifications. Given that a change log entry is associated with a commit of a new version, this requires maintaining a policy of a separate commit for each logical modification.

3.2 Statically Tracking Modifications

In large and quickly evolving software projects, the number of modifications between consecutive versions will be high, even when logical rather than physical modifications are considered. To further narrow down the list of modifications potentially associated with a performance change, we can use the fact that a benchmark that covers a specific feature of a software typically uses only the part of the software that provides the feature. In general, this allows us to consider only the modifications that can influence the result of the benchmark. Given that determining whether a modification can influence the result of a benchmark is difficult, we consider only the modifications that influence the part of the software used by the benchmark instead.

Determining the modifications that influence the part of a software used by a benchmark begins by obtaining the list of the called functions. This can be done either by analyzing the sources of the benchmark and the software, or by running the benchmark and collecting the list of the called functions by a debugger or a profiler. The called functions are then associated with the source files that implement them, and only the modifications that touch these source files are considered.

Alternatively, only the modifications that touch the called functions could be considered, which would further narrow down the list of modifications potentially associated with a performance change.

Applying this method in case of Mono is complicated, because the result of a benchmark depends not only on the application libraries, but also on the virtual machine and the compiler, which are subject to modifications. Obtaining the list of called functions is easiest for the application libraries, where the tracing function of the virtual machine can be used. Obtaining the same list for the virtual machine and the compiler is more difficult, because a debugger or a profiler has to be used. This makes regression benchmarking more time consuming. Furthermore, the usefulness of the list for the functions of the compiler is rather limited, because of the indirect nature of the influence of a compiler function on the result of the benchmark.

In case of the HTTP Ping and TCP Ping benchmarks, determining the modifications that influence the parts of Mono used by the benchmarks narrowed down the list of modifications significantly. For a performance regression of almost 40% in the HTTP Ping benchmark between Mono versions from April 4 and April 5, the list of modifications contained 39 physical modifications of the application libraries, but only 7 physical modifications belonging to 4 logical modifications influenced the parts of Mono used by the benchmark.

Figure 5 shows how considering only the modifications that can influence the result of a benchmark by means discussed above narrows down the size of modifications between consecutive versions of Mono from figure 4 for the TCP Ping benchmark. Figure 5 also demonstrates that the magnitude of detected performance changes needs not to follow the size of the modifications, even when only modifications that can potentially influence performance are taken into account.

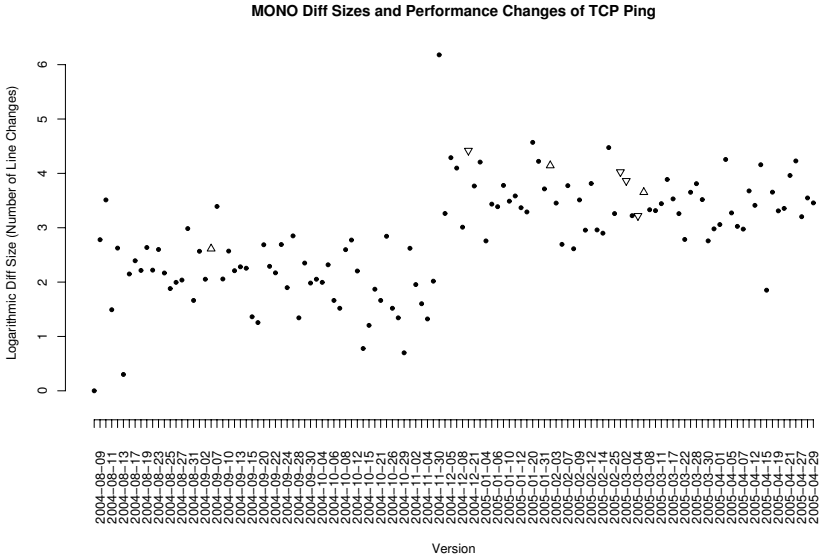


Fig. 4. Diff sizes of all sources with performance changes detected by TCP Ping benchmark

Even when considering only the modifications that can influence the result of a benchmark, it can be difficult to determine which of the logical modifications potentially associated with a performance change is the one that caused the change. When the descriptions of the intent of modifications from the change

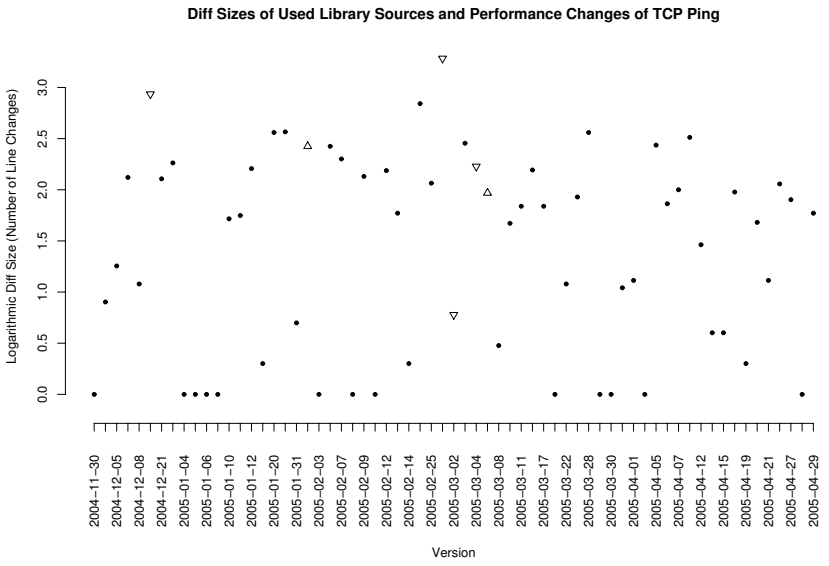


Fig. 5. Diff sizes of sources used by TCP Ping benchmark with detected performance changes

log do not help, additional benchmark experiments can be used for correlating the performance change with one of the modifications.

3.3 Experimentally Tracking Modifications

Given a list of logical modifications potentially associated with a performance change and the version of software where the performance change first exhibited itself, auxiliary versions of software can be created by reverting the modifications one by one, or by applying the modifications one by one to the immediately preceding version. Regression benchmarking of the auxiliary versions can help determine which of the list of logical modifications is the one that caused the performance change.

Applying this method consistently, regression benchmarking can consider all logical modifications on daily versions of software one by one. While such an approach can be automated, it makes regression benchmarking more time consuming.

Alternatively, the modification most likely to be the one that caused the performance change can be selected manually. A pair of auxiliary versions can then be created, one by reverting the modification from the version of software where the performance change first exhibited itself, another by applying the

modification to the immediately preceding version. Regression benchmarking of the auxiliary versions can help confirm or reject the choice of modification.

This method was used on the performance regression of almost 40% in the HTTP Ping benchmark between Mono versions from April 4 and April 5. The list of modifications potentially associated with the regression was first narrowed down to 4 logical modifications. Regression benchmarking has shown that the modification that caused the regression was a rewrite of the function for converting the case of a string.

This method was also used on the performance regression of almost 24% in the FFT benchmark between Mono versions from August 10 and August 11. The FFT benchmark does not use the application libraries and the list of changes potentially associated with the regression therefore concerned only the virtual machine and the compiler. Regression benchmarking has shown that the modification that caused the regression was introduction of a particular loop optimization into the set of default optimizations performed by the just-in-time compiler.

In case of the performance improvement of almost 17% in the FFT benchmark between Mono versions from March 4 and March 7, regression benchmarking has shown that the modification that caused the improvement was a rewrite of the function for passing control between native and managed code and confirmed the improvement.

Finally, by far the biggest performance improvement of almost 99% in the TCP Ping benchmark between Mono versions from December 8 and December 20 was caused by a rewrite of the communication code to pass data in chunks rather than byte by byte. Regression benchmarking confirmed the improvement. Before this particular modification, the HTTP Ping benchmark reported smaller duration of operations than the TCP Ping benchmark, which is intuitively a suspect result. Eventually, similar dependencies between benchmark results could also be described and tested.

It should be noted that while regression benchmarking can detect performance regressions, it cannot detect performance problems that have been part of the software prior to regression benchmarking. The passing of data byte by byte rather than by chunks is an example of such a situation.

We can consider other methods that narrow down the list of modifications potentially associated with a performance change. In case of Mono, these include separate regression benchmarking of modifications that influence the compiler, the modifications that influence the virtual machine, and the modifications that influence the application libraries. These methods would apply to any other platform with a compiler, virtual machine and application libraries, such as the Java platform.

As described, the method of using logical modifications does not depend on the software, but only on the versioning system. Although the implementation of the method of tracing calls to application libraries and then locating sources of the called methods is specific to the Mono platform, it can be implemented for other platforms that use dynamic linking or have debuggers and debug in-

formation in files. The implementation is then only specific to the platform the tested software uses.

A challenge for future work on regression benchmarking lies in automating the methods from sections 3.1, 3.2 and 3.3, so that performance changes can be routinely correlated with modifications.

4 Conclusion

We have developed a regression benchmarking environment that automatically and reliably detects performance changes in software. The features of the environment include a fully automated download, building and benchmarking of new versions of software and statistically sound analysis of the benchmark results. The environment is used to monitor daily versions of Mono, the open source implementation of the .Net platform. From August 2004 to April 2005, the environment has detected a number of performance changes, 15 of those exceeding 10%, and 6 of those exceeding 20% of total performance.

The regression benchmarking environment makes the results of the analysis of the benchmark results available on the web [7], giving the developers of Mono the ability to check the impact of modifications on the performance of the daily versions. The developers of Mono have reacted positively especially to the confirmation of modifications done with the intent of improving performance. Our effort focuses on locating the modifications that caused inadvertent performance changes, which currently requires a good knowledge of the software and therefore an attention of the developers to be successful.

In sections 3.1, 3.2 and 3.3, we have proposed three methods that help narrow down the list of modifications potentially associated with a performance change. The three methods were applied on the daily versions of Mono to locate modifications that caused 5 out of 15 performance changes exceeding 10%, 4 out of 6 of those exceeding 20% of total performance. While these methods have been tested on the Mono platform, they can be implemented for other platforms as well. They specifically do not depend on the tested software itself.

Future work on regression benchmarking includes extending the methods from sections 3.1, 3.2 and 3.3, as well as making the methods automated, so that correlating performance changes with modifications is less demanding. This is necessary to help regression benchmarking achieve the same status as regression testing in the software quality assurance process.

Acknowledgement. The authors would like to express their thanks to Jaromir Antoch, Alena Koubkova and Tomas Ostatnicky for their help with mathematical statistics. This work was partially supported by the Grant Agency of the Czech Republic projects 201/03/0911 and 201/05/H014.

References

1. Jeffries, R.E., Anderson, A., Hendrickson, C.: Extreme Programming Installed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2000)

2. Bulej, L., Kalibera, T., Tuma, P.: Repeated results analysis for middleware regression benchmarking. Performance Evaluation **60** (2005) 345–358
3. Bulej, L., Kalibera, T., Tuma, P.: Regression benchmarking with simple middleware benchmarks. In Hassanein, H., Olivier, R.L., Richard, G.G., Wilson, L.L., eds.: International Workshop on Middleware Performance, IPCCC 2004. (2004) 771–776
4. Kalibera, T., Bulej, L., Tuma, P.: Generic environment for full automation of benchmarking. In Beydeda, S., Gruhn, V., Mayer, J., Reussner, R., Schweiggert, F., eds.: SOQUA/TECOS. Volume 58 of LNI., GI (2004) 125–132
5. Novell, Inc.: The Mono Project. http://www.mono-project.com (2005)
6. ECMA: ECMA-335: Common Language Infrastructure (CLI). ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland (2002)
7. Distributed Systems Research Group: Mono regression benchmarking. http://-nenya.ms.mff.cuni.cz/projects/mono (2005)
8. DOC Group: TAO performance scoreboard. http://www.dre.vanderbilt.edu/-stats/performance.shtml (2005)
9. Prochazka, M., Madan, A., Vitek, J., Liu, W.: RTJBench: A Real-Time Java Benchmarking Framework. In: Component And Middleware Performance Workshop, OOPSLA 2004. (2004)
10. Dillenseger, B., Cecchet, E.: CLIF is a Load Injection Framework. In: Workshop on Middleware Benchmarking: Approaches, Results, Experiences, OOPSLA 2003. (2003)
11. Memon, A.M., Porter, A.A., Yilmaz, C., Nagarajan, A., Schmidt, D.C., Natarajan, B.: Skoll: Distributed continuous quality assurance. In: ICSE, IEEE Computer Society (2004) 459–468
12. Courson, M., Mink, A., Marçais, G., Traverse, B.: An automated benchmarking toolset. In Bubak, M., Afsarmanesh, H., Williams, R., Hertzberger, L.O., eds.: HPCN Europe. Volume 1823 of Lecture Notes in Computer Science., Springer (2000) 497–506
13. Kalibera, T., Bulej, L., Tuma, P.: Benchmark precision and random initial state. In: accepted for 2005 International Symposium on Performance Evaluation of Computer and Telecommunications Systems (SPECTS 2005). (2005)
14. Buble, A., Bulej, L., Tuma, P.: CORBA benchmarking: A course with hidden obstacles. In: IPDPS, IEEE Computer Society (2003) 279
15. Hauswirth, M., Sweeney, P., Diwan, A., Hind, M.: The need for a whole-system view of performance. In: Component And Middleware Performance workshop, OOPSLA 2004. (2004)
16. Gu, D., Verbrugge, C., Gagnon, E.: Code layout as a source of noise in JVM performance. In: Component And Middleware Performance Workshop, OOPSLA 2004. (2004)
17. The FreeBSD Documentation Project: FreeBSD Developers' Handbook. http://-www.freebsd.org/doc/en/books/developers-handbook (2005)
18. Jain, R.: The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling. Wiley-Interscience, New York, NY, USA (1991)
19. Re, C., Vogels, W.: SciMark – C#. http://rotor.cs.cornell.edu/SciMark/ (2004)
20. Pozo, R., Miller, B.: SciMark 2.0 benchmark. http://math.nist.gov/scimark2/ (2005)

Chapter 6

Automated Detection of Performance Regressions: The Mono Experience

Tomáš Kalibera,
Lubomír Bulej,
Petr Tůma

Contributed paper at **13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2005)** [1], acceptance rate 30%.

In conference proceedings,
published by IEEE,
pages 183–190,
ISSN 1526-7539,
September 2005.

The original version is available electronically from the publisher's site at http://doi.ieeecomputersociety.org/10.1109/MASCOT.2005.18.

Automated Detection of Performance Regressions: The Mono Experience

Tomas Kalibera[1] Lubomir Bulej[1,2] Petr Tuma[1]

[1]*Distributed Systems Research Group, Department of Software Engineering*
Faculty of Mathematics and Physics, Charles University
Malostranske nam. 25, 118 00 Prague, Czech Republic
phone +420-221914267, fax +420-2219143232

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

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

Abstract

Engineering a large software project involves tracking the impact of development and maintenance changes on the software performance. An approach for tracking the impact is regression benchmarking, which involves automated benchmarking and evaluation of performance at regular intervals. Regression benchmarking must tackle the nondeterminism inherent to contemporary computer systems and execution environments and the impact of the nondeterminism on the results. On the example of a fully automated regression benchmarking environment for the Mono open-source project, we show how the problems associated with nondeterminism can be tackled using statistical methods.

1 Introduction

The increase in scale and complexity of software, as well as the related increase in size of the development teams, puts a growing emphasis on the process of quality assurance. Indeed, continuous quality assurance is part of Extreme Programming [8] and many distributed development models, which rely on regular testing of all components of the software. The process of testing is often automated and performed either in given time intervals or whenever changes are introduced. This is known as regression testing.

The current practice of regression testing typically limits the testing to correctness and robustness of the software. Another important quality aspect, namely performance, is often neglected. Regression benchmarking addresses this gap by extending the regression testing to benchmarking and evaluation of software performance [3, 2].

In an analogy to regression testing, regression benchmarking must be fully automated. This requirement includes automated downloading and building of the software and the benchmarks, as well as automated executing of the benchmarks in a robust environment that handles typical failure scenarios without supervision. This alone is a technical challenge, if only because the software is under development and therefore prone to exhibiting bugs, crashes, or ending up in a deadlock or an infinite loop.

While many of these problems have already been solved in regression testing, regression benchmarking requires extending the solutions to include minimizing any undesirable influence on the results. During benchmarking, the activity of unrelated system services, the amount of unrelated network communication, and the scope of system configuration changes should all be minimized.

Importantly, regression benchmarking also requires an automated analysis of the results to discover performance changes. The discovery of performance changes is made difficult by the complexity of contemporary platforms and software, which causes the durations of the operations measured by a benchmark to differ each time the operations are executed. Because of this, it is not possible to discover performance changes from one version of the software to another simply by comparing the durations of the same operations in the two versions.

Typically, the measured operations are therefore repeated multiple times and the durations are averaged. When the durations can be assumed to be independent identically distributed random variables, the precision of the averaged result can be determined. The knowledge of the precision is necessary so that a comparison of results that differ by

less than their respective precisions is not interpreted as a performance change. Unfortunately, the requirements of independence and identical distribution are often violated. In Section 4, we describe a method of processing the collected data that overcomes the problem of the violated requirements.

Additionally, the nondeterminism inherent to contemporary computer systems and execution environments is reflected in the form of random initial conditions that influence the durations of the operations measured by a benchmark [9]. The influence of the random initial conditions makes the averaged durations differ each time the benchmark is executed. This difference makes it generally impossible to discover performance changes even by comparing the averaged durations of the same operations in two versions of software. Furthermore, the difference is unrelated to the number of durations that make up the averages and therefore cannot be avoided by repeating the measured operations more times. In [9], we also show that the difference cannot be avoided by simulation or by executing the benchmark immediately after system initialization. In Section 2, we show how to quantify the influence of the random initial conditions.

To summarize, regression benchmarking requires not only repeating the operations measured by a benchmark within the benchmark, but also repeating the execution of the benchmark within the benchmark experiment. The precision of the averaged durations can then be calculated even when the requirements of independence and identical distribution are violated and the influence of the random initial conditions is present, as outlined in Sections 3 and 4. Because of the cost of repeating the execution of the benchmark, however, it is necessary to determine the optimum number of benchmark runs and the optimum number of measurements in a run, as also explained in Section 3.

To verify the applicability of the methods described in the paper, we have created an environment for regression benchmarking of Mono [12]. Mono is being developed by Novell as an open-source implementation of the Common Language Infrastructure specification [5], also known as the .Net platform. The Mono implementation of CLI comprises a C# compiler, a virtual machine interpreting the Common Intermediate Language instructions, and the implementation of runtime classes.

Since August 2004, the environment monitors the performance of daily development snapshots of Mono on four benchmarks focused at numerical calculations and the mechanism of .Net Remoting, which implements remote method invocation. Continuously updated results are publicly available on the web of the project [4].

The structure of the paper is as follows. The analysis and the quantification of the impact of random initial conditions is in Section 2. A method of calculating the precision of the averaged durations influenced by random initial conditions is presented in Section 3. In Section 4, the method of calculating the precision is extended to cope with a violation of the requirements of independence and identical distribution. Section 5 explains how the knowledge of the precision is used to detect performance changes. Finally, Section 6 provides details on applying the methods on Mono in the framework of the Mono Regression Benchmarking Project. Section 7 concludes the paper.

2 Random Initial Conditions of Benchmarks

In contemporary systems, the duration of operations measured by a benchmark depends on a wide spectrum of factors. Within the spectrum, classes of factors can be distinguished depending on when the influence of the factor changes. First is the class of factors that change for each individual operation. Second is the class of factors that stay the same for all operations measured within a single benchmark process, because they depend on random initial conditions of the process. On some systems, there is even a class of factors that stay the same for all operations of all benchmark processes run using the same benchmark binary image. All these classes of factors are analyzed and evaluated in [9].

The impact of random initial conditions of a benchmark process on benchmark results is illustrated by Figure 1. The graph shows the results of the FFT benchmark, which calculates the Fast Fourier Transform. The operation measured by the benchmark is a pair of forward and inverse transformations of a constant vector. The FFT benchmark is based on the SciMark2 benchmark [14, 13].

The same benchmark has been run repeatedly. Each run of the benchmark has measured the same operation repeatedly. The graph in Figure 1 plots the operation times on the vertical axis and the sequential index of the measurement on the horizontal axis, with the measurements from individual runs separated by vertical lines. The graph shows that while the durations from the same run typically differ from each other only in units of percents, the durations from different runs of the same benchmark can differ from each other even in tens of percents. The difference between the durations from the same run illustrates the existence of the influencing factors that change when the benchmark is running. The difference between the durations from different runs illustrates the existence of the influencing factors that do not change when the benchmark is running but still differ every time the benchmark runs.

The graph also shows that the durations from the same run yield only a small number of different operation times, which results in the operation times being grouped in clusters. This effect is discussed in section 4.

The degree of influence of the random initial conditions

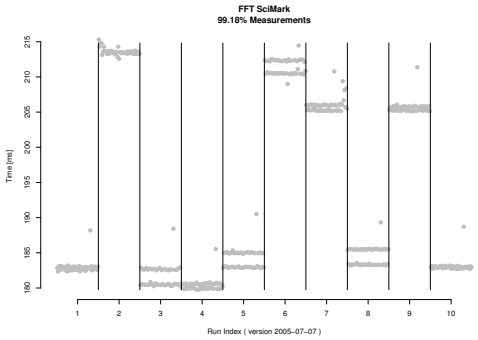


Figure 1. Durations of the same FFT computation in several benchmark runs.

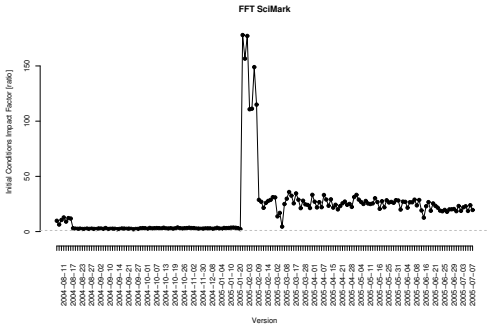


Figure 2. Impact factors of initial conditions for FFT benchmark in different Mono versions.

on the duration of operations depends on the specific platform and the specific benchmark. In [9], we have introduced an impact factor as a metric of the degree of influence of the random initial conditions. The impact factor is defined as a ratio of the standard deviation of durations from different runs to the standard deviation of durations from the same run and is estimated using simulation as described in [9].

A value of the impact factor that is close to 1 suggests a negligible influence of the random initial conditions on the duration of operations. The larger the value of the impact factor, the more the durations from different runs differ than the durations from the same run. Figure 2 shows the values of the impact factor for the FFT benchmark for daily versions of Mono developed between August 2004 and June 2005. Especially in February 2005, the influence of the random initial conditions was significant, as is indicated by the values of the impact factor in the order of tens to hundreds.

In contrast to the FFT benchmark, the values of the impact factor for the HTTP Ping benchmark suggest that the durations from different runs differ only twice or three times as much as the durations from the same run. The HTTP Ping benchmark measures the time it takes to invoke a remote method over an HTTP channel. The input argument of the method is a short string constant, the output argument of the method is the same string. The plots showing impact factors for the HTTP Ping and other Mono benchmarks are available on the web [4].

A value of the impact factor that is close to 1 indicates that a representative set of durations can be obtained even from a small number of runs of the same benchmark. A large value of the impact factor indicates that a large number of runs, rather than a large number of measured durations in a run, is needed to obtain a representative set of durations, as is the case with the FFT benchmark. This line of reasoning is made precise and formalized in section 3.

3 Benchmark Precision

As shown in section 2, to obtain a representative set of operation durations, it is necessary not only to repeat the operations measured by the benchmark within a single run of the benchmark, but also to run the benchmark repeatedly. The factors impacting the benchmark results are often unpredictable and random, covering for example random initialization in hardware or intentionally randomized algorithms in the application or the operating system. Consequently, each benchmark experiment consisting of multiple runs measuring multiple operation durations gives random results. We expect the distribution of the results to have a mean and to be well characterized by the mean. To simplify comparison, we calculate a single result value from each benchmark experiment, which is the average of all operation durations.

For a trustworthy detection of performance changes for the purpose of regression benchmarking, it is necessary to know the precision of such result values, so that a comparison of result values that differ by less than their respective precision is not interpreted as a performance change. An ideal result can be defined as a parameter of a random distribution that depends on the specific benchmark and the specific platform. This parameter is not known but can be estimated using experiments.

We will focus on estimating the mean value of the random distribution using an average of the measured durations. The precision of such an estimate can be determined using statistical methods. In practice, using a median in-

stead of the average can improve robustness in presence of outliers [1], but determining the precision of the estimate analytically is difficult in such a case. Robustness in presence of outliers is addressed in section 4.

Since we consider the result value of a benchmark to be the average of the measured durations, the result precision of the benchmark is the precision with which the result value estimates the mean value of the random distribution. We define the precision as a half-length of the 99% confidence interval for the mean value, therefore shorter interval means higher precision.

The exact formula that expresses the precision of a benchmark result depends on the choice of the statistical model that describes the benchmark. In [9], we have presented a simple additive model of initial conditions, which expects an additive impact of process initial conditions on operation durations. We introduce a more general model in section 3.1.

3.1 Benchmark Precision for Arbitrary Initial Conditions

We presume that the durations of operations measured by a benchmark in a run are random, independent and identically distributed, that the distributions from different runs can differ in parameters, and that the mean values of the distributions from different runs are identically distributed random variables. The result value is the average of the averages of the measured durations as an estimate of the mean value of the random mean values.

Specifically, for $j = 1..m$ as a benchmark run with $i = 1..n$ measurements,

- the durations of operations r_{ji} are observations of random variables R_{ji} identically distributed for $i = 1..n$, $E(R_{j1}|\mu_j) = \mu_j < \infty$, $var(R_{j1}|\sigma_j^2) = \sigma_j^2 < \infty$.
- μ_j are identically distributed random variables for each $j = 1..m$, $E(\mu_1) = \mu < \infty$, $var(\mu_1) = \rho^2 < \infty$.

The result value of a benchmark is

$$\overline{R}_{mn} = \frac{1}{mn}\sum_{j=1}^{m}\sum_{i=1}^{n} R_{ji}$$

as an estimate of the ideal result of a benchmark μ. From the rule of iterated expectations, it follows that μ is also the mean of R_{ji} if we do not know the specific value of μ_j:

$$E\left(R_{ji}\right) = E\left(E\left(R_{ji}|\mu_j\right)\right) = \mu.$$

We will show how to construct a confidence interval for μ. From the Central Limit Theorem (CLT), the distribution of $\overline{\mu}_m$ as an estimate of μ is asymptotically normal:

$$\frac{1}{m}\sum_{j=1}^{m}\mu_j = \overline{\mu}_m \sim N\left(\mu, \frac{\rho^2}{m}\right). \quad (1)$$

From CLT, the average of the averages M_j from run j,

$$M_j = \frac{1}{n}\sum_{i=1}^{n} R_{ji},$$

for the given fixed $\mu_j, \sigma_j, j = 1..m$ also has an asymptotically normal distribution

$$M_j|\mu_j, \sigma_j^2 \sim N\left(\mu_j, \frac{\sigma_j^2}{n}\right).$$

From the properties of the normal distribution:

$$\overline{M}_m|\overline{\mu}_m, \overline{\sigma_m^2} \sim N\left(\overline{\mu}_m, \frac{\overline{\sigma_m^2}}{mn}\right). \quad (2)$$

From (1) and (2), it can be shown that:

$$\overline{M}_m \sim N\left(\mu, \frac{\rho^2}{m} + \frac{\overline{\sigma_m^2}}{mn}\right). \quad (3)$$

The mean and variance of $\overline{M}_m$ in (3) can by verified by the rule of iterated expectations :

$$E[\overline{M}_m] = E\left[E\left[\overline{M}_m|\overline{\mu}_m, \overline{\sigma_m^2}\right]\right] = E\left[\overline{\mu}_m\right] = \mu$$

$$V\left[\overline{M}_m\right] = E\left[V\left[\overline{M}_m|\overline{\mu}_m, \overline{\sigma_m^2}\right]\right] + V\left[E\left[\overline{M}_m|\overline{\mu}_m, \overline{\sigma_m^2}\right]\right] =$$

$$= E\left[\frac{\overline{\sigma_m^2}}{mn}\right] + V\left[\overline{\mu}_m\right] = \frac{\overline{\sigma_m^2}}{mn} + \frac{\rho^2}{m}.$$

If we assume the variances σ_j^2 to be known or fixed and only the means μ_j to be random, it can be shown that the distribution of $\overline{M}_m$ is really normal. For details, see random effects model in one way classifications in [11]. The rationale behind the proof is that a convolution of Gaussians is known to be a Gaussian.

The confidence interval for the estimate of μ can now be constructed from (3). The result value of a benchmark is

$$\overline{M}_m = \overline{R}_{mn}$$

and the half-length of the $1 - \alpha$ confidence interval for the mean is

$$l = u_{1-\frac{\alpha}{2}} \cdot \sqrt{\frac{\rho^2}{m} + \frac{\overline{\sigma_m^2}}{mn}},$$

where u are quantiles of the standard normal distribution.

This result holds asymptotically for large n and large m. The unknown variance of the mean values μ_j can be approximated by the variance of the averages of durations from individual runs, which can be estimated using the S^2 estimate:

$$S_\rho^2 = \frac{1}{m-1}\sum_{j=1}^{m}\left[\left(\frac{1}{n}\sum_{i=1}^{n}R_{ji}\right) - \overline{R}_{mn}\right]^2.$$

The variance of the durations in a run σ_j^2 is still unknown. If the variance of the individual runs were constant, $\sigma_j^2 = \sigma^2$, we could estimate it by

$$S_\sigma^2 = \frac{1}{m(n-1)}\sum_{j=1}^{m}\sum_{i=1}^{n}\left(R_{ji} - \frac{1}{n}\sum_{i=1}^{n}R_{ji}\right)^2$$

to get the half-length of the $1-\alpha$ confidence interval for the mean:

$$l_\sigma = u_{1-\frac{\alpha}{2}} \cdot \sqrt{\frac{nS_\rho^2 + S_\sigma^2}{mn}}. \quad (4)$$

We can proceed using a similar approach when we know a maximum variance of the durations in a run σ_{max}^2 and estimate the upper bound of the length l, in other words a lower bound for the precision. We can also note that the formula for l does not rely on the individual values of variance, but only on the average variance $\overline{\sigma_m^2}$. We can therefore use the formulas for the length l_σ and the variance estimate S_σ^2 for large m, as implied by the Weak Law of Large Numbers (WLLN), except for the error of the estimate itself, which is not included in (4).

In benchmarking experiments, every benchmark process has to be warmed–up by several measurements of operation durations that are not included in the results, as they can be influenced by initialization noise. It is therefore most time-efficient to improve the result precision firstly by increasing the number of measurements in a run n and only secondly by increasing the number of runs m.

For $\rho^2 > 0$, the optimum number of measurements in a run n_{opt} can be derived from (4) and from the definition of the cost of the experiment $c = (w+n)m$, where w is the number of warm–up measurements:

$$n_{opt} = \sqrt{\frac{wS_\sigma^2}{S_\rho^2}}.$$

4 Handling Auto-Dependence and Outliers

An important assumption when determining the result of a benchmark and its precision as described in Section 3 is the independence and identical distribution of the durations of an operation execution in a single benchmark run. Our experience suggests that these assumptions do not generally hold in raw collected data.

The violation of the independence assumption is typically manifested by non-random patterns in the collected data. This was the case for some of the Mono benchmarks, where the violation of independence was probably caused by the just-in-time compiler or the garbage collector. As for the identical distributions, this assumption is typically violated by outlying measurements, caused by relatively infrequent distortions which influence the duration of the measured operation.

We therefore preprocess the collected data before applying the methods from Section 3.

4.1 Quantifying Auto-Dependence

The plot in Figure 1 shows that in each run of the FFT benchmark, we can observe several values that are typical for the run and around which we can find, with certain variance, all the measured values. These typical values differ between benchmark runs and the variance is greater than the variance of the values in a single run, which results in the horizontal stripes or clusters that can be seen in the plot.

The clusters visible in Figure 1 appear to have the same, or at least very similar, variance. This effect can be more accurately quantified with the help of the impact factor of the initial conditions described in Section 2. The approach is similar to determining the extent to which the influence of the initial conditions fits the additive model in [9]. The measured data are passed to a clustering additive filter, which first splits the measured values into clusters using the Mclust algorithm [7, 6]. Then, for each cluster, the average of durations from the cluster is subtracted from each duration in the cluster. This applies the additive filter to the individual clusters. After applying the filter, the impact factor of the initial conditions is computed for the resulting data.

In case of the FFT benchmark, the impact factors for different versions of Mono after applying the clustering additive filter are close to 1, suggesting that the impact of the initial conditions is described well by the model in 3.1 or the additive model described in [9] when applied to individual clusters. The situation can be illustrated by comparing the plot in Figure 3 with the plot in Figure 2.

The violation of the assumption of sample independence in case of the FFT benchmark is clear from the following experiment. First, we number the clusters and transform the original data into a sequence of cluster indices by mapping all values from the same cluster to the respective cluster index. We can then observe that the interleaving of the cluster indices in the resulting sequence is very systematic. This effect is also clearly visible in a lag plot of the measured data,

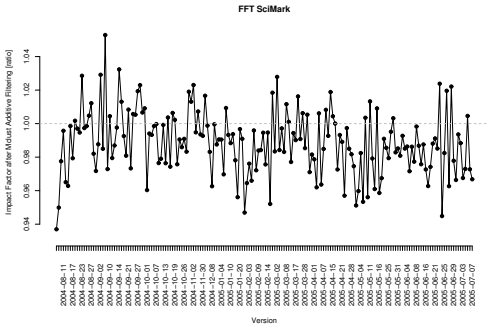


Figure 3. Impact factors of initial conditions after clustered additive filtering in FFT Scimark.

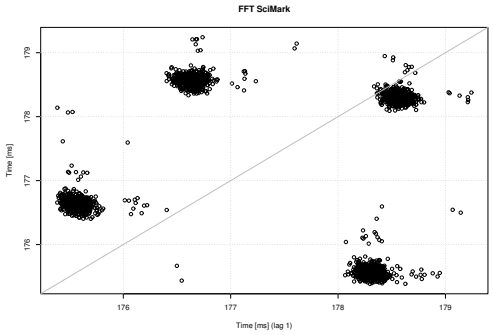


Figure 4. Lag-plot of FFT computation times in a single benchmark run.

which is commonly used for inspecting auto-dependence in time series.

Figure 4 shows a lag plot of the data from a single run of the FFT Scimark benchmark. For comparison, Figure 5 shows a lag plot of the same but randomly reordered data, which represents the scenario where the individual measurements are independent. The plot in Figure 4 shows that observing a value from a particular cluster restricts the possible value of the next observation to a specific cluster.

4.2 Data Preprocessing

The presence of outliers in the data is a typical issue associated with measurements of real systems, and therefore applies to benchmarking computer systems as well [2, 1].

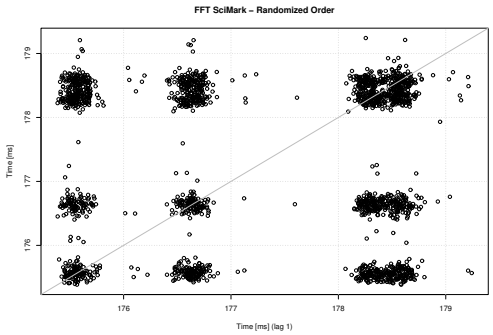


Figure 5. Lag-plot of randomly reordered FFT computation times in a single benchmark run.

The nature of the outliers is such that under certain circumstances, the duration of an operation can be as much as several orders of magnitude longer than in most other cases. Although the occurrence of the outliers is rare, it has a significant impact on the results. Since we do not have a plausible model for the collected data of a generic benchmark on a generic system, we use a simple simulation technique to preprocess the data, allowing us to neglect the impact of the outliers on the results. This technique also tackles the auto-dependence described earlier.

The algorithm for obtaining the result and precision of a benchmark, including the data preprocessing, follows. Symbols correspond to those used in Section 3.1):

- execute m benchmark runs, collecting $w+k$ measured durations r_{ji}, $j = 1..m$, $i = 1..w+k$ of the same operation each run,
- for each benchmark run j, repeatedly (e.g. 100 times) generate a random sub-selection of size n using sampling with replacement, where $n < k$ (e.g. $n = 0.75 \cdot k$) from measurements $r_{j,(w+1)}..r_{j,(w+k)}$, and calculate the median M_j of averages of all sub-selections,
- for each benchmark run j, generate another set of random sub-selections using the method from the previous step and calculate the median $S^2_{\sigma_j}$ of sample variances of all sub-selections,
- the result of the benchmark is $\overline{M}_m = \frac{1}{m}\sum_{j=1}^{m} M_j$,
- the precision of the benchmark result as the half-length of the $1-\alpha$ confidence interval for the mean is

$$l_\sigma = u_{1-\frac{\alpha}{2}} \cdot \sqrt{\frac{nS^2_\rho + S^2_\sigma}{mn}},$$

where

$$S_\sigma^2 = \frac{1}{m}\sum_{j=1}^{m} S_{\sigma_j}^2,$$

$$S_\rho^2 = \frac{1}{m-1}\sum_{j=1}^{m}\left(M_j - \overline{M}_m\right)^2.$$

With the knowledge of the benchmark result and its precision in the form of confidence interval half-length, we can automatically detect statistically significant changes in performance, which is explained in Section 5.

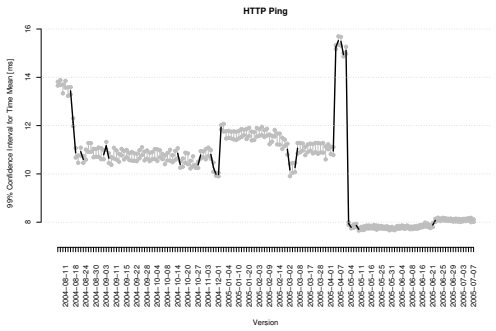
5 Automated Detection of Changes

Regression benchmarking requires automated detection of changes in performance between different versions of the software under development. Performance is assessed using benchmarks that determine the average duration of the measured operation as well as the confidence interval for the mean as a measure of precision.

A performance change is reported whenever the confidence intervals for the mean operation durations of two consecutive versions of the tested software do not overlap. To assess the magnitude of a performance change, we use a ratio of the distance between the centers of the confidence intervals for the older and the newer version to the center of the confidence interval for the older version. The center of the confidence interval is the average of averages calculated as a result of the benchmark. This quantification has only an informative character though, as it does not take into account the lengths of the confidence intervals, i.e. the precision of the benchmark results.

The plot in Figure 6 shows significant changes in performance for different versions of Mono as measured by the HTTP Ping benchmark. The horizontal axis shows an index of the tested Mono version, the vertical axis is the response time. The confidence intervals for the mean as described in Section 3.1 are marked by gray lines, the performance changes are marked by bold black lines. The table below the plot summarizes and quantifies the detected changes. Each row of the table contains the dates of the older and newer versions between which the change was detected and the size of the change as percentage of the older version. Changes quantified as positive in the table are therefore regressions.

For practical employment of regression benchmarking in software development, it is important to be able to locate modifications in sources that are suspect causes of the detected performance changes. This issue is addressed in [10].



Newer Version	Older Version	Change
2004-08-17	2004-08-13	-9.67%
2004-08-18	2004-08-17	-10.44%
2004-08-23	2004-08-20	-2.95%
2004-09-03	2004-09-02	3.4%
2004-09-07	2004-09-03	-4.57%
2004-10-15	2004-10-14	-4.29%
2004-10-29	2004-10-27	3.96%
2004-11-05	2004-11-04	-4.9%
2004-11-30	2004-11-05	-2.87%
2004-12-20	2004-12-01	19.64%
2005-03-02	2005-02-28	-7.81%
2005-03-07	2005-03-04	7.77%
2005-04-05	2005-04-04	39.29%
2005-04-06	2005-04-05	1.77%
2005-04-08	2005-04-07	-3.58%

Figure 6. Confidence intervals for mean response time in HTTP Ping with detected significant changes.

6 Mono Regression Benchmarking Project

The Mono Regression Benchmarking Project applies the methods described in the paper at detecting performance regressions in daily development snapshots of Mono, an open-source implementation of the .Net platform. The project serves as a testbed for development and validation of methods for benchmarking and analysis of data for the purpose of regression benchmarking.

The project currently includes five benchmarks - the Fast Fourier Transform (FFT) benchmark, the HTTP Ping and TCP Ping benchmarks which test remove method invocation, Scimark [14, 13] which tests floating point computation and Rijndael which tests a single encryption algorithm. A more detailed description of the benchmarks can be found in [10].

The benchmarking environment is fully automated and the results are continuously updated on the web of the project [4]. The presented graphs are similar to the graph on Figure 6 and other graphs presented in this paper.

7 Conclusion

Regression benchmarking, as a part of regression testing, is a promising approach that allows the developers to monitor the performance of software during development. Regression benchmarking comprises regular execution of many benchmarks. For practical use, the detection of performance changes must be automated, which in turn requires the knowledge of the precision of the benchmark results.

We bring attention to a frequently overlooked dependency of benchmark results on random initial conditions, present methods for quantifying their influence on various benchmarks and characterize their influence on benchmark results.

For determining the precision of benchmark results, we present methods that take into account the random initial conditions and auto-dependence in the data from a single benchmark run, which is also an often-overlooked dependency. The presented methods allow determining the optimal number of benchmark runs and the number of measurements that should be collected in each run in order to maximize the precision of a benchmark result in given time.

Most of the proposed methods and approaches have been implemented in a simple and fully automated regression benchmarking system that monitors performance and detects performance changes in daily development snapshots of the Mono project. Future development will focus on integrating the method for determining the optimal number of benchmark runs and the number of measurements in a run with the benchmarking system.

A challenge for future work comprises automated, or at least partially automated, correlation of source code modifications with the detected performance changes.

Acknowledgment. The authors would like to express their thanks to Jaromir Antoch, Alena Koubkova and Tomas Ostatnicky for their help with mathematical statistics. This work was partially supported by the Grant Agency of the Czech Republic projects 201/03/0911 and 201/05/H014.

References

[1] A. Buble, L. Bulej, and P. Tuma. Corba benchmarking: A course with hidden obstacles. In *IPDPS*, page 279. IEEE Computer Society, 2003.

[2] L. Bulej, T. Kalibera, and P. Tuma. Regression benchmarking with simple middleware benchmarks. In H. Hassanein, R. L. Olivier, G. G. Richard, and L. L. Wilson, editors, *International Workshop on Middleware Performance, IPCCC 2004*, pages 771–776, 2004.

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

[5] ECMA. *ECMA-335: Common Language Infrastructure (CLI)*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, Dec. 2002.

[6] C. Fraley and A. E. Raftery. Mclust: Software for model-based clustering, density estimation and discriminant analysis. Technical Report 415, Department of Statisticis, University of Washington, WA, USA, Oct 2002.

[7] C. Fraley and A. E. Raftery. Model-based clustering, discriminant analysis, and density estimation. *Journal of the American Statistical Association*, 97:611–631, 2002.

[8] R. E. Jeffries, A. Anderson, and C. Hendrickson. *Extreme Programming Installed*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[9] T. Kalibera, L. Bulej, and P. Tuma. Benchmark precision and random initial state. In *accepted for 2005 International Symposium on Performance Evaluation of Computer and Telecommunications Systems (SPECTS 2005)*, July 2005.

[10] T. Kalibera, L. Bulej, and P. Tuma. Quality assurance in performance: Evaluating mono benchmark results. In *accepted for Second International Workshop on Software Quality (SOQUA 2005)*, Sept. 2005.

[11] C. E. McCulloch and S. R. Searle. *Generalized, Linear and Mixed Models*. Wiley–Interscience, New York, NY, USA, 2001.

[12] Novell, Inc. The Mono Project. `http://www.mono-project.com`, 2005.

[13] R. Pozo and B. Miller. Scimark 2.0 benchmark. `http://math.nist.gov/scimark2/`, 2005.

[14] C. Re and W. Vogels. Scimark – c#. `http://rotor.cs.cornell.edu/SciMark/`, 2004.

Chapter 7

Precise Regression Benchmarking with Random Effects: Improving Mono Benchmark Results

Tomáš Kalibera,
Petr Tůma

Contributed paper at **3rd European Performance Engineering Workshop (EPEW 2006)** [9], acceptance rate 40%.

In *Formal Methods and Stochastic Models for Performance Evaluation*,
published by Springer-Verlag,
LNCS 4054,
pages 63–77,
ISSN 0302-9743,
June 2006.

The original version is available electronically from the publisher's site at http://dx.doi.org/10.1007/11777830_5.

Precise Regression Benchmarking with Random Effects: Improving Mono Benchmark Results

Tomas Kalibera and Petr Tuma

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
{kalibera,tuma}@nenya.ms.mff.cuni.cz

Abstract. Benchmarking as a method of assessing software performance is known to suffer from random fluctuations that distort the observed performance. In this paper, we focus on the fluctuations caused by compilation. We show that the design of a benchmarking experiment must reflect the existence of the fluctuations if the performance observed during the experiment is to be representative of reality.
We present a new statistical model of a benchmark experiment that reflects the presence of the fluctuations in compilation, execution and measurement. The model describes the observed performance and makes it possible to calculate the optimum dimensions of the experiment that yield the best precision within a given amount of time.
Using a variety of benchmarks, we evaluate the model within the context of regression benchmarking. We show that the model significantly decreases the number of erroneously detected performance changes in regression benchmarking.

Key words: performance evaluation, benchmark precision, random effects, regression benchmarking.

1 Introduction

Software performance engineering is generally understood as a systematic process of planning and evaluating software performance [1]. One of the principal approaches to evaluating performance is benchmarking, where the system under test executes a model task, called benchmark, and the observed performance is used for the evaluation. An important feature of benchmarking is that a choice of a realistic benchmark and a realistic configuration of the benchmarking experiment makes the observed performance representative of the performance of a real system. This makes benchmarking an indispensable complement of other approaches to evaluating performance based on modeling and simulation.

Both the performance of a benchmarking experiment and the performance of a real system are subject to random fluctuations. Well known causes of these fluctuations include for example the asynchronous device interrupts, whose often unpredictable occurrence can add the device interrupt service time to the observed

performance. To keep the observed performance representative, benchmarking experiments typically measure the benchmark multiple times. Averaging over the multiple measurements is then used to filter out the random fluctuations. In [2], however, we show that this practice suffers from a lack of understanding of the causes of random fluctuations. Consequently, even after averaging, the performance of a benchmarking experiment is not necessarily representative of the performance of a real system.

In order to correctly understand the causes of random fluctuations in observed performance, a benchmarking experiment must be viewed as a sequence of steps. This sequence begins with the compilation of the benchmark and proceeds through booting of the system under test to the execution of the process implementing the benchmark and the measurement of the benchmark itself as the final steps. Importantly, each of the steps has the potential to influence the observed performance, and each of the steps can be subject to nondeterminism that makes the influence assume the form of random fluctuations. In [2], we illustrate this influence by showing how the choice of physical memory pages used to store the benchmark impacts the observed performance. This choice cannot be practically influenced and as such is one of the sources of nondeterminism in the execution of a benchmark.

The common practice of averaging can still be made to cover all the causes of random fluctuations. To achieve this, all the steps of the benchmarking experiment would have to be done once for each measurement, rather than just once for all the measurements. Unfortunately, some of the steps of the benchmarking experiment can take a long time and repeating them enough times to obtain enough measurements for a representative average would take a prohibitively long time. To avoid this problem, we propose a novel statistical model that reflects the understanding of the benchmarking experiment as a sequence of steps that can be repeated starting with any step of the experiment and finishing with the measurement step (e.g. compiling multiple times, executing each compiled binary multiple times, collecting multiple measurements for each execution).

The model makes it possible to derive the asymptotic distribution of the average of the observed performance, and use this distribution to create the asymptotic confidence interval for the mean observable performance, as well as determine the optimal ratio of the repetitions of the individual benchmark experiment steps. The model can describe benchmark experiments where at most three of the steps influence the observed performance, and is an extension of the model from [3] that could describe benchmark experiments where at most two of the steps influenced the observed performance.

As a proof of concept, we apply the statistical model in regression benchmarking. Regression benchmarking [4] is a new methodology for automated tracking of performance during software development. In our evaluation, we apply the methodology on omniORB [5] and Mono [6] as large open source projects with frequent changes. The omniORB platform is an open source implementation of the CORBA standard, consisting of an IDL compiler, an object request broker and object services, totaling almost 200k lines of code. The Mono platform is an

open source implementation of the Common Language Infrastructure [7], also known as Microsoft .NET, consisting of a C# compiler, a virtual machine and application libraries, totaling almost 3M lines of code. Our evaluation relies on the Mono Regression Benchmarking Project [8], which tracks performance of daily Mono versions on several different benchmarks since August 2004, with the results continuously available on the web [8].

In the proof of concept, we focus on the nondeterminism in the compilation step of a benchmark experiment, thus complementing [3], where only the nondeterminism in the execution and measurement steps of a benchmark experiment is tackled. The quantification of the benefits is based on the percentage of "false alarms" in the form of spurious reports of performance changes by the regression benchmarking methodology, which can be reduced from as high as 50% when using the model from [3] to as low as 4% when using the proposed model.

The paper follows by analysis and quantification of the random effects of compilation in Section 2. A new statistical model that describes benchmarking experiments with random effects of compilation is described in Section 3. The model is evaluated in the context of the regression benchmarking methodology in Section 4. The paper is concluded in Section 5.

2 Problem of Random Effects of Compilation

The compilation of benchmarks for complex software is necessarily a complex task in itself. Using the example of the omniORB platform, compiling a typical benchmark includes compiling the core libraries, compiling and linking the IDL compiler, using this IDL compiler to generate stubs and skeletons, compiling the benchmark itself and linking the benchmark with the core libraries. Similarly, using the example of the Mono platform, compiling a typical benchmark includes compiling and linking the virtual machine, compiling the C# compiler using another bootstrap compiler and using this compiler to compile the core libraries and the benchmark itself. It is important to note that the process of compilation is not always entirely reproducible.

In [2], we have identified one particular source of nondeterminism in compilation of C++ code by the GNU C++ compiler [9]. The compiler generates random names for symbols defined in anonymous namespaces. As a consequence, the linker places these symbols in different locations within the binary for each compilation. During execution, a difference in the location of the symbols is reflected as a difference in the number of cache misses. This source of nondeterminism can influence the compilation of the omniORB platform, other sources of nondeterminism exist that can influence the compilation of the Mono platform.

It should be emphasized that various sources of nondeterminism exist in various processes of compilation [10]. These are frequently associated with the internal workings of a particular compiler on a particular platform. An exhaustive search for all sources of nondeterminism in compilation with the goal of eliminating them from benchmarking experiments is therefore not a feasible approach. To characterize how much the random effects of compilation impact the

observed performance in a way that is independent of the particular sources of nondeterminism in compilation, we have introduced a metric called "impact factor of random effects of compilation" [2]. The metric is defined as a ratio of the standard deviation of the mean response times from different binaries to the standard deviation of the mean response times from the same binary. An impact factor of 1 indicates no impact of random effects on the response time, values larger than 1 indicate an impact of the random effects. The value of the impact factor is estimated by simulation (bootstrap). More details can be found in [2].

In Figure 1, we show the impact factors for selected benchmarks that cover a range of software applications. The Ping and Marshal benchmarks are omniORB benchmarks that assess remote method invocation, the other benchmarks are Mono benchmarks that assess remote method invocation, numerical computation and cryptography, see Appendix C and [8]. The figure also lists the variation of the results attributed to the random effects in compilation, related to the mean. Figure 1 shows that random effects of compilation influence results of almost all of the selected benchmarks. For these benchmarks, ignoring these effects can therefore mean that the performance of a benchmarking experiment will not be representative of the performance of a real system. The practical impact of relying on such benchmarking experiments depends on the particular use of the experiment. An evaluation in the context of regression benchmarking follows in Section 4.

Benchmark	Impact Factor	Relative (%) Variation
FFT	1.18	4.1
FFT (NA)	1.08	3.35
FFT (NA,OPT)	1.08	3.42
FFT (OPT)	1.13	4.41
HTTP	1.03	0.19
HTTP (OPT)	1.03	0.23

Benchmark	Impact Factor	Relative (%) Variation
Rijndael	1.01	0.38
Rijndael (OPT)	1.	0.38
TCP	1.05	0.56
TCP (OPT)	1.04	0.56
Marshal	1.05	2.
Ping	1.12	0.81

Fig. 1. Impact factor of random effects in compilation and relative variation caused by these effects for selected benchmarks.

3 Benchmarking with Random Effects of Compilation

As suggested in Section 1, a simplistic solution to the problem of random effects of compilation is to repeat all the steps of the benchmarking experiment that preceed the measurement once for each measurement rather than just once for all the measurements, and to estimate the response time of the benchmark from the individual response times collected one in each measurement. Formally, the mean response time can be estimated by average and the precision of the estimate by an asymptotic confidence interval. Increasing the number of repetitions improves

the precision, with an obvious drawback – the repetition of the compilation step takes too long.

In this section, we provide a statistical model of a benchmark experiment, that covers random effects at all three levels – compilation, execution and measurement. The model allows both to estimate the result precision and to choose the optimal number of measurements per execution and the optimal number of executions per binary. These numbers are optimal in respect that they minimize the time needed for the benchmarking experiment. The model is designed to be as generic as possible, so that it covers the widest possible range of benchmarks. In particular, the model works both for benchmarks where repeating measurements or executions helps as well as for benchmarks where it does not help. As a consequence, the model requires to always repeat the executions and measurements several times to adapt to a particular benchmark. This is not a problem, since compilation of large projects takes several orders of magnitude longer than execution or measurement.

3.1 Statistical Model of Benchmark with Random Effects

The intuitive idea behind the model is that the mean of measured response times in each execution is in fact a realization of a random variable, which is characteristic for the respective binary (the response times in an execution are prone to random effects). Similarly, the mean of this random variable is also in fact a realization of another random variable, which is characteristic for the respective software version (the execution means are prone to random effects).

We will now formalize the intuitive idea. Let $Y \sim F_Y\left(\mu_Y, \sigma_Y^2\right)$ denote a random operation response time in a given software version. The distribution F_Y of Y is unknown; we assume that it has finite mean μ_Y and finite variance σ_Y^2. The parameter of interest is the mean response time μ_Y.

We assume that response times in each benchmark execution are independent identically distributed (i.i.d.), with a finite variance σ_E^2 that is fixed for all executions in a given software version, and with a finite mean μ_E that differs for each execution. The parameter μ_E is in fact a sample from a random variable M_E. For better readability, we will write "μ_E" and "$Y|\mu_E$" instead of "M_E" and "$Y|\left[M_E = \mu_E\right]$":

$$E\left(Y|\mu_E\right) = \mu_E, \quad var\left(Y|\mu_E\right) = \sigma_E^2. \tag{1}$$

We assume that the execution mean times μ_E for each binary are random i.i.d., with a finite variance σ_B^2 that is fixed for all binaries in a given software version, and with a finite mean μ_B that differs for each binary:

$$E\left(\mu_E|\mu_B\right) = \mu_B, \quad var\left(\mu_E|\mu_B\right) = \sigma_B^2. \tag{2}$$

We assume that binary mean times μ_B for each software version are random i.i.d., with a finite mean μ_V and a finite variance σ_V^2, which are fixed for a given software version:

$$E\left(\mu_B\right) = \mu_V, \quad var\left(\mu_B\right) = \sigma_V^2. \tag{3}$$

In this model, $\mu_Y = \mu_V$. This can be easily shown using The Rule Of Iterated Expectations [11], which says that for random variables X and Y, assuming the expectations exist,

$$E\left[E(Y|X)\right] = E(Y) : \tag{4}$$

$$\begin{aligned}\mu_Y = E(Y) =^{(4)} E\left[E(Y|\mu_E)\right] =^{(1)} E(\mu_E) =^{(4)} \\ = E\left[E(\mu_E|\mu_B)\right] =^{(2)} E(\mu_B) =^{(3)} \mu_V.\end{aligned}$$

It can also be shown, that $\sigma_Y^2 = \sigma_E^2 + \sigma_B^2 + \sigma_V^2$, using The Rule Of Iterated Expectations and a known property of conditional variance [11], which says that for random variables X and Y,

$$var(Y) = E\left[var(Y|X)\right] + var\left[E(Y|X)\right] : \tag{5}$$

$$\begin{aligned}\sigma_Y^2 = var(Y) =^{(5)} E\left[var(Y|\mu_E)\right] + var\left[E(Y|\mu_E)\right] =^{(4),(1)} \\ = \sigma_E^2 + var(\mu_E) =^{(5)} \sigma_E^2 + E\left[var(\mu_E|\mu_B)\right] + var\left[E(\mu_E|\mu_B)\right] =^{(4),(2)} \\ = \sigma_E^2 + \sigma_B^2 + var(\mu_B) =^{(3)} \sigma_E^2 + \sigma_B^2 + \sigma_V^2.\end{aligned}$$

The parameter of interest μ_Y is unknown, we will estimate it from the data: let us assume that we have compiled a given software version l times creating l binaries, and that we have executed each benchmark binary m times, getting n post–warmup measurements in each execution. In the rest of this section, we will show that μ_Y can be estimated by average of all the measurements

$$\overline{Y}_{\bullet\bullet\bullet} \stackrel{def}{=} \frac{1}{lmn}\sum_{k=1}^{l}\sum_{j=1}^{m}\sum_{i=1}^{n} Y_{ijk},$$

and that this estimate is asymptotically normal:

$$\overline{Y}_{\bullet\bullet\bullet} \approx N\left(\mu_Y, \frac{\sigma_E^2}{lmn} + \frac{\sigma_B^2}{lm} + \frac{\sigma_V^2}{l}\right). \tag{6}$$

Lemma 31 *Let $X_1, ..., X_n$ be i.i.d. with mean μ and finite positive variance σ^2. Then, $\overline{X}_\bullet$ has asymptotically normal distribution: $\overline{X}_\bullet \approx N\left(\mu, \frac{\sigma^2}{n}\right)$. Lindeberg–Levy Central Limit Theorem.*

Lemma 32 *Let $X_1, ..., X_n$ be independent, $X_i \sim N\left(\mu_i, \sigma_i^2\right)$. From the properties of normal distribution [11], it follows that: $\overline{X}_\bullet \sim N\left(\overline{\mu}_\bullet, \overline{\sigma^2}_\bullet\right)$.*

Lemma 33 *Let $X \sim N\left(\mu_X, \sigma_X^2\right)$ and $Y|\left[X = x\right] \sim N\left(x, \sigma^2\right)$. Then, $Y \sim N\left(\mu_X, \sigma_X^2 + \sigma^2\right)$. The proof is outlined in Appendix A.*

By Lemma 31 we have, from (1),(2),(3):

$$\overline{Y}_{kj\bullet}|\mu_{E_{kj}} \approx N\left(\mu_{E_{kj}}, \frac{\sigma_E^2}{n}\right), \tag{7}$$

$$\overline{\mu_E}_{k\bullet}|\mu_{B_k} \approx N\left(\mu_{B_k}, \frac{\sigma_B^2}{m}\right), \tag{8}$$

$$\overline{\mu_B}_{\bullet} \approx N\left(\mu_V, \frac{\sigma_V^2}{l}\right). \tag{9}$$

By applying Lemma 32 on (7),(8), we get by turns (10),(11). Then, by applying the same lemma again on (10), we get (12):

$$\overline{Y}_{k\bullet\bullet}|\overline{\mu_E}_{k\bullet} \approx N\left(\overline{\mu_E}_{k\bullet}, \frac{\sigma_E^2}{mn}\right) \tag{10}$$

$$\overline{\mu_E}_{\bullet\bullet}|\overline{\mu_B}_{\bullet} \approx N\left(\overline{\mu_B}_{\bullet}, \frac{\sigma_B^2}{lm}\right) \tag{11}$$

$$\overline{Y}_{\bullet\bullet\bullet}|\overline{\mu_E}_{\bullet\bullet} \approx N\left(\overline{\mu_E}_{\bullet\bullet}, \frac{\sigma_E^2}{lmn}\right) \tag{12}$$

By applying Lemma 33 on (9) and (11), we get

$$\overline{\mu_E}_{\bullet\bullet} \approx N\left(\mu_V, \frac{\sigma_B^2}{lm} + \frac{\sigma_V^2}{l}\right). \tag{13}$$

Finally, by applying Lemma 33 on (13) and (12), we get (6). □

3.2 Change Detection

In regression benchmarking, we need to detect a performance change between two consecutive versions of selected software. Currently, we focus only on mean response time. In terms of the model described above, we want to detect a change, whenever μ_Y changes between two consecutive versions. Because we cannot assume to have a long period of versions without a change, we cannot directly use methods of change–point detection or quality control. The option of modifying some of these methods for regression benchmarking is left for future work.

Currently, we use a simple comparison method based on confidence intervals: we detect a change whenever confidence intervals for the mean from two consecutive versions do not overlap. The method is similar to the Approximate Visual Test described by Jain [12], where t–test is used to detect changes in case the center of one confidence interval falls into the other confidence interval.

The asymptotic confidence interval for μ_Y can be constructed using (6). We can estimate the unknown variances σ_E^2, σ_B^2 and σ_E^2 by S_E^2, S_B^2 and S_V^2 as follows:

$$S_E^2 = \frac{1}{lm(n-1)} \sum_{k=1}^{l} \sum_{j=1}^{m} \sum_{i=1}^{n} \left(Y_{kji} - \overline{Y}_{kj\bullet}\right)^2 \tag{14}$$

$$S_B^2 = \frac{1}{l(m-1)} \sum_{k=1}^{l} \sum_{j=1}^{m} \left(\overline{Y}_{kj\bullet} - \overline{Y}_{k\bullet\bullet} \right)^2 \tag{15}$$

$$S_V^2 = \frac{1}{l-1} \sum_{k=1}^{l} \left(\overline{Y}_{k\bullet\bullet} - \overline{Y}_{\bullet\bullet\bullet} \right)^2 \tag{16}$$

Since we do not assume normal distributions of μ_B, $\mu_E|\mu_B$ and $Y|\mu_E$, we cannot assume $\overline{Y}_{\bullet\bullet\bullet}$ to follow the t–distribution. We therefore have to rely on the asymptotic normality of $\overline{Y}_{\bullet\bullet\bullet}$, even after the estimates of the variances are used instead of the unknown variances. The asymptotic $(1-\alpha)$ confidence interval for μ_Y used for change detection therefore is

$$\overline{Y}_{\bullet\bullet\bullet} \pm u_{1-\frac{\alpha}{2}} \sqrt{\frac{S_E^2}{lmn} + \frac{S_B^2}{lm} + \frac{S_V^2}{l}}, \tag{17}$$

where $u_\bullet$ is the quantile function of the standard normal distribution. Thus, the probability that μ_Y lies within this interval is asymptotically $(1-\alpha)$.

3.3 Determining Optimum Number of Executions and Measurements

When detecting changes using confidence intervals as described above, the shorter the interval is, the higher is the chance of discovering a performance change. The width of the confidence interval (17) can be reduced only by proper selection of the numbers of measurements, executions and binaries – n, m, l, because the confidence level $(1-\alpha)$ is fixed and the variance estimates S_E^2, S_B^2 and S_V^2 are properties of the given software version.

From (17), it is clear that increasing the number of binaries l always reduces the interval width. Increasing the number of executions m reduces the width only partially, because it does not reduce the impact of S_V^2 (random effects in compilation). Similarly, increasing the number of measurements n does not reduce the impact of S_B^2 (random effects in execution) and S_V^2 . On the other hand, increasing the number of measurements n is usually less expensive than increasing the number of executions m, which is in turn less expensive than increasing the number of compilations l. Therefore, optimum values of n and m should exist, that guarantee the shortest confidence interval given a fixed time for the benchmarking experiment. The optimum values would depend on S_E^2, S_B^2 and S_V^2. This intuitive idea will be formalized further in this section.

We define the cost c of a benchmarking experiment:

$$c = (b + (w+n) \cdot m) \cdot l, \tag{18}$$

where w is the number of measurements in the warm–up stage of each benchmark execution (price for a new execution) and b is the number of measurements that could be taken in the time needed for compilation (price for a new binary). The values of w and b have to be estimated or determined by experience, as discussed

below. Our objective is to find m,n such that for the fixed cost c, $f(m,n,l)$ is minimal:

$$f(m,n,l) = \frac{S_E^2}{lmn} + \frac{S_B^2}{lm} + \frac{S_V^2}{l}. \tag{19}$$

After eliminating l using (18), $f(m,n)$ is

$$f(m,n) = \frac{mw + mn + b}{c} \cdot \left(\frac{S_E^2}{mn} + \frac{S_B^2}{m} + S_V^2 \right). \tag{20}$$

It is shown in Appendix B that the minimum is reached in

$$m_0 = \sqrt{\frac{b}{w} \cdot \frac{S_B^2}{S_V^2}}, \quad n_0 = \sqrt{w \cdot \frac{S_E^2}{S_B^2}}. \tag{21}$$

In practice, the length of the warm–up stage w depends on the benchmark platform and benchmark application and can be set by experience. It is important not to understate w in order to get relevant results [13]. The value of b can be estimated by experiments, it depends on the used compiler, the build scripts and the code size. From our experience, neither b nor w vary significantly between software versions. Still, the variances σ_E^2, σ_B^2 and σ_V^2 do vary between versions, and we have to collect enough measurements in enough executions for enough binaries to get variance estimates S_E^2, S_B^2, S_V^2. How much is enough depends on each benchmark and platform. With these estimates, we can calculate the confidence interval width (17), and if the width is too large, we can run an additional experiment with the optimum values of m and n using (21).

Some benchmarks measure only the response time of a part of a larger operation, where the whole operation is repeatedly invoked. An example of such a benchmark is the Marshal benchmark, which in fact repeatedly runs a remote procedure call, but measures only the marshaling part of the call. Let us assume that the measured operation takes q times less time than the repeated operation. The cost of the experiment is then still expressed in the number of measurements of the measured operation:

$$c = (b + (w + n) \cdot m \cdot q) \cdot l. \tag{22}$$

It is shown in Appendix B that the optimum numbers of measurements and executions are:

$$m_0 = \frac{1}{\sqrt{q}} \cdot \sqrt{\frac{b}{w} \cdot \frac{S_B^2}{S_V^2}}, \quad n_0 = \sqrt{w \cdot \frac{S_E^2}{S_B^2}}. \tag{23}$$

The optimum number of executions m_0 is smaller than in (21), because the cost of the execution has been understated compared to the cost of the compilation. The optimum number of measurements n is the same, because the cost of the measurement compared to the cost of the execution did not change: both in warm–up phase and non warm–up phase, the whole operation is repeated. The value of q can be estimated by experiments. By our experience, it does not vary significantly between software versions.

4 Evaluation

The evaluation of the proposed statistical model is done in the context of regression benchmarking [4]. The essential part of regression benchmarking is an automated comparison of observed performance between different software versions, with the goal of identifying instances of performance changes from version to version. Regression benchmarking is therefore sensitive to random fluctuations in the observed performance, which exhibit themselves as "false alarms" – spurious reports of performance changes that are caused by the random fluctuations rather than differences between software versions.

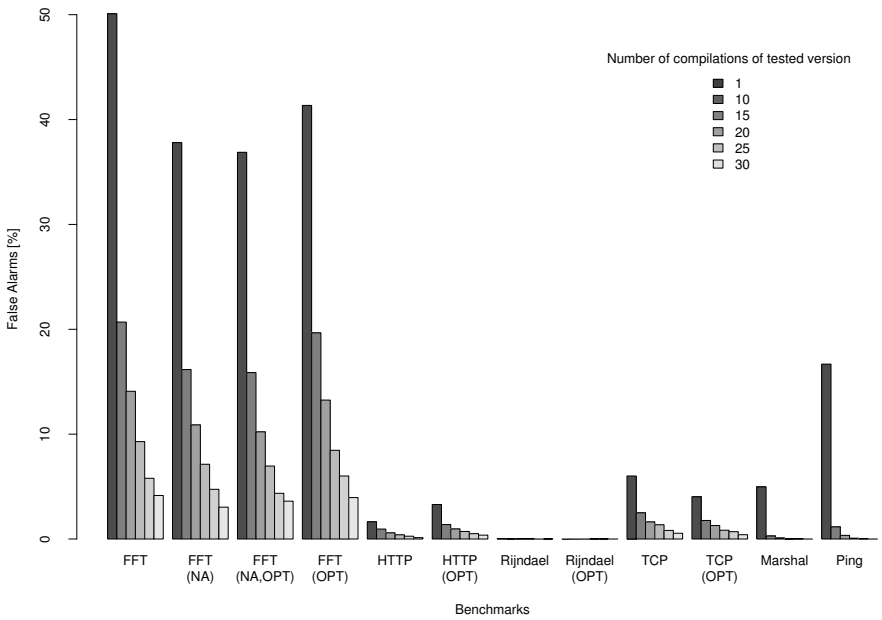
The evaluation is made difficult by the fact that deciding whether a change in observed performance corresponds to a change between software versions requires manual analysis of the software versions in question. Such an analysis becomes prohibitively expensive when enough data for a statistically significant evaluation needs to be collected. We overcome this obstacle by comparing multiple benchmarking experiments on the same software version in place of multiple benchmarking experiments on multiple software versions. Then, all the detected changes are necessarily false alarms.

In more detail, the evaluation begins with compiling the same software version many times into a number of binaries, executing each binary a number of times and collecting a number of measurements from each execution. The exact numbers of compilations, executions and measurements are chosen to maximize the reliability of the evaluation. The evaluation proceeds with simulation (bootstrap). For each benchmark, the simulation is repeated a number of times, each time two groups of binaries are chosen by random and compared using the proposed statistical model. The results are shown in Figure 2, contrasted against the results obtained using the model from [3] with only a single binary per group.

The evaluation suggests that different benchmarks suffer from false alarms to different degrees. The FFT benchmarks suffer most – this can be explained by the fact that they use a lot of memory and are therefore sensitive to the performance of the memory cache. On the other hand, the Rijndael benchmark does not suffer from false alarms at all – the encryption and decryption is computationally intensive, but does not need much memory. It is also interesting that in omniORB benchmarks, the decrease in the number of false alarms with the growing number of binaries is much faster than in Mono benchmarks. We attribute this to the fact that the random effects of compilation in Mono benchmarks are more complex than in omniORB benchmarks.

5 Conclusion

The compilation of large applications is often a non–repeatable process. Compiling the same sources with the same compiler under the same settings can and often does result in different binaries that deliver different performance. As a result and contrary to the common practice, multiple binaries should be used for benchmarking. We show on a diverse set of benchmarks how using only a



Benchmark	False Alarms (%) for Different Numbers of Compilations					
	1	10	15	20	25	30
FFT	50.09	20.69	14.09	9.29	5.79	4.15
FFT (NA)	37.80	16.16	10.88	7.13	4.74	3.04
FFT (NA,OPT)	36.88	15.87	10.22	6.96	4.36	3.61
FFT (OPT)	41.35	19.66	13.25	8.46	6.01	3.95
HTTP	1.64	0.95	0.59	0.40	0.27	0.13
HTTP (OPT)	3.29	1.38	0.96	0.72	0.51	0.37
Rijndael	0.03	0.01	0.02	0.02	0.00	0.01
Rijndael (OPT)	0.00	0.00	0.00	0.01	0.01	0.00
TCP	6.01	2.50	1.63	1.36	0.82	0.55
TCP (OPT)	4.03	1.77	1.29	0.84	0.70	0.41
Marshal	4.97	0.29	0.10	0.01	0.02	0.00
Ping	16.68	1.16	0.35	0.08	0.03	0.00

Fig. 2. Reduction of false alarms in regression benchmarking for different numbers of compilations. The same values are presented both in the graph and in the table.

single binary for benchmarking can lead to severe distortion of the benchmark results.

We introduce a new statistical model of a benchmark experiment, one which allows to estimate the precision of benchmark results, taking into account the random effects in compilation, but also the random effects in benchmark execution described in [2] and the widely known random effects in individual measurements. In addition to this, the model makes it possible to determine the optimum number of measurements within each benchmark execution and the optimum number of executions for each benchmark binary, which allows us to achieve the best possible precision for a given time limit on the benchmark experiment.

As an application of the model, we demonstrate a significant reduction of the number of erroneously detected performance changes between different versions of the same software in the context of regression benchmarking [4]. As a striking example, with 25 Mono binaries, the number of erroneous detections using a standard numerical benchmark falls down from 50% to 6%, as illustrated in Figure 2. This improvement is achieved by incorporating the random effects of compilation into the precision estimates of the results.

There are numerous related projects that track performance changes during software development, such as [14, 15]. Although these projects do not attempt to detect the changes in performance automatically, their results would benefit from using the proposed statistical model. At the time of this writing, we are not aware of any other project that would attempt to handle the problems associated with random effects of compilation in performance.

Acknowledgement. This work was partially supported by the Grant Agency of the Czech Republic project GD201/05/H014 and the Czech Academy of Sciences project 1ET400300504.

References

1. Smith, C.U., Williams, L.G.: Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software. Addison–Wesley, Reading, MA, USA (2001)
2. Kalibera, T., Bulej, L., Tuma, P.: Benchmark precision and random initial state. In: Proceedings of SPECTS 2005, SCS (2005) 853–862
3. Kalibera, T., Bulej, L., Tuma, P.: Automated detection of performance regressions: The Mono experience. In: MASCOTS, IEEE Computer Society (2005) 183–190
4. Bulej, L., Kalibera, T., Tuma, P.: Repeated results analysis for middleware regression benchmarking. Performance Evaluation **60** (2005) 345–358
5. Lo, S.L., Grisby, D., Riddoch, D., Weatherall, J., Scott, D., Richardson, T., Carroll, E., Evers, D., , Meerwald, C.: Free high performance orb. http://omniorb.sourceforge.net (2006)
6. Novell, Inc.: The Mono Project. http://www.mono-project.com (2006)
7. ECMA: ECMA-335: Common Language Infrastructure (CLI). ECMA (2002)
8. Distributed Systems Research Group: Mono regression benchmarking. http://nenya.ms.mff.cuni.cz/projects/mono (2005)
9. Free Software Foundation: The gnu compiler collection. http://gcc.gnu.org (2006)

10. Gu, D., Verbrugge, C., Gagnon, E.: Code layout as a source of noise in JVM performance. In: Component And Middleware Performance Workshop, OOPSLA 2004. (2004)
11. Wasserman, L.: All of Statistics: A Concise Course in Statistical Inference. Springer, New York, NY, USA (2004)
12. Jain, R.: The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling. Wiley–Interscience, New York, NY, USA (1991)
13. Buble, A., Bulej, L., Tuma, P.: CORBA benchmarking: A course with hidden obstacles. In: IPDPS, IEEE Computer Society (2003) 279
14. DOC Group: TAO performance scoreboard. http://www.dre.vanderbilt.edu/stats/performance.shtml (2006)
15. Prochazka, M., Madan, A., Vitek, J., Liu, W.: RTJBench: A Real-Time Java Benchmarking Framework. In: Component And Middleware Performance Workshop, OOPSLA 2004. (2004)
16. Weisstein, E.W.: Mathworld–a wolfram web resource. http://mathworld.wolfram.com (2006)

A Proof of Lemma 33

Let f be the probability density function of the normal distribution with mean μ and variance σ^2:

$$f(x;\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right), \quad exp(z) = e^z.$$

The density functions of X and $Y|X$ from Lemma 33 are:

$$f_X(x) = f(x;\mu_X,\sigma_X), \quad f_{Y|X}(y|x) = f_{Y|x}(y) = f(y;x,\sigma).$$

By the definition of conditional density:

$$f_{Y,X}(y,x) = f_{Y|X}(y|x)\cdot f_X(x).$$

It follows, that:

$$\begin{aligned} f_Y(y) &= \int f_{Y,X}(y,x)\,dx = \int f_{Y|X}(y|x) f_X(x) dx = \\ &= \int \frac{1}{\sigma\sqrt{2\pi}} exp\left(-\frac{(y-x)^2}{2\sigma^2}\right)\cdot\frac{1}{\sigma_X\sqrt{2\pi}} exp\left(-\frac{(x-\mu_X)^2}{2\sigma_X^2}\right) dx = \\ &= \int \frac{1}{\sigma\sqrt{2\pi}} exp\left(-\frac{(y-\mu_X-u)^2}{2\sigma^2}\right)\cdot\frac{1}{\sigma_X\sqrt{2\pi}} exp\left(-\frac{u^2}{2\sigma_X^2}\right) du = \\ &= \int f(y-u;\mu_X,\sigma) f(u;0,\sigma_X) du. \qquad |u = x-\mu_X \end{aligned}$$

Lemma A1 *Let $f(t;\mu_1,\sigma_1)$,$f(t;\mu_2,\sigma_2)$ be density functions of normal variates. Then,*

$$\int f(\tau;\mu_1,\sigma_1) f(t-\tau;\mu_2,\sigma_2)) d\tau = f\left(t;\mu_1+\mu_2,\sqrt{\sigma_1^2+\sigma_2^2}\right).$$

In other words, convolution of Gaussians is also a Gaussian (Convolution, [16]).

By Lemma A1:

$$f_Y(y) = \int f(y-u;\mu_X,\sigma)f(u;0,\sigma_X)du = f\left(y;\mu_X,\sqrt{\sigma^2+\sigma_X^2}\right),$$

and thus

$$Y \sim N(\mu_X, \sigma_X^2+\sigma^2). \quad \square$$

B Proof of (21) and (23)

We will show only (23), because (21) is a special case of (23), where $q = 1$. Let f,g be defined as follows:

$$g(l,m,n) = (b+(w+n)\cdot mq)\cdot l - c,$$
$$f(l,m,n) = \frac{S_E^2}{lmn} + \frac{S_B^2}{lm} + \frac{S_V^2}{l}.$$

Our objective is to find a minimum of $f(l,m,n)$, subject to the constraint $g(l,m,n) = 0$. Using Lagrange Multiplier Theorem [16], we can find l,m,n where the minimum must be, provided that the minimum exists. The partial derivatives are:

$$\left(\frac{\partial g}{\partial l},\frac{\partial g}{\partial m},\frac{\partial g}{\partial n}\right)(l,m,n) = \left((w+n)\cdot mq + b, (w+n)\cdot ql, mql\right),$$
$$\left(\frac{\partial f}{\partial l},\frac{\partial f}{\partial m},\frac{\partial f}{\partial n}\right)(l,m,n) = \left(-\frac{S_E^2}{l^2mn} - \frac{S_B^2}{l^2m} - \frac{S_V^2}{l^2}, -\frac{S_E^2}{lm^2n} - \frac{S_B^2}{lm^2}, -\frac{S_E^2}{lmn^2}\right).$$

By Lagrange Multiplier Theorem, the local extremum can only be in l,m,n, that solve the following system of equations:

$$\frac{\partial f}{\partial l}(l,m,n) + \lambda\frac{\partial g}{\partial l}(l,m,n) = 0 \tag{24}$$
$$\frac{\partial f}{\partial m}(l,m,n) + \lambda\frac{\partial g}{\partial m}(l,m,n) = 0 \tag{25}$$
$$\frac{\partial f}{\partial n}(l,m,n) + \lambda\frac{\partial g}{\partial n}(l,m,n) = 0 \tag{26}$$
$$g(l,m,n) = 0 \tag{27}$$

We can express m^2 and l^2 from (26), for $\lambda > 0, q > 0$:

$$m^2 = \frac{S_E^2}{\lambda q l^2 n^2}, \quad l^2 = \frac{S_E^2}{\lambda q m^2 n^2}. \tag{28}$$

By substituting m^2 from (28) into (25), we get for $n > 0$:

$$n_0 = n = \sqrt{\frac{wS_E^2}{S_B^2}}.$$

By substituting l^2 from (28) into (24), we get for $m > 0, w > 0$:

$$m_0 = m = \sqrt{\frac{bS_B^2}{qwS_V^2}}.$$

We are not interested in the values of l and λ solving the system of equations. Still, it remains to be shown that there really is a local minimum of $f(l, m, n)$ in $m = m_0, n = n_0$. This can be done directly by checking the first and the second partial derivatives of $f(m, n)$,

$$f(m, n) = \frac{mqw + mqn + b}{c} \cdot \left(\frac{S_E^2}{mn} + \frac{S_B^2}{m} + S_V^2 \right),$$

as described in Second Derivative Test [16]. The procedure is quite straightforward, but involves some labor algebra. We do not include the details here.

C Description of Used Benchmarks

All benchmarks were run on a single machine, Dell Precision 340, with a single Pentium 4 processor, 512M RAM. The CORBA benchmarks were run on Fedora 2 operating system, the Mono benchmarks were run on Fedora 4. All benchmarks were run with a disconnected network interface and with all unnecessary system services shut down.

The Ping benchmark measures the response time of a simple CORBA remote procedure call, the Marshal benchmark measures only marshaling part of the remote call. Both benchmarks comprise of a client and a server process, both of which are restarted in each execution. The evaluation was done with 100 CORBA/benchmark binaries, each benchmark binary was executed 25 times. The Ping and Marshal benchmarks are described in [2] in more detail, including the platform information.

The other benchmarks are from the Mono Regression Benchmarking Project [8]. The TCP Ping and HTTP Ping benchmarks measure response time of a single remote procedure call using TCP and HTTP channels, both benchmarks comprise of two processes. The Rijndael benchmark measures the aggregated time for encryption and decryption of a constant short text in memory. The FFT benchmark measures the aggregated time for forward and inverse Fast Fourier Transformation of a constant vector. There are two versions of the FFT benchmark: the original version allocates the memory for computation repeatedly at the beginning of each measurement, the NA ("no allocation") version allocates the memory once at the benchmark process start–up. Each benchmark was run both with the default virtual machine optimizations turned on, and with all the implemented virtual machine optimizations turned on (OPT). The evaluation was carried out with 150 binaries, each benchmark binary was executed 100 times. Detailed description of the benchmarks and platform information are available on the web [8].

Chapter 8

Generic Environment for Full Automation of Benchmarking

Tomáš Kalibera, Lubomír Bulej, Petr Tůma

Contributed paper at **First International Workshop on Software Quality (SOQUA 2004)** [4] of Net.ObjectDays 2004 conference.

In conference proceedings,
published by tranSIT GmbH,
pages 35–41,
ISBN 3-9808628-3-6,
September 2004.

Also in *Testing of Component-Based Systems and Software Quality*,
published by Gesselshaft für Informatik,
Lecture Notes in Informatics P-58,
pages 125–132,
ISSN 3-88579-387-3,
December 2004.

The original version is available electronically from the publisher's site at http://www.gi-ev.de/fileadmin/redaktion/2004_LNI/PDF/p-58.pdf.

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.

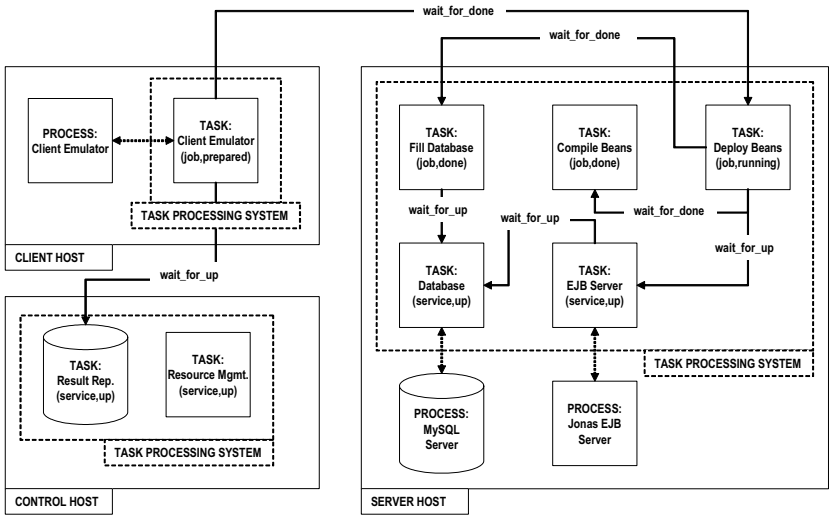


Figure 1: Example configuration of the benchmarking environment running the RUBiS benchmark.

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.

Chapter 9

Automated Benchmarking and Analysis Tool

Tomáš Kalibera, Jakub Lehotský, David Majda, Branislav Repček, Michal Tomčányi, Antonín Tomeček, Petr Tůma, Jaroslav Urban

Technical report [12] at Department of Software Engineering, Faculty of Mathematics and Physics, Charles University in Prague.

Published as technical report,
no. 8,
June 2006.

The paper is available electronically from http://nenya.ms.mff.cuni.cz/publications/Submitted_1404_BEEN.pdf.

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

{kalibera,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).

I. 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 [1]–[3]. This technique, also known as regression benchmarking [4], 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 [1]–[3], 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 [5], [6]. The random effects are present in real systems, have impact on performance and cannot be filtered out [5]. 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 [6].

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

The authors are listed in alphabetic order.

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 [7]. Currently, BEEN is available in a beta version, described in this paper in more detail. The related projects are described in Section II. The architecture of the tool is outlined in Section III and the functionality is detailed in Sections IV and V. The current implementation is evaluated on a distributed remote procedure call benchmark in Section VI. The paper is concluded in Section VII.

II. 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 [1], A Real-Time Java Benchmarking Framework [2], Lockheed Martin ATL Benchmarking Tools [8] and Mono Regression Benchmarking Project [3]. These tools were all created for use in a particular software project. [1] and [8] are focused at CORBA, [2] is for Java applications and [3] is for Mono/C# applications. Only [3] features automated detection of changes. Porting the tools for use in other software projects would require a significant additional effort.

The Skoll Project [9] 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 [10], 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 [11] 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 [11] for load injection, adding runtime monitoring, results repository and automated evaluation of results. The results repository of [11] is limited in comparison.

The NIST Automated Benchmarking Toolset [12] 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 [13] 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.

III. 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.

A. 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. The Host Manager 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, 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.

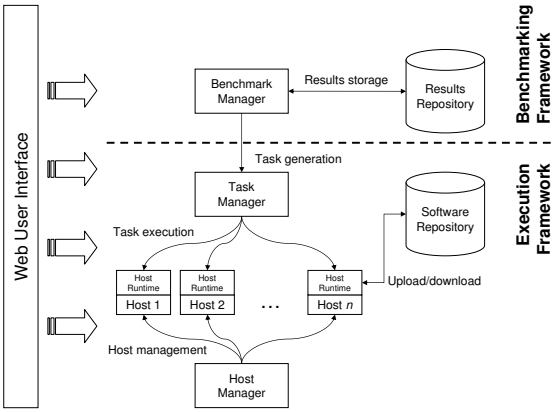


Fig. 1. BEEN architecture.

B. 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.

C. 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.

IV. 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 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. The server task 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.

A description of the individual components of the execution framework follows.

A. 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 locally 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 as a part of the task context, 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.

B. 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 IV-C 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.

C. 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.

D. 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. The supported package types are:

- *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.

V. Benchmarking Framework

The benchmarking framework supports fully automated benchmarking on top of the distributed execution framework described in Section IV. 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 [5]. 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 [6], [14].

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.

A detailed description of the Benchmark Manager and the Results Repository follows.

A. 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 warmup 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.

B. 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 [15] 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 [5], [14]. 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 [16], 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 Reposi-

tory 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 [5], [14], 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 [6], [14]. 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.

VI. 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 [17], 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 [18]. 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 [19]. Detailed step–by–step instructions to perform the evaluation, as well as screenshots of the user interface running the evaluation, are also available [19]. A more thorough evaluation will be in order as the BEEN implementation matures and is used for more benchmarks on more platforms.

VII. Conclusion

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 [19]. Future work will focus on finishing the implementation so that more types of analysis and more benchmarks on more platforms are readily supported.

Acknowledgment. This work was partially supported by the Grant Agency of the Czech Republic project GD201/05/H014 and the Czech Academy of Sciences project 1ET400300504.

REFERENCES

[1] DOC Group, "TAO performance scoreboard," http://www.dre.vanderbilt.edu/stats/performance.shtml, 2006.

[2] 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.

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

[4] L. Bulej, T. Kalibera, and P. Tuma, "Repeated results analysis for middleware regression benchmarking," *Performance Evaluation*, vol. 60, no. 1–4, pp. 345–358, May 2005.

[5] T. Kalibera, L. Bulej, and P. Tuma, "Benchmark precision and random initial state," in *Proceedings of the 2005 International Symposium on Performance Evaluation of Computer and Telecommunications Systems (SPECTS 2005)*. San Diego, CA, USA: SCS, July 2005, pp. 853–862.

[6] ——, "Automated detection of performance regressions: The Mono experience." in *MASCOTS*. IEEE Computer Society, 2005, pp. 183–190.

[7] ——, "Generic environment for full automation of benchmarking," in *SOQUA/TECOS*, ser. LNI, S. Beydeda, V. Gruhn, J. Mayer, R. Reussner, and F. Schweiggert, Eds., vol. 58. GI, 2004, pp. 125–132.

[8] Advanced Technology Labs, Lockheed Martin Corp, "Agent and distributed objects quality of service," http://www.atl.external.lmco.com/projects/QoS, 2006.

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

[10] 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*, G.-C. Roman, W. G. Griswold, and B. Nuseibeh, Eds. ACM, 2005, pp. 293–302.

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

[12] M. Courson, A. Mink, G. Marçais, and B. Traverse, "An automated benchmarking toolset." in *HPCN Europe*, ser. Lecture Notes in Computer Science, M. Bubak, H. Afsarmanesh, R. Williams, and L. O. Hertzberger, Eds., vol. 1823. Springer, 2000, pp. 497–506.

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

[14] T. Kalibera and P. Tuma, "Precise regression benchmarking with random effects: Improving mono benchmark results," Accepted for European Performance Engineering Workshop (EPEW 2006)., Mar. 2006.

[15] University Corporation for Atmospheric Research, "Network Common Data Form (NetCDF)," http://www.unidata.ucar.edu/software/netcdf, 2006.

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

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

[18] S.-L. Lo, D. Grisby, D. Riddoch, J. Weatherall, D. Scott, T. Richardson, E. Carroll, D. Evers, , and C. Meerwald, "Free high performance orb," http://omniorb.sourceforge.net, 2006.

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

Chapter 10

Intelligent Source Dependency Tool

Tomáš Kalibera

Contributed paper at **13th ALADIN workshop on ALADIN applications in very high resolution** [5], non-refereed.

In workshop proceedings,
published by Czech Hydrometeorological Institute,
pages 73–81,
ISBN 80-86690-13-X,
November 2003.

Intelligent Source Dependency Tool

Tomáš Kalibera

Czech Hydrometeorological Institute, Na Šabatce 17, 143 06 Prague, Czech Republic

*Distributed Systems Research Group, Department of Software Engineering,
Faculty of Mathematics and Physics, Charles University, Malostranské nám. 25, 118 00 Prague,
Czech Republic*

*tomas.kalibera@chmi.cz,
tomas.kalibera@mff.cuni.cz*

Abstract

With the fast evolution of Aladin/Arpege source code, the complexity of actions needed to test even simple modifications of source code becomes inadequate. However, it is possible to automatize most of these actions based on the knowledge of dependencies in the source code. The source dependency tool developed at CHMI automatically detects and manages dependencies of mixed Fortran 90/C source code and handles huge applications efficiently, being an essential part of an automatized compilation environment of Aladin/Arpege at CHMI.

1. Introduction

Knowing well that it is easier to find and fix errors earlier during application development, developers prefer to test relatively small modifications of code. Given this common practice, a fast and automatized procedure or a build tool that compiles the modified sources and creates a new application binary, becomes a necessary part of the development environment.

Although such a build tool seems to be naturally required by huge applications from the very beginning of programming, there are surprisingly no really usable generic tools currently available that would be suitable for large applications like Aladin/Arpege. This important lack of an essential tool can be caused by several factors. One of them is that the requirements for such a tool are often very specific for different applications and institutions, as they impose different

sets of coding conventions affecting dependencies (like whether the header files are included by relative path name or only by filename). Another reason is perhaps that computer scientists who design build tools use programming languages like C or Java, not Fortran. This preference becomes a problem because of Fortran 90 specification and its concept of modules, that introduces complex types of dependencies between source files. These dependencies have no counterparts in C or Java applications.

Although there are no usable generic build tools for Fortran 90 code, a huge part of the problem is solved by a well known and often inadequately rejected tool, Make [3]. Make is an expert system that, given a list of rules and dependencies, controls the process of making a target (or application) up to date. Make is easy to use, but developers often do not use it efficiently. When employed for building applications, Make requires a list of dependencies between source files on input. Trivial scripts that get dependencies of a single Fortran source files exist, but they do not scale well for applications of size of Aladin/Arpege.

The source dependency tool developed at CHMI manages mergeable databases of source dependencies of mixed Fortran 90/C language applications. These databases can be exported into the form required by Make. During the development of Aladin/Arpege, it has become very popular to link the application binary from a list of not necessarily disjunct libraries (or"patch" libraries), replacing equally named objects by the version provided in the first library that defines the object, in the given order of precedence. Although this is a really error prone practice, as one does not know all internals of the patched subroutines, it is also a common practice in Aladin/Arpege development. The dependency tool can be used to generate a list of files that are needed to compile such a patch library, fetching information from the dependency database.

More applications of the dependency tool include creation of a compilable subset of the application, for example to provide a supercomputer benchmark suite. Knowing the source dependencies may also help developers to get more insight into the code, such as finding by what files is the given module used.

This paper follows by a general analysis of dependencies in source code in section 2 accompanied by language specific analysis for the C language in section 2.1 and for Fortran 90 language in section 2.2. An abstract model of source dependencies is proposed in section 2.3. The tasks required for Aladin/Arpege development that can be accomplished knowing the source dependencies are described in section 3. A short overview of the dependency tool implementation and its integration to the versioning system interface at CHMI is provided in section 4. The paper is concluded in section 5.

2. Dependencies in source code

In the source code of applications, there are many types of dependencies. Some of them are complicated and it does not make sense to describe them by other means than the programming language, these can be for example dependencies of variables, where a variable contains a result of a computation that requires values of other variables on input. Such dependencies belong to the application logic, it is the responsibility of the application to recompute the result when the input values change. And it is the responsibility of the compiler to check that the variables used as inputs are defined and at least initialized.

Finally, it is the responsibility of the programmer to advice the compiler in which files the variables are defined (including a header or indirectly using a module in Fortran). The build tool then takes this piece of advice and does not check whether the dependency of any variables from the referenced file really exists. A common strategy for a dependency tool is to give a superset of dependencies, not omitting any, but possibly including some that do not really exist under the given conditions. Actually, variable dependencies may become important also at link time, when the linker must check that global variables are defined in some of the given object files. It is a good practice to embody access to such variables into elements such as objects, modules or at least functions and subroutines. Then only link time dependencies of those embodying elements have to be analyzed.

More dependencies in source code are introduced by functions (or subroutines, procedures). Functions can have similar types of dependencies as variables, but in case of functions, the link time dependencies cannot be avoided (in pure object oriented languages, functions are embodied by objects, for which the link time dependencies are then important). It is the responsibility of the build tool to provide linker with a list of object files that include code of the used functions, the linker only checks that the list is complete. In order to get the link time dependencies automatically, the build tool would have to parse the source code of the whole application. There is no piece of advice provided by programmers where the functions are defined in Fortran or C, unlike with compile time dependencies, where headers and modules must be specified. Therefore the link time dependency detection at the level of source files has to be incorporated deeper into the compiler.

Another approach to this problem is to get link time dependencies by traversing already generated object files or even to first link the binary with all possible object files, and then check which of them were actually used. The following analysis and the CHMI implementation focus only on compile time dependencies at the granularity of a source file.

2.1. Dependencies in C code

It is a good practice to separate the definition (source code) of a function from the code that uses the function, making the source code easier to read and allowing existence of libraries and reuse of function's code. In C language, before a library (external) function is used, it must be declared. These declarations, together with definitions of types of arguments of the function, are usually placed in separate files (header files) which are then referenced by "#include" directive from the C source files, both by the C file that implements the function as well as from the C file that uses it.

A C language compiler suite consists of several parts, one of which is a C preprocessor that just includes code of the headers into the code of source files that use it by the "#include" directive. The C preprocessor also implements a simple macro language. The header files may use other header files, recursively. There can be circular dependencies of header files, a header file may include itself via a chain of any number of included header files, the indefinite recursion of the C preprocessor must be avoided by the macro language.

The headers are identified by file names provided with the "#include"directive. The file names either should not contain path names (this is the case of Aladin/Arpege code), or it should preferably contain path names relative to the project directory tree. In both cases, it is quite simple to find the header file because its name is known, however in the first case possibly all directories of the project have to be scanned, although few of them really contain headers in Aladin/Arpege. By a convention honored by Aladin/Arpege code, header files have suffix ".h". During initialization the dependency tool can detect directories containing header files and then while processing the sources it only has to look for header files in previously detected directories.

As the C language is traditionally used for operating system development, core system libraries are written in C and have a C interface defined in system header files. Along with its own header files, applications need to include also system header files in order to have access to basic system functions, such as file input/output. The dependency tool must be aware of the existence of the system headers and must be aware that the structure and location of these headers differ greatly for different operating systems and platforms. The dependency tool must be provided with the list of directories that contain the system header files.

2.2. Dependencies in Fortran code

Although Fortran is a predecessor of C language, it is continuously being extended by important

features of the C language and modern programming languages. Fortran has its own directive for plain text inclusion of a file, but current Fortran applications including Aladin/Arpege use a more popular and more powerful C preprocessor. A slightly modified C preprocessor is currently a typical part of many Fortran compiler suites. Therefore dependencies introduced by header files described in section 2.1 apply fully to Fortran code, too.

An important improvement of the Fortran language was provided by the specification of Fortran 90, which introduced modules into the language. Modules are important language abstractions which allow to group related functions and subroutines together with variables they use, introducing a finer structure into the application and making it easier to read, manage and reuse.

Modules are defined in Fortran sources and are compiled by the Fortran compiler into module information files. Multiple modules can be defined in a source file, but programmers commonly place modules into separate files, each containing only a definition of a single module. Each module has its name defined in the source code that is also employed by the source code that uses the given module. During compilation of modules, for each module the compiler creates a module information file which has the same name as the module (with some extension, like ".mod"). The name of the file the module (or modules) was defined in is not important to the compiler, it is not reflected in the names of files generated by the compiler.

Although typically files that contain a single module definition have the same name as the module (with ".F90" or similar extension), not all programmers follow this principle. Therefore in general it is very hard to find a module source for the given module (a module given by name), it is necessary to parse all source files of the project. A dependency tool can accomplish this task during its initialization.

The Fortran compiler does not solve the problem of locating module sources. The build tool must inform the compiler in which directories module information files are stored before a source file that uses any modules can be compiled. It means that the build tool also has to make sure that the module information files exist before the source file that use the corresponding module can be compiled. This procedure becomes more complicated as modules can recursively use other modules. The build tool must be aware that the module information file has to be recompiled whenever the module source was modified.

The use of modules can be intermixed with the use of headers in Fortran source and the dependency tool must be aware of that, too.

2.3. Model of dependencies

The compile time source dependencies common in mixed Fortran 90/C language source code, described in the previous sections, must be handled by a dependency tool. The dependency tool should be simple enough to be manageable, reliable and extensible, as more types of dependencies can be introduced in the future. To achieve these goals, it is helpful to keep aside precision. It is fine if the build tool sometimes decides to recompile some files that for some specific technical reasons need not to be recompiled, as long as it is guaranteed that all files that have to be recompiled for any reason are really recompiled.

The nature of compile time source file dependencies is simple, file A "uses" file B. In our situation B would be a header or a module. The build tool requires two basic operations from the dependency tool. For a given file A, it needs to know all files B, such that A uses B. The build tool uses this information to make sure that the headers or modules file A uses are not modified. In case they were modified, the build tool would have to recompile file A. Suppose that A is also a module. Then after A is recompiled, all source files that use A have to be recompiled, too. Therefore, the build tool requires also that the dependency tool for a given file name A returns all files it is used by, it means names of all files X such that X uses A.

In summary, two basic operations of the dependency tool defined on a source file are **is_used_by()** and **uses()**. These operations return only directly used files, the task of finding all used files recursively can be accomplished by multiple invocations of these operations. It is described in the following chapter how the built tool uses these operations in more detail, focusing on Aladin/Arpege development requirements.

However, the underlying data model of the dependencies is rather more complicated. It was already mentioned that some source files use platform dependent system headers which are not part of the application source. These files are called external headers and a list of them can be obtained by **uses_external_header()** operation. Moreover, the Aladin/Arpege sources are divided into projects such as Arp, Ald, Xrd, Ta, Tf, etc. These projects use modules and headers defined in other projects, but for each application, different combination of these projects may be required.

The dependency tool manages dependency databases (or caches), each database contains dependencies of a single project. When used for Aladin/Arpege, the projects are those named above, Arp, Ald, etc. Although the dependency database could be built for each used combination of projects from scratch, it would be inadequately time consuming. Instead, the dependency tool is capable of merging multiple dependency databases into a single one. During the merge operation, the dependency tool employs operations **uses_external_header()** and

uses_external_module(), so that it does not need to re-parse the source files.

The build tool then operates on a dependency database that contains only external references to system headers and no external references to modules. In the automatized compilation environment of Aladin/Arpege in CHMI, such a database is a result of a merge operation on multiple dependency database for projects like Arp, Ald, Xrd, etc.

3. Compilation driven by dependencies

The aim of the construction of an automatized build tool is to provide developers with a fast and reliable means for iterative development, improving the software a little bit and testing immediately. Assuming the scale of Aladin/Arpege source code, with more than 30 megabytes of source code and an executable file of 100 megabytes, it is necessary to recompile only the subset of the application altered by the modifications. It means that all files that were modified as well as all files that use them, recursively, must be recompiled.

The actual control of compilation can be accomplished by running Make. Although with the dependency database capable of **is_used_by()** operation it would be simple to implement a trivial control of compilation, Make can do much more, allowing parallel compilation, compilation of all files that can be compiled without error, etc. Still, the dependency database is suitable for creation of dependency information files to be imported by Make. Although there are many freely available scripts that generate such dependency information files for a single source file, and one such script was developed as a part of the automatized compilation environment at CHMI, running such script sequentially for all source files is too time consuming, repeating tasks that can be cached over and over again.

3.1. Order of compilation

The dependency information files for Make must also describe dependencies of modules, instructing Make to compile Fortran sources only after module information files for modules it uses are available. The necessity of compiling modules first distinguishes Fortran from traditional languages like C, in which the order of compilation of individual source files can be arbitrary.

3.2. Extraction of subapplications

In the field of weather numerical forecasting, it is important to select appropriate hardware for the computations carefully. One of the most popular and transparent ways to choose the best platform is to provide the hardware vendors with a small benchmark application and ask them to build hardware that would run the benchmark as fast as possible. Such a benchmark will be a functional subset of the Aladin/Arpege source. The dependency database can help the benchmark designer to add all necessary files (modules and headers) to the benchmark, so that it could be compiled.

Moreover, the dependency database can be employed by a tool that would generate a build script for the benchmark. Such a build script would compile the benchmark in the correct order, modules before files that use them, without requiring the vendors to install full development environment.

It is important to note that having a tool that could also handle link time dependencies would help with the construction of the benchmarks even more. However, benchmark construction is quite a rare task, and thus tools to support it are currently not of a highest priority.

3.3. Patch creation

The most important option provided by the dependency tool is to allow the user to get a small set of compilable files that fully describe the modification (a patch). These files can be compiled into object files and packaged into a patch library, that can be then used to re-link the application binary, getting the same result as if the application was re-compiled and re-linked in a conventional way.

The described method of getting an up-to-date binary is based on an assumption that there is a system wide repository of ready to use (static) libraries containing object files for individual projects, like Arp, Ald, Xrd, etc. Those libraries are created from well known releases, usually cycles and bugfixes. The developers start with sources of those well known releases and modify them. Once they are done or want to test their work, they create a patch, compile it separately, and link it using the method described above.

The patch contains all files the developer has modified from the release version plus all files used by them, recursively, plus all files required to compile the patch, recursively. In a more formal way, let P be the set of patch files. The algorithm to construct it is as follows:

1. add files modified by user to P
2. for all files F from P, add transitive closure of F.**is_used_by()** to P

3. for all files F from P, add transitive closure of F.**uses()** to P

The patch constructed in this way compiles independently (3) and contains updates for all possibly altered files (1),(2).

4. Implementation overview

The source dependency tool described in the previous sections was implemented and is in use at CHMI for 2 years, being a base part of the automatized compilation environment for Aladin/Arpege at CHMI. While the compilation environment is tied with the local computing environment and is not mature yet, the dependency tool itself is quite independent and bugs are rarely found in it.

The dependency tool is implemented in Perl as a set of modules. Each dependency database is stored in a single binary file, it is a serialized form of a nested Perl hash structure. The serialization/deserialization is accomplished by CPAN [1] module Storable. The performance of the Perl implementation is very good, while its source code is still maintainable. Building a dependency database for 7 Aladin/Arpege projects having 34 megabytes of sources in total takes less than 10 seconds on a COMPAQ Proliant Intel/Linux SMP server. Creating a patch including dependency information makefiles for project Ald takes less than 3 seconds.

The dependency tool has a command line interface allowing to create a dependency database, merge multiple dependency databases and create a patch. The patch creation can be accompanied by generation of dependency information files for Make. The form of these files is the only local specific feature of the tool, it is tied with the Make strategy of makefiles used at CHMI which is not the topic of this paper. Its features include correct parallel compilation, inter-project dependencies and compiler options specific to projects. Recursive invocation of Make is strongly avoided, as suggested in [4].

The practical usability of the dependency tool was verified by its integration into the configuration management tool used at CHMI called CVSTUC, an implementation of a subset of the interface provided by TUC in Meteo France, which uses CVS [2] instead of Rational ClearCase. The cc_depend command of CVSTUC is just a wrapper of the dependency tool patch creation functionality.

CVSTUC keeps a cache of dependency databases of all official releases of individual Aladin/Arpege projects in a world readable directory. At the same time, it keeps caches of dependency databases of the CVSTUC views of each user in the user's home directory. Before

actually calculating the dependencies, cc_depend merges the user's cache of a dependency database with dependency databases of the projects the user chooses to build with, no matter whether those are the user's own projects or the official ones. The use of cached dependency databases is fully transparent to CVSTUC users.

As a result, cc_depend executes really fast, taking several seconds to complete on the PC described above. The speed of the execution is gained by using the right data structures, hashes in Perl, and by reading whole databases into memory. The amount of memory required scales with the number of source files and their inter-dependencies in the projects, but only linearly. The tool may use tens of megabytes of memory for Aladin/Arpege. It is unlikely that the number of source files and their inter-dependencies of Aladin/Arpege would grow faster than the amount of memory available in cheap desktop hardware.

There was an attempt to implement the same functionality in bash scripts. The database was stored in separate files, for each source file keeping a list of files it uses and list of files it is used by. The scripts were hard to maintain and took minutes to run.

During the development of the dependency tool, many problems concerning lazy to no coding conventions arose. Clearly the coding conventions shall never be designed only to make development of a dependency tool easier, but strong coding conventions should exist to make the code easier to understand, most of all by programmers. It will make it easier to understand by the tools as well. One of the fundamental problems in Aladin/Arpege is the ambiguity between projects. There are duplicate modules and headers in projects. At the design time of the dependency tool, it was assumed that modules and headers are unique in the whole code. Later it turned out the assumption did not hold. The current implementation doesn't handle ambiguous modules and headers seamlessly. However, the existence of modules or headers with the same name in more projects is highly suspicious and undesirable.

Although the situation is solved in the current implementation, it is still worth to note that all but one of the modules in Aladin/Arpege are defined in single files named after the modules. The existence of the single module (fullpos_types) defined in a file named differently (fullpos_descriptors.F90) required a major implementation rewrite of the dependency tool. It was not possible to say that the file is named incorrectly because there are no conventions accepted that it broke. If there was a convention that modules had to be defined in separate files named after the modules, it would be easier for the developer and the dependency tool as well to find the source file for each module.

Some of the new types of dependencies cannot be handled by a dependency tool. These are dependencies not defined in the sources, but rather by command line options to the compiler. Such dependencies include inline functions in the NEC/SX Fortran compiler. With this compiler,

it can be specified at compile time which functions have to be inlined and the compiler requires sources of such functions.

5. Conclusion

The Aladin/Arpege developers are often distracted from their work by technical problems they have to solve while compiling and linking the application. During the years of development, the task of building Aladin/Arpege projects has become complex, but it can still be automatized so that developers can spent more time and effort doing their work.

A robust source dependency tool that can manage information on compile time dependencies between source files and calculate transitive closures for both altered and required files is an essential part of any automatized build tool. The dependency tool, although well tested for Aladin/Arpege, should be general enough to work with any project written in C and Fortran 90 languages, so that it does not have to be upgraded with Aladin/Arpege itself. And finally the tool must be fast enough to achieve high responsiveness in interactive work on traditional desktop hardware, having the size of Aladin/Arpege project in mind.

Such a dependency tool was developed at CHMI and is used regularly, being integrated into the automatized compilation and configuration management environment of Aladin/Arpege. The tool is capable not only of calculating dependencies within Aladin/Arpege projects (like Arp, Ald, Xrd), but also inter-project dependencies. The tool is written fully in Perl and is easily portable. Open issues include rigorous handling of duplicate modules and headers.

6. References

[1] Comprehensive Perl Archive Network, http://www.cpan.org

[2] Concurrent Versioning System, http://www.cvshome.org

[3] GNU Make, http://www.gnu.org/manual/make

[4] Miller P., Recursive Make Considered Harmful, AUUGN Journal of AUUG Inc., 19(1), pp. 14-25, http://aegis.sourceforge.net/auug97.pdf

Chapter 11

Distributed Component System Based On Architecture Description: The SOFA Experience

Tomáš Kalibera,
Petr Tůma

Contributed paper at **Fourth International Symposium on Distributed Objects and Applications (DOA 2002)** [10], acceptance rate 25%.

In *On the Move to Meaningful Internet Systems 2002: CoopIS, DOA, and ODBASE*,
published by Springer-Verlag,
LNCS 2519,
pages 981–994,
ISSN 0302-9743,
October 2002.

The original version is available electronically from the publisher's site at
http://www.springerlink.com/link.asp?id=bjqd4eh732awvyk3.

Distributed Component System Based On Architecture Description: The SOFA Experience

Tomáš Kalibera and Petr Tůma

Charles University
Faculty of Mathematics and Physics,
Department of Software Engineering
Malostranské náměstí 25, 118 00 Prague 1,
Czech Republic
kalibera@nenya.ms.mff.cuni.cz petr.tuma@mff.cuni.cz

Abstract. In this paper, the authors share their experience gathered during the design and implementation of a runtime environment for the SOFA component system. The authors focus on the issues of mapping the SOFA component definition language into the C++ language and the integration of a CORBA middleware into the SOFA component system, aiming to support transparently distributed applications in a real-life environment. The experience highlights general problems related to the type system of architecture description languages and middleware implementations, the mapping of the type system into the implementation language, and the support for dynamic changes of the application architecture.

Keywords. Architecture description languages, ADL, component definition languages, CDL, middleware, CORBA, language mapping, dynamic architectures.

1 Introduction

The notion of components enjoys significant interest in the software engineering community. Components are considered to be useful *units of code sharing and reuse*, as well as useful *building blocks of software architectures*. While the former view is supported by practical component systems [15, 19, 24], the latter view appears to be lagging behind. The current trend of modeling software architectures using UML is criticized as being inadequate [8], and while the research in component systems based on architecture description languages (ADL component systems) cites inarguable benefits of such systems [1, 5, 13, 14, 23], the described projects rarely get past research prototypes.

The discrepancy between the cited benefits of ADL component systems and the lack of their practical employment leads us to believe that there are unresolved issues that prevent this employment. In order to investigate these issues, we have designed and implemented a runtime environment for the SOFA ADL component system [21] (SOFA environment).

Our chief goal in the design and implementation of the SOFA environment is to support development of transparently distributed applications. The development centers around a hierarchical description of the application architecture. This description is gradually refined from a coarse granularity level, where components correspond to implementation modules, to a fine granularity level, where components correspond to implementation objects. These components are then mapped to implementation objects using a standardized language mapping, with the architecture description defining the interconnection of these objects into the component application. When the application is run, its components can be deployed onto several network hosts. The components that share a host are interconnected through linking and run in one address space. The components that run on different hosts are interconnected through connectors.

The SOFA environment describes the application architecture using the SOFA component definition language [10, 21] (SOFA CDL). SOFA CDL is mapped into C++, which is used to implement the components. The connectors are built using CORBA [18]. The SOFA environment also allows interfacing the application with GNOME [26] to provide user interface support. The choice of GNOME as a representative of a component framework and CORBA as a representative of an off-the-shelf middleware allows us to evaluate how the SOFA environment supports real-life applications in a real-life environment.[1]

The paper continues by a brief introduction of the SOFA component model and SOFA CDL in Sect. 2. The description of the design and implementation of the SOFA environment follows in Sect. 3. Our experience with the CDL to C++ mapping and the integration of CORBA, as well as the ability of the SOFA environment to support applications, is evaluated and generalized for a broad class of ADL component systems in Sect. 4. Section 5 relates this paper to other work in the field of ADL component systems. The paper is concluded in Sect. 6.

2 SOFA Component Model And SOFA CDL

The SOFA component model [21] views an application as a hierarchy of nested software components. A component is an instance of a component *template*, which consists of a component *frame* and a component *architecture*. The frame lists all interfaces that the component *requires* and *provides*. The architecture implements the operations of the provided interfaces, relying only on the operations of the required interfaces. A frame can be implemented by several architectures.

An architecture is either *composed* or *primitive*. A composed architecture defines a *composed component* as built from *subcomponents* by listing the frames of the subcomponents and the *ties* between the interfaces of the component and the subcomponents. A primitive architecture defines a *primitive component* as implemented in an implementation language outside the scope of the component model.

[1] The SOFA ADL component system also includes a Forte IDE and a Java runtime. These are outside the scope of this paper.

A tie between the interfaces of a component and its subcomponents can be of three types. *Binding* denotes connecting a required interface of a subcomponent to a provided interface of a subcomponent. *Delegating* denotes connecting a provided interface of a component to a provided interface of its subcomponent. *Subsuming* denotes connecting a required interface of a subcomponent to a required interface of its component.

An example of the interface, frame and architecture definitions in SOFA CDL is in Fig. 1. The example defines a variation of the ubiquitous "Hello World" application that prints a greeting. The application is an instance of a component with the `ApplicationArch` architecture, which implements the `ApplicationFrame` frame. The `Message` subcomponent provides the greeting to be displayed, the `Display` subcomponent provides the functionality to display a message, the `HelloWorld` subcomponent uses the two other subcomponents to display the greeting. The application defined by the example will be used in other examples throughout the paper.

```
interface MessageIface { string message (); };
interface DisplayIface { void print (in string message); };

frame MessageFrame { provides: MessageIface MessageProv; };
frame DisplayFrame { provides: DisplayIface DisplayProv; };

frame HelloWorldFrame {
  requires: MessageIface MessageReq; DisplayIface DisplayReq;
  provides: ApplicationIface ApplicationProv;
};

architecture MessageArch implements MessageFrame primitive;
architecture DisplayArch implements DisplayFrame primitive;
architecture HelloWorldArch implements HelloWorldFrame primitive;

architecture ApplicationArch implements ApplicationFrame {
  inst MessageFrame Message;
  inst DisplayFrame Display;
  inst HelloWorldFrame HelloWorld;
  bind HelloWorld:MessageReq to Message:MessageProv;
  bind HelloWorld:DisplayReq to Display:DisplayProv;
  delegate ApplicationProv to HelloWorld:ApplicationProv;
};
```

Fig. 1. A SOFA CDL definition of an application architecture.

SOFA CDL can also specify semantics of interfaces and frames using behavior protocols [22] and employ complex connectors [3, 4]. These are outside the scope of this paper.

3 SOFA Environment

The SOFA environment defines and implements a mapping of SOFA CDL into C++ used to map components to implementation objects, implements a deployment mechanism used to deploy the components onto network hosts and to interconnect the components, and implements the connector generator used to produce connectors between components that run on different hosts. These three parts of the SOFA environment are described in this section.

3.1 Mapping SOFA CDL Into C++

The CDL to C++ mapping is based on the IDL to C++ mapping of CORBA [16]. Similar to CORBA IDL, the type system of SOFA CDL is independent of the implementation languages of components and has a standardized mapping into these languages. Making the type system independent on the implementation language makes it easier to generate connectors and potentially also to support multiple implementation languages of components.

The mapping of the types that SOFA CDL shares with CORBA IDL follows the IDL to C++ mapping. The types original to SOFA CDL, namely frames and architectures, are mapped into the frame and architecture classes that follow the approach used to map interfaces with attributes.

A frame class has accessor methods for the provided and required interfaces of the frame, which are represented as protected references to the classes that map the interfaces. An example of a generated frame class is in Fig. 2. To allow substitution of components with the same frame but different architectures, the frame class is a virtual base class that is inherited by architecture classes of the architectures implementing the frame.

```
class HelloWorldFrame : virtual public FrameBase {
  public:
    // Accessor methods generated for provided and required interfaces
    inline virtual ApplicationIface_ptr ApplicationProv () {
      return (ApplicationIface::_duplicate (pApplicationProv)); };
    inline virtual void ApplicationProv (const ApplicationIface_ptr value) {
      pApplicationProv = ApplicationIface::_duplicate (value); };
  protected:
    ApplicationIface_ptr pApplicationProv;
    ...
};
```

Fig. 2. A generated C++ mapping of HelloWorldFrame.

The implementation of an architecture class differs for composed and primitive architectures. An architecture class of a composed architecture has accessor

methods for the subcomponents of the architecture, which are represented as private references to the frame classes of the frames of the subcomponents. The architecture class also contains code that allows to set up the ties between interfaces as defined by the `bind`, `delegate` and `subsume` clauses in the architecture definition.

For performance reasons, the code does not interconnect the interfaces of the composed component with the interfaces of its subcomponents directly. Instead, it allows propagating references to the provided interfaces of primitive components along the ties of the architecture definition by the `createBindingsAndDelegates` and `createSubsumes` methods. The required interfaces of primitive components are thus tied directly to the provided interfaces, with the composed components whose boundaries the ties cross adding no overhead to the invocations of methods accessible through these ties.

An example of a generated composed architecture class is on Fig. 3.

```
class ApplicationArch :
  virtual public ApplicationFrame, virtual public ArchitectureBase
{
  public:
    // Methods generated for setting up the ties between interfaces
    virtual void createBindingsAndDelegates () {
      iHelloWorld->MessageReq (iMessage->MessageProv ());
      iHelloWorld->DisplayReq (iDisplay->DisplayProv ());
      pApplicationProv = iHelloWorld->ApplicationProv (); };
    virtual void createSubsumes () {
      iMessage->createSubsumes ();
      iDisplay->createSubsumes ();
      iHelloWorld->createSubsumes (); };
  private:
    HelloWorldFrame_ptr iHelloWorld;
    ...
};
```

Fig. 3. A generated C++ mapping of ApplicationArch.

An architecture class of a primitive architecture is a virtual base class that the implementation of the primitive component inherits from. An example of an implementation of a primitive component is on Fig. 4. The example uses nested classes to implement the provided interfaces, and demonstrates how both the provided and the required interfaces of a frame are accessed by the implementation of the primitive component.

The frame and architecture classes also inherit from base classes that define methods for generic access to the provided and required interfaces of the frame and the subcomponents and the ties of the architecture. These methods are required by the deployment mechanism.

```
class HelloWorld : public virtual HelloWorldArch {
  public:
    // Implementation of the ApplicationIface interface
    class Application : public virtual ApplicationIface {
      public:
        Application (HelloWorld *frame) { me = frame; };
        // Displaying the greeting using the other subcomponents
        virtual Short run (const StringSequence& args) {
          char *message = me->MessageReq()->message ();
          me->DisplayReq()->print (message);
          return 0;
        };
      private:
        HelloWorld *me;
    };
    // Initialization of the HelloWorldArch architecture
    virtual void initialize () {
      HelloWorldArch::initialize ();
      ApplicationProv (new Application (this));
    };
};
```

Fig. 4. A C++ implementation of HelloWorldArch.

3.2 Deploying Application Components

The deployment is configured by a deployment descriptor. For each frame, the deployment descriptor specifies the architecture that the component will use and the host where the component will run. The deployment is controlled from a single place and expects each host to run a simple server that allows remote instantiation of components. The initialization and interconnection methods of the component are then invoked remotely on the component itself.

The control flow of the deployment mechanism follows the hierarchical architecture of the application being deployed. The architecture forms a tree with each node representing a component. Nodes representing composed components are parents of nodes representing their subcomponents. Nodes representing primitive components are leaves. The references to provided and required interfaces are attributes of each node.

At the beginning of the deployment process, the references to provided interfaces are stored in the attributes of nodes representing primitive components. The references are then propagated toward the root of the tree along the bind and delegate ties in one tree traversal pass, and toward the leaves of the tree along the subsume ties in another traversal pass. The process uses the `createBindingsAndDelegates` and `createSubsumes` methods defined by the language mapping of the component architectures.

```
typedef sequence<string> StringSequence;
interface ApplicationIface { short run (in StringSequence args); };
frame ApplicationFrame { provides: ApplicationIface ApplicationProv; };
```

Fig. 5. The application frame.

The deployment expects the application to implement a standardized frame in Fig. 5. After the application is deployed, the `run` method of `Application-Iface` provided by the application is invoked to launch the application.

3.3 Generating Connectors Using CORBA

Connectors are used to interconnect components that run on different network hosts by delivering remote method invocations to the components. As the hosts where components should run are only known at deployment time, the connectors have to be generated and dynamically loaded at deployment time.

Although the SOFA environment does not place any principal restrictions on the middleware used to implement connectors, we have focused on connectors that are generated by off-the-shelf CORBA middleware. The connector generator is flexible enough to support a number of CORBA middleware implementations.

CORBA middleware generates connectors from a CORBA IDL definition of the interfaces that the connector delivers invocations to. An IDL compiler accepts the CORBA IDL definition of an interface as input and generates C++ source code of the stub and skeleton parts of the connector as output. Both parts need to be compiled, the stub part of the connector is then called by the components that require the interface, the skeleton part of the connector then calls the components that provide the interface.

A development environment that includes both an IDL compiler and a C++ compiler is needed to generate a connector. To avoid the need of having this environment available at deployment time, the SOFA environment pregenerates a set of connectors for all interfaces of an application.

For each interface, a CORBA IDL file that contains the definition of the interface and includes the definitions of all types that the interface relies on is generated by the SOFA environment. The file is compiled by the IDL compiler to yield the C++ source code of the stub and skeleton parts of the connector. The SOFA environment also generates C++ source code of the connectors that uses the code generated by the IDL compiler and interfaces it with the components. The C++ source code is compiled into a pregenerated connector. At deployment time, the pregenerated connectors are dynamically linked with the components.

The SOFA environment can be configured at deployment time to use several middleware implementations. All connectors of a single middleware implementation are managed by a single connector manager. The task of the connector manager is to provide access to the listening loop of the middleware and to enable creation of stub and skeleton parts of a connector in a middleware independent manner.

```
class ConnectorManager {
  public:
    virtual ObjectBase_ptr loadStubPart (const char *reference) = 0;
    virtual char *loadSkeletonPart (ObjectBase_ptr servant) = 0;
    virtual void startListening () = 0;
    virtual void stopListening () = 0;
};
```

Fig. 6. The connector manager interface.

The interface of the connector manager is in Fig. 6. The `loadSkeletonPart` method creates the skeleton part of a connector, returning a stringified reference of the target interface. The `loadStubPart` method creates the stub part of a connector, accepting this stringified reference. The `startListening` and `stopListening` methods control the listening loop of the middleware.

4 Experience in Retrospective

4.1 Shareable Language Mapping

The initially most visible feature of the SOFA environment was the CDL to C++ mapping, based on the IDL to C++ mapping of CORBA [16]. The CDL to C++ mapping of the data and interface types, which SOFA CDL shares with CORBA IDL, is almost as complex as the IDL to C++ mapping of these types. In addition to the data and interface types, the CDL to C++ mapping also supports the frame and architecture types. Considering the size of the IDL to C++ language mapping, over 170 pages of specification at this time, the CDL to C++ language mapping is obviously far from trivial.

The complexity of the mapping can introduce extra cost in terms of code size and runtime overhead. In principle, the extra cost of a mapping designed solely for use by the component code does not have to exceed the extra cost introduced by other libraries that provide useful types in the C++ environment, such as STL [11]. The problem particular to the CDL to C++ mapping, and a language mapping used by any other component system that aspires to employ off-the-shelf middleware to build connectors, is that the mapping is used both by the component system and by the middleware. A typical situation in this case is that the mapping used by the component system is not compatible with the mapping used by the middleware, prompting the need for deep copying at best, and deep copying and data conversion at worst, of all data passed through the middleware. Given the performance of contemporary middleware implementations [25], the copying and conversion might be acceptable in an explicitly distributed application that employs the middleware in a few carefully selected points, but not in a transparently distributed application that relies on the middleware for interconnecting its components at fine granularity levels, where components correspond to implementation objects.

The problem of compatibility of the language mappings used by the component system and the middleware implementations employed to build connectors can be solved by sharing the mapping among the component system and the middleware implementations. In most cases, this requires extending the language mappings of contemporary middleware implementations.

A language mapping of a contemporary middleware implementation is typically designed to make it possible to write applications that are portable across middleware implementations. The mapping is defined so that the application employing the middleware can easily access the mapped types, but it does not define how the middleware itself accesses the mapped types.

A language mapping that is to be shared among a component system and middleware implementations has to extend the contemporary mappings by defining how the middleware itself accesses the mapped types. Such a language mapping makes it possible not only to write applications that are portable across middleware implementations, but also to write middleware implementations that can share language mappings and thus coexist in a single application without incurring extra cost in terms of code size and runtime overhead. A step in this direction are the ORB portability interfaces in the IDL to Java mapping of CORBA [17].

4.2 Connectors Built Using CORBA

The separation of development and deployment phases of the application lifecycle implies a need to postpone the decision on what connectors to employ from the development time to the deployment time. This goes contrary to the typical usage of off-the-shelf middleware, where the connectors are generated and integrated into the application at the development time.

Although it is theoretically possible to use off-the-shelf middleware to generate connectors at deployment time, such an approach runs into a number of practical difficulties. First, it is unusual to require the development system of the middleware to be available at deployment time. Second, the development system of the middleware is often interactive and thus hard to integrate into the component system.

Alternatively, a set of connectors for all interfaces of an application can be pregenerated at the development time. Only those pregenerated connectors that are actually employed will be used at the deployment time. Our experience demonstrates that while feasible, this approach runs against the typical usage of off-the-shelf middleware, where connectors for multiple interfaces are generated from a single CORBA IDL file.

When connectors for multiple interfaces are generated from a single CORBA IDL file, the middleware produces a monolithic module that contains the marshalling code together with the mapping of all types used by the connectors. When used to generate the connectors for one interface at a time, the middleware produces modules that are largely redundant in mapping of those types that are shared by the connectors. Even though the redundancy can be removed

during function level linking, the time spent generating and compiling redundant code is prohibitive even for relatively small number of types and interfaces.

To avoid the problems of redundancy when employed in a component system, an off-the-shelf middleware should provide features that allow for separated generation of the marshalling code and the mapping of the types used by the connectors. Provided that the mapping of the types could be shared among the component system and the middleware implementations, this would allow for generating the mapping of the types at development time, and generating the marshalling code on demand at deployment time.

4.3 ADL Type System Not Suitable

In retrospect, the most constraining decision with respect to the usability of the component system was basing the SOFA type system on the CORBA type system. The type system of CORBA is tailored to suit the underlying remote procedure call mechanism, which is acceptable because a CORBA application uses IDL interfaces in a few carefully selected points. When carried over to SOFA, the type system becomes much more restrictive because a SOFA application uses CDL interfaces for interconnecting its components at fine granularity levels, where components correspond to implementation objects.

Looking at the differences between the type system of C++, which is normally used in the environment we consider, and the type system of SOFA, we can see that C++ relies heavily on reference and pointer types that may not have a counterpart in the SOFA type system.

Reference and pointer types that are used to pass data by reference, whether merely for sake of efficiency or to allow modification of the data, have a good match in the SOFA types used to pass the same data in one of the in, out or inout directions.

Reference and pointer types that are used to build dynamic data structures do not have a good match in the SOFA types. Even if the dynamic data structure happens to match the SOFA sequence or value types, the sharing semantics applied by C++ will not match the copy semantics applied by SOFA. The sharing semantics of the reference and pointer types is difficult to mimic in a component system that supports transparently distributed applications. When building dynamic data structures, it is therefore better to employ high level tools such as containers and iterators rather than low level tools such as references and pointers.

A component system can provide containers and iterators modeled after STL [11] or another well tested framework. These can be employed to build dynamic data structures without having to associate a specific sharing or copy semantics with the type, which would be difficult to implement when the type is used both by C++ and by SOFA.

Reference and pointer types that are used to denote objects may appear to have a good match in the SOFA object reference type. Instances of both types

give their holder the ability to invoke methods on an object. Implementation of a component system that employs this similarity is possible, although not without difficulties [3]. Reference and pointer types that are used to denote functions represent a similar case.

4.4 Need Anticipated Dynamic Changes

From the architectural point of view, passing references that denote objects has the effect of creating new ties between components. Together with the ability to instantiate components, this provides a mechanism for dynamically changing the architecture of the application. The mechanism is similar to the one normally employed by object oriented applications to introduce dynamic changes by creating and linking objects. This similarity makes it well suited for supporting anticipated dynamic changes of the architecture of component applications. The flexibility and ease of use of the mechanism supersedes that of many contemporary component systems with architecture description languages [5, 14].

The downside of the mechanism is that the new connections and components are not reflected in the architecture description. This makes the architecture description lose its relevancy to the application architecture it is to describe. This problem exists in most component systems that employ architecture description languages, where the architecture description is either static [5, 23], or expressed in a way that does not lend itself to describing anticipated dynamic changes [2, 12].

Anticipated dynamic changes of the application architecture appear to be of fundamental importance, much more so than the unanticipated dynamic changes the software architecture research community focuses on. If the architecture description is to be used in a component system at fine granularity level, it is necessary to extend the architecture description language to support such changes. Following the approach suggested for building dynamic data structures, the dynamic architectures could be described as dynamic collections of components.

4.5 Legacy Components And Connectors

Integrating the component system with CORBA and GNOME gave rise to the need of supporting legacy components, especially the components of GNOME used to build the user interface. Besides running into problems with the type system outlined earlier, we also encountered problems related to legacy distribution mechanisms.

The graphical user environment of GNOME runs on top of the X Window System [20], which relies on its own distribution mechanism. The legacy components of GNOME use X resource identifiers as references. The distribution mechanism of the X Window System should therefore be regarded a middleware and X protocol connectors should be introduced to interconnect X components. This would have the advantage of using the X protocol, which is more efficient than the protocols of general purpose middleware. More work needs to be done to design a mechanism for cooperation between multiple types of middleware.

5 Related Work

Although a number of ADL component systems exists, most share the basic architectural concepts related to components and connectors. The component model of SOFA is no exception, being similar to the component model of Darwin [13]. It also fits well into the ACME framework [9] and the xADL toolkit [7], which provide a basis for sharing and manipulating architectural information.

What distinguishes our work on SOFA from that carried out on other component systems is the close integration of our SOFA implementation with CORBA and GNOME. To our knowledge, few other ADL component systems come close to this level of implementation. The notable exceptions are the C2 [14] and Rapide [12] projects, both exerting effort to support real-life applications in real-life settings. Neither project, however, aims at supporting transparently distributed applications.

With its design and implementation, the SOFA environment is also close to component systems that are not based on formal architecture description, such as Microsoft COM [15] or Sun EJB [24]. Besides the lack of the architecture description itself, these systems differ from the SOFA environment also by omitting the explicit specification of interfaces required by components, which is needed for rigorous assembly of components.

Although also lacking the formal architecture description, more similar to the SOFA environment is the CORBA Component Model [19], which provides a definition of components with explicit specification of provided and required interfaces. We believe that the need for shareable language mapping, identified in this paper, also concerns the CORBA Component Model.

Also related to our work is the development in middleware implementations, especially in the area of reflective middleware. Reflective middleware implementations are generally more modular and thus lend themselves better to integration with a component system. Reflective middleware can also employ the formal architecture description for its configuration [6].

6 Conclusion

We have presented the design and implementation of a runtime environment for the SOFA component system. The implementation is integrated with GNOME and CORBA as representatives of a contemporary component framework and a distributed middleware.

The SOFA environment features a CDL to C++ mapping, a deployment mechanism and a connector generator. The language mapping is easy to use, introduces little overhead per se, and enables component substitution. The deployment mechanism is configurable and supports both single-host and distributed deployment transparent to the application. The connector generator produces connectors independent of the application and can integrate several CORBA middleware implementations.

The SOFA environment meets our goals of supporting real-life applications in real-life settings, vital to discover the limitations of ADL component systems with respect to applications. The paper further highlights our findings in this respect, related to the type system of architecture description languages and middleware implementations, the mapping of the type system into the implementation language, and the support for dynamic changes of the application architecture.

We argue that the type system of the architecture description languages needs to be enriched to support building of dynamic data structures without having to resort to the low level tools such as references and pointers, which do not lend themselves well to transparent distribution.

We point out that the mapping of the type system used by contemporary off-the-shelf middleware needs to be extended to define those features of the mapped types that the implementations of the middleware rely upon. This allows sharing the language mapping among the component system and the middleware implementations used to build the connectors. For efficiency reasons, the middleware should generate the marshalling code and the mapping of the types separately.

We also emphasize that the dynamic changes of the application architecture should be allowed through a mechanism similar to the one normally employed by applications to introduce dynamism, such as creating and linking objects. The architecture description languages should reflect this mechanism and provide support for anticipated dynamic changes.

We believe that our findings are not constrained to the particular design and implementation of the SOFA environment we have described, but can be generalized to cover the broad class of component systems that employ architecture description languages or other forms of formal architecture description.

The implementation of the SOFA environment is available for download at http://nenya.ms.mff.cuni.cz.

Acknowledgments

The authors would like to thank František Plášil, Stanislav Višňovský and Adam Buble for valuable comments, and all the members of the Distributed Systems Research Group at Charles University for their work on the SOFA project.

References

1. Allen R. J.: A Formal Approach to Software Architecture, Doctoral thesis at Carnegie Mellon University, USA, 1997
2. Allen R. J., Douence R., Garlan D.: Specifying and Analyzing Dynamic Software Architectures, Proceedings of FASE 1998, Portugal, 1998
3. Bálek D.: Connectors in Software Architectures, Doctoral thesis at Charles University, Czech Republic, http://nenya.ms.mff.cuni.cz, 2002
4. Bálek D., Plášil F.: Software Connectors and Their Role in Component Deployment, Proceedings of DAIS 2001, Poland, 2001

5. Bellissard L., Ben Atallah S., Boyer F., Riveill M.: Distributed Application Configuration, Proceedings of ICDCS 1996, Hong Kong, 1996
6. Blair G., Blair L., Issarny V., Tůma P., Zarras A.: The Role of Software Architecture in Constraining Adaptation in Component-based Middleware Platforms, Proceedings of Middleware 2000, USA, 2000
7. Dashofy E. M., van der Hoek A., Taylor R. N.: An Infrastructure for the Rapid Development of XML-based Architecture Description Languages, Proceedings of ICSE 2002, USA, 2002
8. Garlan D., Kompanek A.: Reconciling the Needs of Architectural Description with Object-Modeling Notations, Proceedings of UML 2000, United Kingdom, 2000
9. Garlan D., Monroe R., Wile D.: ACME: An Architecture Description Interchange Language, Proceedings of CASCON 1997, Canada, 1997
10. Hnětynka P., Mencl V.: Managing Evolution of Component Specifications using a Federation of Repositories, Technical report 2001/2, Department of Software Engineering, Charles University, Czech Republic, 2001
11. International Organization for Standardization: C++ Programming Language, ISO/IEC standard 14882, 1998
12. Luckham D. C., Kenney J. J., Augustin L. M., Vera J., Bryan D., Mann W.: Specification and Analysis of System Architecture Using Rapide, IEEE Transactions on Software Engineering 21(4), 1995
13. Magee J., Tseng A., Kramer J.: Composing Distributed Objects in CORBA, Proceedings of ISADS 1997, Germany, 1997
14. Medvidovic N., Taylor R. N., Whitehead E. J.: Formal Modeling of Software Architectures at Multiple Levels of Abstraction, Proceedings of CSS 1996, USA, 1996
15. Microsoft: Component Object Model Specification 0.9, http://www.microsoft.com, 1995
16. Object Management Group: C++ Language Mapping Specification, formal/99-07-41, ftp://ftp.omg.org/pub/docs/formal/99-07-41.pdf, 1999
17. Object Management Group: Java Language Mapping Specification, formal/99-07-53, ftp://ftp.omg.org/pub/docs/formal/99-07-53.pdf, 1999
18. Object Management Group: Common Object Request Broker: Architecture and Specification, CORBA 2.6.1, formal/02-05-08, ftp://ftp.omg.org/pub/docs/formal/02-05-08.pdf, 2002
19. Object Management Group: CORBA Component Model Specification, ptc/01-11-03, ftp://ftp.omg.org/pub/docs/ptc/01-11-03.pdf, 2001
20. Open Group: X Windows System, http://www.x.org, 2002
21. Plášil F., Bálek D., Janeček R.: SOFA/DCUP: Architecture for Component Trading and Dynamic Updating, Proceedings of ICCDS 1998, USA, 1998
22. Plášil F., Višňovský S.: Behavior Protocols for Software Components, IEEE Transactions on Software Engineering 28(9), 2002
23. Shaw M., DeLine R., Klein D. V., Ross T. L., Young D. M., Zelesnik G.: Abstractions for Software Architecture and Tools to Support Them, IEEE Transactions on Software Engineering 21(4), 1995
24. Sun Microsystems: Enterprise JavaBeans Specification 2.0, http://www.microsoft.com, 2002
25. Tůma P., Buble A.: Open CORBA Benchmarking, Proceedings of SPECTS 2001, USA, 2001.
26. GNOME Documentation Project, http://developer.gnome.org/projects/gdp, 2002

Chapter 12

Contribution

summary of contribution

The first contribution of the thesis is a regression benchmarking methodology, which allows an automated and statistically sound detection of performance changes, and is robust with respect to non-determinism in current computer systems. The methodology makes it possible to monitor and fix performance in software development cycle.

The second contribution is the implementation of a fully automated regression benchmarking suite for large open-source project Mono [70], the Mono Regression Benchmarking Project [6]. The suite is a proof of concept of the new regression benchmarking methodology and of the regression benchmarking concept in general.

The third contribution of the thesis is a high-level design of a generic environment for fully automated benchmarking that supports regression benchmarking. The environment makes incorporation of regression benchmarking into software projects easier and the benchmarking experiments more transparent.

12.1 Regression Benchmarking Methodology

non-determinism in performance

The most important results directly related to regression benchmarking methodology are the identification and solution of problems caused by non-determinism in software compilation and execution. The impact of the non-determinism on performance is quantified and a general statistical model, which allows precise benchmarking in presence of non-determinism, is designed and evaluated. Both the problem and its solution are applicable not only to regression benchmarking, but also to benchmarking in general.

Based on the statistical model and on experience with Mono and CORBA benchmarking projects, a regression benchmarking methodology is formu-

lated. The methodology describes all steps needed to start and perform regression benchmarking of a given software system using a given benchmark. The description of the methodology distilled from [9, 1, 2, 3], follows.

Quantification of Non-determinism in Performance

The non-determinism in compilation and execution could, on an abstract level, be quantified and modeled by random effects or ANOVA (analysis of variance) models [13]. These models, however, typically depend on normality assumptions, which are not, in our experience, met by software performance data. [1] We have therefore introduced a new simple metric, an *impact factor*, which is robust to deviations from normality and is easy to comprehend. The metric is based on a statistical simulation (bootstrap).

impact factor of random effects

We define the impact factor both for *random effects in execution* and *random effects in compilation*. The impact factor is a ratio of standard deviations, estimated by bootstrap. The impact factor of 1 means that there is no impact of the random effects (non-determinism); a larger value of the impact factor signifies an impact. A precise definition and experimental results for different platforms and benchmarks are in [8, 9], included in Sections 4 and 7.

The main contribution of the quantification metric and the experimental results is the evidence that the non-determinism is present in current systems and has a significant effect on benchmark results. Consequently, not only a benchmark, but a whole benchmark experiment consisting of compilation, execution, and measurement, has to be considered for precise benchmarking.

Statistical Model of Benchmark with Non-determinism

We present a new statistical model of benchmark performance that allows precise performance measurements in face of non-determinism. In this model, a software system's performance is considered random. The objective of benchmarking is to estimate the (unknown) mean performance, and, as a measure of reliability of the result, to assess the precision of the estimate.

distribution of average performance

The statistical model uses a benchmark experiment that repeats all its steps: the compilation (l times), the execution (m times for each compilation), and the measured operation (n times for each execution). The result of a benchmark experiment is the average $\overline{Y}_{\bullet\bullet\bullet}$ of all measurements Y_{kji} (k indexes compilations, j executions and i operations):

[1] The related statistical methods are listed in Section 13.

$$\overline{Y}_{\bullet\bullet\bullet} \stackrel{def}{=} \frac{1}{lmn}\sum_{k=1}^{l}\sum_{j=1}^{m}\sum_{i=1}^{n} Y_{kji}.$$

Under very generic assumptions, it is possible to derive an asymptotic distribution of the average, and thus to estimate the mean performance and precision of the estimate. The asymptotic distribution of the average is

$$\overline{Y}_{\bullet\bullet\bullet} \approx N\left(\mu_Y, \frac{\sigma_E^2}{lmn} + \frac{\sigma_B^2}{lm} + \frac{\sigma_V^2}{l}\right),$$

where μ_Y is the mean performance of interest, and σ_E^2, σ_B^2, and σ_V^2 are unknown variances, which can be estimated by S_E^2, S_B^2 and S_V^2:

$$\begin{aligned}
S_E^2 &\stackrel{def}{=} \frac{1}{lm(n-1)}\sum_{k=1}^{l}\sum_{j=1}^{m}\sum_{i=1}^{n}\left(Y_{kji} - \overline{Y}_{kj\bullet}\right)^2,\\
S_B^2 &\stackrel{def}{=} \frac{1}{l(m-1)}\sum_{k=1}^{l}\sum_{j=1}^{m}\left(\overline{Y}_{kj\bullet} - \overline{Y}_{k\bullet\bullet}\right)^2,\\
S_V^2 &\stackrel{def}{=} \frac{1}{l-1}\sum_{k=1}^{l}\left(\overline{Y}_{k\bullet\bullet} - \overline{Y}_{\bullet\bullet\bullet}\right)^2.
\end{aligned}$$

benchmark result and precision

Knowing the asymptotic distribution, we can construct an asymptotic $(1-\alpha)$ confidence interval for the mean performance μ_Y,

$$\overline{Y}_{\bullet\bullet\bullet} \pm u_{1-\frac{\alpha}{2}}\sqrt{\frac{S_E^2}{lmn} + \frac{S_B^2}{lm} + \frac{S_V^2}{l}},$$

where $u_\bullet$ is the quantile function of the standard normal distribution. The probability that the mean performance μ_Y lies within this interval is asymptotically $(1-\alpha)$. The center of the interval $\overline{Y}_{\bullet\bullet\bullet}$ is the estimate of mean performance μ_Y (the benchmark result). The precision of the result can be defined as the length of the interval, for a given α.

optimizing experiments

The model also allows determining the optimum numbers of repetitions of each experiment step (l, m and n) that ensure the best result precision within a given time for the experiment. The optimal values of m and n are

$$m_0 = \sqrt{\frac{b}{w}\cdot\frac{S_B^2}{S_V^2}}, \quad n_0 = \sqrt{w\cdot\frac{S_E^2}{S_B^2}}, \tag{12.1}$$

where w and b are the costs of benchmark execution and compilation. The cost is expressed as the number of operations that could be measured in the

time consumed by a single execution (cost w) or a single compilation (cost b). There is no optimum value for the number of compilations l, as increasing the number of compilations always helps to improve the result precision.

The cost of a benchmark execution w covers benchmark initialization and warming-up. The cost of benchmark compilation b covers re-building of the whole tested system and the benchmark application. Both costs have to be estimated experimentally. In our experience, the cost w depends on the platform and the benchmark, and thus does not change significantly between consecutive versions of software. The cost b also depends on code size, which changes during software development cycle. The estimation of these costs in regression benchmarking is proposed later in this section.

The statistical model is described in detail and derived mathematically in [9], included in Section 7. The background of the model and robustness issues are discussed in [1], included in Section 6.

Detection of Changes

We present a two-sample method for detecting changes in performance results, based on our statistical model of performance with non-determinism. By incorporating the model, the method is robust to random fluctuations in performance caused by the non-determinism. We use a two-sample approach to detect a performance change in the first software version in which it appears. [2]

The statistical model alone already allows designing a two-sample test. However, some of the assumptions of the model may not hold on some systems, and the numbers of repetitions may not be large enough for its asymptotic validity. We therefore choose a simpler comparison technique that is more conservative and allows us to easily quantify the detected changes. Additional experimental or heuristic methods can then base their level of trust in the detected changes on this quantification.

We propose a comparison method that detects a change whenever the confidence intervals for mean performance in the compared software versions do not overlap. This method is inspired by the approximate visual test described in [16]. When no change is detected, the quantification is zero. When a change is detected, the quantification is the percentage of the change in averages (the newer version's average against the older version's average). This method is also suitable for visualization of the results, as evidenced by the Mono regression benchmarking, described later in this section.

[2] A different statistical approach, potentially applicable to detecting changes in regression benchmarking, is discussed in Section 13.

The method for change detection is published with the statistical model of benchmark performance in [9, 2, 1], included in Sections 7, 5 and 6.

Application to Regression Benchmarking

benchmark design

The statistical model described above can be easily applied to regression benchmarking with simple benchmarks, as identified in [3]. First, a benchmark must be designed to cover a complete benchmark experiment that consists of a given number of compilations (l), executions (m) of each binary, and measurements (n) in each execution. The benchmark provides raw data for all measurements at the end of each execution.

initial steps

The number of initial measurements w affected by benchmark initializations (warm-up) can be estimated experimentally by several executions of the benchmark experiments set up for very large values of n (hundreds to hundreds of thousands, depending on the benchmark and platform). The initial cost of benchmark compilation b should also be estimated experimentally. The initial numbers of compilations, executions and measurements (l, m and n) have to be set manually, based on the experiments and on the formula (12.1). As a special case, the comparison of compiled binaries can indicate that the compilation is deterministic, thus l can be set to 1. Minimum limits for l, m and n should be set that will ensure reliable estimates of the variance needed by the statistical model. These experiments have to be performed manually before regression benchmarking of a given software system by a given benchmark can be started. Once the initial values are set up, the regression benchmarking is fully automated.

for each version

Each new software version is automatically downloaded, compiled l times creating l binaries, each binary executed m times, collecting $w + n$ measurements in each execution. Based on the results, a confidence interval for mean performance is created. The parameter b can be adjusted to match a possible increase in code size. Parameters m and n can be adjusted using (12.1) to match possible changes in the impact of the non-determinism. However, in order to keep the variance estimates reliable, the values of m and n must not fall under the pre-set limits.

evaluating results

Performance changes can be detected by comparing the confidence intervals as described above. The detected changes can be linked to suspect modifications of source files and presented to the developers. The presentation should include results from different platforms and benchmarks, and should also include the quantification of the changes. This information helps the developers to assess the importance of the changes and pinpoint the modifications causing the changes. In particular, the developers can decide whether

a detected performance regression can be fixed, or whether it is imposed by functionality improvements. The developers can also easily verify the impact of changes targeted at performance.

12.2 Mono Regression Benchmarking Project

As a proof of concept of the regression benchmarking methodology, we have designed, set-up and are maintaining the Mono Regression Benchmarking Project [6]. We chose the Mono [70] project for its size (over 2 million lines of code), active development, and open-source license. The operations of the Mono Regression Benchmarking Project are fully automated, from downloading daily Mono versions, through benchmarking and analysis, to visualization of the results. The results are publicly available [6] and cover daily Mono versions since August 2004 (currently nearly 400 versions).

connection to Mono

The Mono project founder and leader Miguel de Icaza expressed his interest in the Mono Regression Benchmarking Project and informed about it in his blog [73] and in the Mono Project News [74]; his posts were cited by numerous electronic sites. The official Mono project site [75] links to the Mono Regression Benchmarking Project and informs the Mono developers that their regression tests will be run automatically as benchmarks, with performance results on the web the next day. In the first five months of year 2006, there were 40,000 visits [3] to the Mono Regression Benchmarking Project's web. More than 10,000 of those were in April, when Miguel informed about benchmarking using the regression tests.

We are actively updating the project to incorporate new research results. This section gives a brief up-to-date description of the project, more information can be found on the project web itself and in [1, 2, 9], included in Sections 6, 5, and 7.

Benchmarks and Platforms

The project uses a diverse set of benchmarks targeted at different aspects of the tested system. Each benchmark is run both with the default JIT (Just-In-Time compiler) optimizations turned on, and with all JIT optimizations turned on.

[3] Number of "visits" reported by the AWSTATS tool.

Mono benchmarks

The HTTP Ping benchmark is a client/server benchmark that measures the response time of a remote procedure call that uses the HTTP (SOAP) protocol. The benchmark consists of a client process and a server process both run on a single machine. The TCP Ping benchmark is the same except that it uses a binary TCP protocol for remote communication. The FFT benchmark measures the computation time of a Fast Fourier Transform, based on the SciMark2 benchmark [69, 68]. The benchmark is included in two versions that differ in the memory allocation strategy, and thus differ in the impact of non-determinism on performance during benchmark execution. The Rijndael benchmark measures encryption and decryption time using the Rijndael algorithm, implemented in the Mono cryptographic libraries. All the benchmarks are run on Pentium 4/Linux and on Itanium/Linux platforms.

regression tests

To increase the coverage of the Mono project by the benchmarks and to make benchmark implementation easier, we also run Mono regression tests as benchmarks (currently about 100 tests). One execution of a regression test is interpreted as one invocation of a benchmark operation. The repetitive execution in one address space is achieved using the C# language reflection. New regression tests are automatically incorporated into the operations as soon as they appear in the Mono sources. The regression tests are run on the Pentium 4/Linux platform.

Presentation of the Results

The automatically detected performance changes are presented in the *Changes Summary Chart* and in benchmark-specific *Detailed Changes Charts* [6].

changes summary

The Changes Summary Chart is a table that sums up performance changes in all benchmarks on all platforms. The chart includes seven latest Mono versions only. The main purpose of the chart is to allow developers to quickly check if there were any performance changes, asses the importance (percentage) of the changes, and check which platforms and benchmarks were affected. For each benchmark, platform and version, the chart shows whether there was a regression (positive number of percents), an improvement (negative number of percents) or no significant change. A sample Changes Summary Chart is shown in Figure 12.1. [4] The correlated information in Changes Summary Charts helps the developers to find causes of the detected changes. There is a separate summary chart for the Mono benchmarks that we have created and for the regression tests created by Mono developers.

[4]The sample chart only shows three daily Mono versions. The web version [6] includes seven Mono versions.

	2006-06-18		2006-06-17		2006-06-15	
	P4	IA64	P4	IA64	P4	IA64
HTTP Ping	=	=	=	=	=	=
HTTP Ping (opt.)	=	=	=	=	=	=
TCP Ping	=	=	=	=	=	=
TCP Ping (opt.)	=	=	=	=	=	=
FFT SciMark	=	N/A	=	N/A	=	N/A
FFT SciMark (opt.)	-4.15%	N/A	13.7%	N/A	-1.43%	N/A
Rijndael	=	N/A	=	N/A	=	N/A
Rijndael (opt.)	=	N/A	=	N/A	=	N/A
FFT SciMark (no alloc.)	=	N/A	=	N/A	=	N/A
FFT SciMark (no alloc., opt.)	=	N/A	-9.4%	N/A	=	N/A

Figure 12.1: Changes Summary Chart.

detailed changes

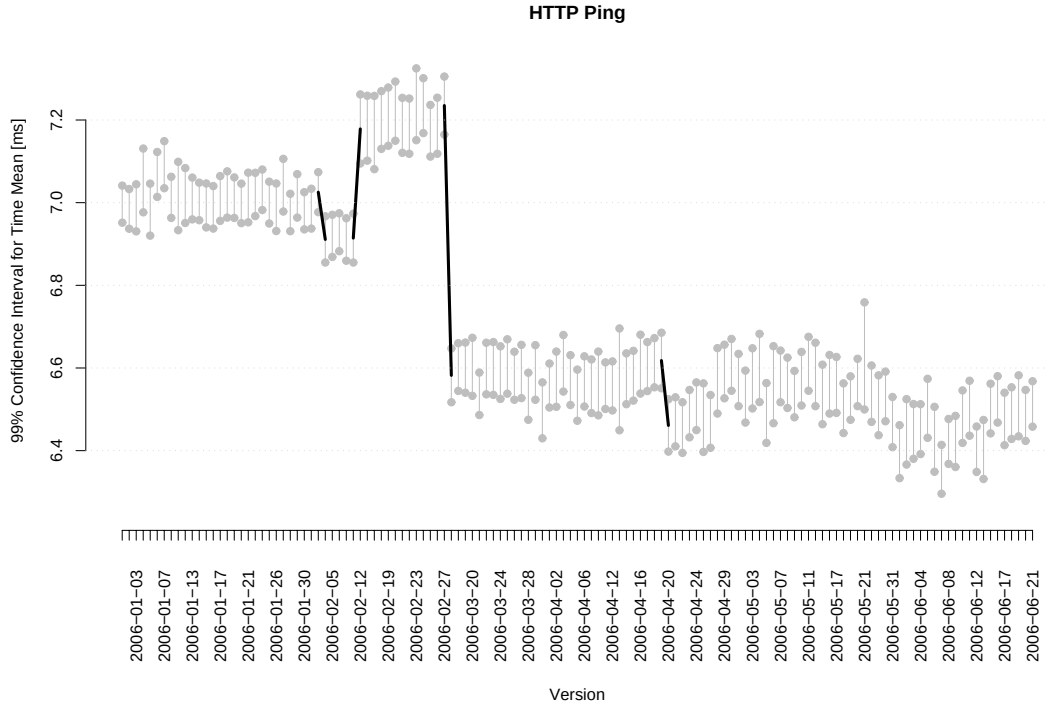
One Detailed Changes Chart is created for each benchmark. It consists of a graph and a table of changes detected by the benchmark. By including the full history of all Mono versions, it helps the developers to identify the long-term performance trends. The graphs also help the developers to visually verify performance trends local to detected performance changes, and thus to asses the importance of the changes. A sample Detailed Changes Chart is shown in Figure 12.2. [5] The form of the Detailed Changes Chart is explained below.

The graph of changes shows the Mono versions on the horizontal axis, and 99% confidence intervals for mean response time on the vertical axis. The detected changes are marked by bold green lines (improvements) and bold red lines (regressions) on the web, and all detected changes are marked by bold black lines in Figure 12.2. The table of changes lists the same changes as those shown in the graph and links them to suspected source code modifications (diffs). The suspected modifications are of a general form (all modifications between respective versions) and of a restricted form (only modifications in Mono libraries that were really used by a specific benchmark). By this linking, the table helps developers in tracking down the causes of the detected performance changes.

run-sequence plots

The results of Mono benchmarks are also presented in traditional *run sequence plots*, which are graphs with Mono versions on the horizontal axis and measured response time on the vertical axis. For each Mono version, a run sequence plot of a given benchmark shows all the measured operation

[5] The sample chart only shows daily Mono versions since January 2006. The chart for all daily Mono versions since August 2004 is available on the web [6].



Newer Version	Older Version	Change
2006-02-04	2006-02-03	-1.63%
2006-02-12	2006-02-08	3.82%
2006-03-17	2006-02-27	-9.02%
2006-04-20	2006-04-19	-2.37%

Figure 12.2: Detailed Summary Chart for the HTTP Ping benchmark.

response times. The plots are automatically scaled on the vertical axis and up to 5% of "outliers" is filtered out. The scaling and filtering aim at selecting the smallest range that has at least 95% of all the measurements. Thanks to this scaling, the run sequence plots are useful for visual checking of all trends in performance, not only in the mean, but also in variance, modality (number of clusters in the results), etc.

All of the plots are described on the web [6]. Additional plots of research and experimental value are presented as well, targeting different methods for change detection and quantification of the impact of non-determinism on benchmark results. Some of these experimental plots are used and described in [1], included in Section 6.

Verified Performance Changes

We have tracked down and experimentally verified some of the most significant performance changes detected. The objective of the verification was to locate the specific source code modifications that caused the changes, and thus to show that the changes were not false alarms. To verify our results, we have manually run benchmark experiments, where we have compared the performance of extra Mono versions differing only in the suspected modifications. A list of verified performance changes we have found is available in [2], included in Section 5. Our verification of a significant performance patch in TCP performance [73] has been particularly welcome by the Mono developers.

12.3 Regression Benchmarking Environment

Based on our experience with the Mono Regression Benchmarking Project, we have formulated requirements and created a high level design of a generic environment for fully automated benchmarking. Such an environment is important for a wide adoption of regression benchmarking, because the benchmark automation is, in our experience, a surprisingly technically challenging and time-consuming task.

The design of the environment is published in [4, 12], included in Sections 8 and 9. The main features of the environment are: support for repeating different levels of a benchmark experiment (compilations, executions, measurements), a common format of benchmark results, a results repository supporting the common format, support for adaptive planning of benchmark experiments based on statistical evaluation of the results, support for automated regression benchmarking, and support for all steps of a benchmark

experiment: download, compilation, deployment, execution, monitoring, data collection, data storage, data analysis and visualization.

The project is being implemented by master students, under the supervision of the author of the thesis. The implementation is currently in early beta stage, which is described in [12]. Up-to-date information is available on the project web page [7].

Chapter 13

Related Projects and Methods

This section provides a short overview and references to projects and software vendors that are performing, planning or advocating regression benchmarking, to statistical methods that are generally related to modeling software performance with non-determinism or to detecting changes in performance results, and to projects that are aiming at or are related to the automated running of benchmarks.

13.1 Regression Benchmarking Projects

calls for regression benchmarking

Although some ad-hoc regression benchmarking projects and our research on regression benchmarking are older, the importance of regression benchmarking in software development cycle has only been accented by the important players of the software industry in the last one or two years. In his post to the Linux Kernel mailing list from March 2005 [33, 34], Linus Torvalds, the initiator and coordinator of the Linux kernel development, asks for automated daily performance benchmarking of the Linux kernel. Mark Mitchell, the release manager and developer of the GNU Compiler Collection (GCC), calls for regular performance regression testing of GCC in his post to the GCC mailing list in November 2005 [36]. In an official document from April 2004 [37], Apple Computer advises developers to monitor software performance during development, and check for performance changes. John Clingan, a technical specialist at Sun Microsystems and an author of many of Sun's press releases in electronic media, argues for incorporation of performance benchmarking into regression testing in his blog from March 2005 [38].

GCC

There are several regression benchmarking projects for the GNU C Compiler (**GCC**) [66, 67]. The older projects, run by Suse (now Novell) and Redhat, focus on regression benchmarking of daily GCC versions using the

SPEC 95 [50] and SPEC 2000 [49, 39] CPU benchmark suites. Some of the results are available for versions since 2002. The measured metrics are binary sizes, compilation time and execution time. The results are presented by simple off-line-generated graphs. Results for different GCC versions and hardware platforms are not integrated into a single site.

There is a newer GCC regression benchmarking project [40, 48] at the University of Szeged, which uses open-source benchmarks similar to the SPEC ones and features a very flexible presentation site. The benchmarks originally measured only the binary size, but have been extended to measure also the compilation time and the execution time. The presentation site supports on-line generation of graphs based on user input (GCC versions to compare, metrics, graph type, etc.). The measurement methodology is very simple: each benchmark is run three times and the median is taken a result.

Linux kernel

A regression benchmarking project of the **Linux kernel** [35], sponsored by Intel, monitors performance changes in the kernel between consecutive releases and release candidates. The project uses a diverse set of both open-source and commercial benchmarks on several Intel platforms; the results are presented in graphs and in tables with changes in percents. The methodology of measurements and evaluation is not described. The changes are most likely all changes, with no relation to statistical significance. Although the results have helped to discover some performance regressions, benchmarking daily versions would be more useful for locating source code modifications causing the changes [34].

TAO

The TAO Performance Scoreboard [72] monitors **TAO** (The ACE ORB) performance twice a day, measuring throughput of remote procedure calls using different CORBA configurations and invocation types. The results are presented in simple graphs and as raw data in textual format. The Scoreboard also shows the binary sizes of different TAO libraries in textual format and shows the percentage changes of these sizes.

RTJBench

RTJBench [62] is a benchmarking framework for Java real-time applications that conform to the Real-Time Specification for Java (RTSJ). RTJBench is regularly used for regression benchmarking of Open Virtual Machine (**OVM**) [52]. The framework is an extension of the popular JUnit [53] framework for regression testing. RTJBench allows sequential running of benchmarks (tests) and allows configuring the memory allocation and garbage collection strategies for each benchmark. It also covers the storage, statistical evaluation and visualization of the results. The solution of extending JUnit has the advantage that the infrastructure built for functionality regression testing may be re-used for regression benchmarking.

OVM

The OVM regression benchmarking focuses at the minimum, average and

maximum latencies measured using the SPEC JVM 98 [51] benchmarks and the latency benchmarks designed by the OVM developers. The results are presented in pre-generated graphs, which show the history of OVM latencies on different platforms and compare the latencies of different Java Virtual Machine implementations supporting RTSJ. The design of the benchmarking framework is described in [62]. The sources are not publicly available.

Solaris

Regression benchmarking is used by Sun Microsystems for verifying that the **Solaris** operating system patches do not cause performance regressions [41]. The benchmarks are described in [41], but the results are not publicly available.

Open Office

Sun is also planning regression benchmarking of **Open Office** [42]. With the APPR [43] toolset, performance of operations such as opening a given document, viewing a given presentation or starting the Office application can be measured without user intervention. Currently, performance of different builds (versions) of Open Office is compared, but the benchmarking is not performed daily. The tool is presently used more for profiling than for automated regression benchmarking. There are no publicly available results that would compare the builds.

ZOPE

Regression benchmarking is planned in the scope of the **ZOPE** project [44]. ZOPE is a large open-source web application server written in Python. The Python runtime environment supports performance measurements using platform-independent metrics ("pystones"). The planned regression benchmarking is aimed at detecting poor performance in absolute numbers, expressed in pystones. The performance limits would be set per language function.

Valgrind

Valgrind is a set of tools for debugging and profiling Linux applications. The project already uses regular regression testing, and the incorporation of benchmarks and nightly performance regression benchmarking is planned [45].

Globus

The **Globus** toolkit is a large software project for GRID applications. Within the project, performance testing is considered for OGSA (Open Grid Services Architecture) [46]. Results from some performance tests are already available on the web [46] in raw numbers.

The existing projects related to regression benchmarking indicate a growing popularity of the methodology. However, none of the existing projects performs an automated detection of performance changes. In addition, the projects do not handle non-determinism in performance, and thus changes visually present in the reported graphs may be caused by random fluctuations due to the non-determinism.

13.2 Statistical Methods

During our research, we have encountered several problems in the experimental performance evaluation that are similar to problems already known and solved by the methods of mathematical statistics. These problems include detection of benchmark steady state, adaptive stopping of measurements based on variance, filtering of outliers, clustering of benchmark results, selecting benchmark configurations to test, detecting performance changes during application execution. Describing the statistical methods related to all of these problems is beyond the scope of the thesis.

In this section, we therefore give references to the statistical methods that are related to the main problems solved in the thesis: modeling non-determinism in performance and detecting performance changes. We describe the statistical methods informally, in the context of benchmarking and regression benchmarking, and give references to formal and detailed descriptions of the methods.

Modeling Non-determinism

The objective of modeling the non-determinism is two-fold. First, we need to verify that the non-determinism has an impact on performance in a given benchmark. If the impact exists, we want to estimate the variance in performance based on the variance estimates at different levels of the benchmark experiment: at the level of operations, executions, and binaries. This problem is in general solved by the analysis of variance (ANOVA). In the words of [13], the model we need is a random effects model with nested effects (at binary level, at execution level, at operations level).

random effects model

The random effects model in one way classification [13] can be used for modeling two levels of non-determinism (e.g. non-determinism in executions and in operations), but with an assumption of normal distribution in operation performance and in mean execution performance. The model is based on conditional probability:

$$\begin{aligned} \mu_i &\sim \textit{i.i.d. } N\left(\mu, \sigma_\mu^2\right), \\ y_{ij}|\mu_i &\sim \textit{indep. } N\left(\mu_i, \sigma^2\right). \end{aligned}$$

In the model, μ corresponds to binary (version) mean performance – the parameter of interest, μ_i is the "random" mean performance in execution i, and $y_{ij}|\mu_i$ is the distribution of operation j's performance in execution i. The model assumes that all executions have the same variance σ^2. With m

executions and n operations in each execution, we can estimate the parameter μ by $\overline{y}_{\bullet\bullet}$:

$$\begin{aligned} \overline{y}_{\bullet\bullet} &= \sum_{i=1}^{m}\sum_{j=1}^{n} y_{ij}, \\ E\left(\overline{y}_{\bullet\bullet}\right) &= \mu. \end{aligned}$$

The variance of the estimate is

$$var\left(\overline{y}_{\bullet\bullet}\right) = \frac{\sigma^2 + n\sigma_\mu^2}{mn}.$$

The model allows constructing a precise (not only asymptotic) confidence interval for μ:

$$\overline{y}_{\bullet\bullet} \pm t_{m-1,\frac{\alpha}{2}} \sqrt{\left(\frac{\sum_{i=1}^{n}\left(\overline{y}_{i\bullet} - \overline{y}_{\bullet\bullet}\right)^2}{m(m-1)}\right)},$$

where α is the confidence level and $t_{m-1,\frac{\alpha}{2}}$ is the $\frac{\alpha}{2}$ quantile of the t–distribution with $m-1$ degrees of freedom.

normality

Although such an extension is not included in [13], the model could be extended to also support the additional level of non-determinism, namely the non-determinism in compilation. The model, however, relies on the normality assumption. In our experience, benchmark results are often multi-modal, and thus this assumption does not hold. The model we use, as described in [9], included in Section 7, can be viewed as an asymptotic extension of this model that is robust to non-normality, and supports three levels of non-determinism.

References to many scientific papers and books on ANOVA methods can be found in [13].

Detecting Changes in Performance

Because of the random nature of performance due to non-determinism, a statistical method is required for detecting performance changes. Without a statistical method, random fluctuations would be mistaken for performance changes. The methods that are available depend on how we choose to detect the changes: either between two versions, or in a sequence of (consecutive) versions.

two-sample approach

Detecting changes between two versions can be done using two-sample tests. The tests for equal mean or variance, both parametric and robust, are widely known and described in most statistical handbooks (parametric t–tests for means, robust Mann Whitney test for medians). For regression

benchmarking, we use a two-sample detection, because we want a change to be detected in the first version in which it appears. Our solution is described in Section 12.

The detection of changes in a sequence of software versions has the potential of re-using results from older versions and thus saving some benchmarking iterations of each version. The statistical methods related to this problem are known as statistical process control methods. A good overview of these methods is given in [18, 17].

on-line process control

The on-line statistical process control methods [18] aim at detecting that a system has changed. The system is assumed to be *in control* initially, and is monitored regularly. The methods attempt to detect that a system gets *out of control*, as indicated by an abrupt change in the monitored output. There is a trade-off between minimizing the probability of false alarms and minimizing the time (number of monitoring points) needed to detect changes after they happen. These methods seem to be readily applicable to run-time monitoring of system performance, aimed at detection of permanent performance changes. The direct application to regression benchmarking, however, seems problematic, because the methods do not allow balancing their precision by taking multiple measurements at the monitoring points. While not important in the current applications of the methods, such balancing is important in regression benchmarking, where the monitoring points would correspond to software versions.

off-line process control

The off-line statistical process control methods [17] allow testing whether a system has changed in the past, detect the number of changes, and find the times when the changes have happened. In regression benchmarking, these methods would allow re-fining of already-detected changes when results from newer versions become available. The method we focus on as potentially applicable to regression benchmarking is the method of Horvárth [19, 17]. The method is based on a statistical test where the null hypothesis (H) assumes no change against the alternative (A) of one change:

change in mean and/or variance

$$
\begin{aligned}
H &: Y_1, \ldots, Y_n \sim N\left(\mu, \sigma^2\right) \\
A &: \exists m \in \{2, \ldots, n-2\} : \\
&\quad Y_1, \ldots, Y_m \sim N\left(\mu_1, \sigma_1^2\right), \\
&\quad Y_{m+1}, \ldots, Y_n \sim N\left(\mu_2, \sigma_2^2\right), \\
&\quad \left(\mu_1, \sigma_1^2\right) \neq \left(\mu_2, \sigma_2^2\right).
\end{aligned}
$$

The test rejects the null hypothesis at significance level α, when

$$Z_n \cdot a_n - b_n > ln\left(-\frac{2}{ln(1-\alpha)}\right),$$

where

$$\begin{aligned}
Z_n &= \max_{1\le k\le n-1}\{|\widetilde{Z}_k^2|\},\\
\widetilde{Z}_k^2 &= n\cdot ln\left(\frac{1}{n}\sum_{i=1}^{n}\left(Y_i-\overline{Y}_n\right)^2\right) - k\cdot ln\left(\frac{1}{k}\sum_{i=1}^{k}\left(Y_i-\overline{Y}_k\right)^2\right) -\\
&\quad -(n-k)\cdot ln\left(\frac{1}{n-k}\sum_{i=k+1}^{n}\left(Y_i-\frac{1}{n-k}\sum_{j=k+1}^{n}\right)^2\right),\\
a_n &= \sqrt{2lnln(n))},\\
b_n &= 2lnln(n)+lnlnln(n).
\end{aligned}$$

The test is asymptotic for large n (long history of measured versions). The assumption of normality can be reduced to less restrictive assumptions [19]. In regression benchmarking, Y_i would be an average of all measurements in version i, and thus the normality would hold asymptotically by Central Limit Theorem (CLT).

multiple changes

It is known that if there is a change, it is in the version k in which $|\widetilde{Z}_k|$ is maximal. Multiple changes can then be tracked down recursively: when a change is detected, we check also the versions before the change and the versions after the change, separately. The obvious disadvantage of this approach is that changes detected later in the process are in general more likely to be false alarms, because the asymptotic test is used on fewer versions. This approach therefore does not work well when the changes in performance happen frequently. These conclusions correspond with our experiments with the method, which suggest that as it is, the method cannot be used for regression benchmarking.

13.3 Environments for Running of Benchmarks

Any environment for automated benchmarking in a distributed heterogeneous environment requires a robust batch execution system. Some of the benchmarking environments implement their own execution system, some use an

existing one. In both cases, the current state of the art in the batch execution systems is important for designing a benchmarking environment. In this section, we provide an overview of several different batch execution systems, and of existing benchmarking environments.

Batch Execution Environments

NQS

Most of the current batch execution systems originated from the Network Queueing System (**NQS**) [29]. There are many different implementations of NQS, both commercial and open-source. The latest open-source implementation is Generic NQS [30]. NQS is a system for transparent remote execution of *jobs* in the UNIX operating systems. The jobs are implemented as shell scripts and specify their requirements on system *resources* (CPU time, memory, or devices). Jobs are submitted to *queues* (possibly on remote hosts), from which they may be routed to other queues or executed, based on their resource requirements and resource limits of the queues. The output of jobs is redirected to files and automatically transferred to the host from which the jobs are submitted.

NQE (Network Queueing Environment) is a commercial version of NQS by Silicon Graphics. It has additional features, such as events for synchronizing execution of jobs, load balancing and a graphical user interface.

NQS, although still used, has been superseded by other execution systems. The main limitations of NQS are its dependence on UNIX, limited security, lack of distributed scheduling, no support for distributed computing and synchronization.

DQS

The Distributed Queueing System (**DQS**) [32, 31], developed at Florida State University, is an open-source descendant of NQS. Its main new features are distributed scheduling, support for parallel computing, and improved security. The scheduling is implemented within *cells* of computers: each cell has a single *master scheduler*, through which all jobs (shell scripts) are submitted to the cell. The scheduling supports load balancing and execution of jobs on systems that are not used locally from console. The system can run distributed applications implemented using different communication libraries, such as MPI and PVM. DQS is used for executing benchmarks by the NIST Benchmarking Toolset, described later in this section.

DQS is UNIX dependent, does not allow defining dependencies between jobs, and is not supported anymore. It is only available as a package for the Debian Linux operating system [32]. More information on DQS can be in [31], a detailed but currently rather dated comparison study of different batch execution systems.

Condor [27] is an open-source distributed computing system developed at University of Wisconsin-Madison. The main objective of Condor is transparent access to a network of computers provided by volunteers. An important design goal is that the owners keep control over their computers. When a computer with a running job is used from console, Condor can temporarily suspend the job or migrate it to a different computer. From the user's perspective, each computation is controlled by a *problem solver*. A problem solver, being itself a Condor job, solves a given problem by submitting other jobs to Condor. There are two supplied problem solvers, one for parallel scientific computing, and one for execution of inter-dependent jobs. The second problem-solver is very close to what is needed for benchmark experiments, which consist of inter-dependent tasks. The problem solver also supports automatic restarting of failed jobs. Condor

Each job is submitted to Condor through an *agent*. Based on requirements of the job, an agent tries to find out a *resource* (such as a remote computer) for running the job. The agent advertises the job requirements at one or more *match-makers*, where also the resources advertise the requirements on jobs they are willing to serve. If a match-maker finds a match in requirements, the advertising agent exchanges more details with the resource, executes the job on the resource, and mediates access of the running job to the file system of the originating host. The Condor architecture allows both the owners of resources and the users submitting jobs to keep control over their systems. The requirements of jobs and resources are described as *classified advertisements* in a special language, with name-value pairs and a three-valued logic with values "true", "false" and "undefined". This language is both simple and flexible for use on different computing systems. resource advertising

Aside from its architecture that keeps the provided computers in control of their owners, Condor differs from NQS in that it supports distributed computing, execution of jobs based on their requirements, better security, migration, check-pointing, and transparent access to file system of the host a job is submitted from. The submitted jobs can be both UNIX shell scripts and Java applications.

Condor fully supports Linux and partially supports the Windows platforms. It is not very portable because of the transparent access to the file system of the original host and migration, which require modification of the operating system kernel. Condor can interoperate with other execution systems that support the GRAM protocol (Grid Resource Access and Management), defined within the Globus grid project [47]. portability

Condor solves problems not present in automation of benchmarking, such as migration or the transparent access to the host from which a job was Condor summary

submitted. Many of the concepts, however, are important for benchmark automation and might be reused: most notably the language for resource advertising (classified advertisements) and the problem-solver that supports job inter-dependencies and automated restarts of failed jobs.

summary

As comparing all the batch execution systems is out of scope of the thesis, we chose NQS for it is the predecessor of most of the systems, DQS for it is used by the NIST benchmarking toolset and Condor for its unique features and architecture. References to more systems can be found in [31].

Benchmarking Environments

In this section, we provide an overview of generic environments for automated execution of benchmarks and discuss their applicability to regression benchmarking.

NIST toolset

The oldest generic benchmarking environment we have found information on is the **NIST benchmarking toolset** [76, 77]. The toolset is aimed at benchmarking of parallel applications in a cluster, both for analyzing the parallel performance of the applications and for comparing the performance of the clusters. The majority of the toolset is constructed from existing open-source tools. All the design decisions and choices of the used tools are well described in the project's technical reports, available from [77].

The toolset consists of *experiment design and control module*, *analysis and visualization module*, and *data collection module*.

experiment design

The experiment design and control module allows creating *benchmarking experiments*. Each benchmark experiment specifies a benchmark application to use, the number of processors, and, optionally, the compiler options, the communication library, the communication options, the command line parameters, and the custom parameters. Some of the parameters can be specified as ranges of values – the experiment module then runs a benchmark for each combination of the parameter values (full-factorial experiment design).

The module generates UNIX shell scripts for the DQS batch execution system. The scripts include resource requirements, instrumentation code, tasks for collecting and uploading results, and execution of the benchmark application itself. The experiment module automatically checks results already available in *results database*, and allows the user to re-use results from experiments with similar parameters. The experiment design and control module has an interactive graphical user interface.

results analysis

The analysis and visualization module is responsible for off-line statistical analysis and visualization of results. The module uses existing commercial

applications for statistical data evaluation – Matlab or IDL, which directly access the results stored in the results database. Both the statistical evaluation and the visualization use methods specific for parallel computing, which are in detail described in the technical reports available from [77].

results storage

The results are stored in a relational database (MySQL). The individual performance measurements are stored in *time histograms*, originally used by the Paradyn project [28]. A time histogram consists of a fixed number of samples from logarithmically lengthening periods. As a result, the initial "noisy" part of the benchmark is well covered, but the data are available from the whole benchmark execution time. Time histograms are therefore useful for long running benchmarks that have stable results after an initialization period and produce a lot of data. The performance results are annotated with configuration information and experiment description.

NIST toolset summary

The toolset does not handle all the tasks that usually form a benchmark experiment, such as software download and compilation. These tasks have to be re-implemented by each benchmark and their inter-dependencies cannot be handled by the toolset. The results stored in the database use a fixed schema, and thus cannot be flexibly extended with statistics and configuration information not foreseen at schema design time. The dependencies on different existing tools, many of which were not listed in this short overview, necessarily complicate portability and maintenance. In addition, the use of DQS has the disadvantage of restricting to UNIX platforms and Windows NT. The use of the toolset on Windows NT is restricted to remote execution of tasks from a UNIX machine and requires special maintenance effort. The toolset is not publicly available and is a discontinued project.

Clif

Clif [64] is a distributed load injection framework for Java applications, written using the Fractal component model. The main objective of the project is the distributed load injection (load generation) for any system accessible through protocols such as HTTP, DNS, JDBC, TCP/IP, DHCP, SIP (Session Initiation Protocol) and LDAP. The load injection is programmed in a specialized language with support for virtual users, conditional loops and branches, probabilistic branches, and cooperative multi-threading. Programs in this language (*scenarios*) can be created using a graphical user interface. The scenarios are run by *load injectors*, which can be distributed on different hosts. The actual deployment of the load injectors and the deployment of *probes* that measure various kinds of system utilization is defined in a *test plan* and carried out automatically.

The results are currently stored in CSV (Comma Separated Values) text files, but a more robust results storage is planned. The plans for future work also include adding data analysis modules. Clif is open-source; the sources

including the documentation are available from [60]. Clif currently lacks data analysis support, has only a very simple results repository, and does not handle all the steps of a benchmark experiment. The missing steps are download, compilation, deployment, execution, and monitoring of the system under test. The current version only deploys, executes, and monitors the components for load injection and data collection.

EPITA system

Automatic Regression Benchmark System is an environment for regression benchmarking being implemented at the EPITA Research and Development Laboratory [61] in France. The design requirements are based on [76] and our work [4], but the proposed architecture is less ambitious. The environment uses a central database for all data, including benchmark results and metadata. There is a common format of benchmark results (XML), which is not directly specified, but is accessed through pre-defined programming language API. Prior to implementing a benchmark, the benchmark developer has to use the environment to create the benchmark metadata and generate code for printing the benchmark results in the requested format. This approach is argued to be less prone to programmer errors and less fragile to the changes of the common format of results.

Each benchmark can report only a single value on output, but the benchmarks can be nested. This mechanism should allow repeating execution of some parts of each benchmark. The benchmark execution itself has to be implemented fully by the benchmark author. There is no support for execution monitoring or performance measurements. After a benchmark finishes, its results can be uploaded to the central database using tools provided by the environment.

The environment periodically checks the repository for missing results in any benchmark of any project, and automatically executes benchmarks to produce the missing results. The results are accessible through a graphical web user interface. According to [61], the environment is yet to be implemented.

summary

None of the listed project can be readily used for fully automated regression benchmarking in a heterogeneous environment, with support for repeating compilations and benchmark executions to face non-determinism in performance.

Chapter 14

Conclusion, Evaluation and Future Prospects

14.1 Conclusion

Aimed at automated detection of performance regressions during software development cycle, regression benchmarking needs a reliable method of detecting performance changes and a generic environment that would automate all the benchmarking and analysis.

We have shown that software performance is subject to significant random fluctuations due to non-determinism in compilation and benchmark execution. A reliable detection of performance changes cannot be achieved without solving the problem of non-determinism. We have solved the problem by statistical modeling of the non-determinism and have proposed a method of detection of performance changes that uses this model.

We have implemented an automated regression benchmarking environment for a large open-source project Mono, sponsored by Novell. This environment (Mono Regression Benchmarking Project [6]) monitors performance changes in daily Mono versions since August 2004, automatically detects the changes, and links them to modifications in the Mono sources. The Mono regression benchmarking environment is used by the Mono developers for verifying the impact of their patches to the source code.

We have designed a generic environment that would make the automated regression benchmarking easily portable to any project, not only the Mono project. We are supervising the implementation of the environment, carried out by master student project. The implementation is currently in early beta stage that allows execution of a distributed CORBA benchmark and has over 65,000 lines of code.

14.2 Evaluation

All of the achievements mentioned above were published as refereed papers in proceedings from conferences, workshop or journals. The papers are included in this thesis.

The Mono Regression Benchmarking Project is advertised on the official Mono project web, and the Mono developers are using it in their work. There were 40,000 accesses to the Mono Regression Benchmark Project pages from January to May of this year. The benchmarking project is known mostly thanks to the two announcements in electronic media by Miguel de Icaza, the founder and leader of the Mono project [73, 74]. We believe it is a significant validation of this work that although originally designed as a test-bed for the statistical methods, it is directly useful for software development of a concrete software system.

14.3 Future Work

In our research we have discovered many problems related to experimental performance evaluation that might possibly be solved with the help of statistical methods, but a concrete application would require further research. These problems include the detection of the duration of a benchmark warm-up phase, classification and filtering of outliers from performance results, or automated clustering of performance results.

The statistical model of software performance should either be extended, or analytically proven to be robust against departures from homogeneity of variance. The model should as well be extended to cover non-determinism in memory allocation at execution time, because this type of non-determinism introduces dependencies between measurements within a single benchmark execution. Both of these extensions might save some benchmark iterations needed for testing a single software version.

More research needs to be done in locating the modifications in sources that cause the detected performance regressions. The possible directions are static analysis of source code, run-time behavior monitoring, or correlating multiple metrics and results from different benchmarks and platforms.

Author's References

[1] T. Kalibera, L. Bulej, and P. Tuma, "Automated detection of performance regressions: The Mono experience." in *MASCOTS*. IEEE Computer Society, 2005, pp. 183–190.

[2] T. Kalibera, L. Bulej, and P. Tuma, "Quality assurance in performance: Evaluating Mono benchmark results." in *QoSA/SOQUA*, ser. Lecture Notes in Computer Science, R. Reussner, J. Mayer, J. A. Stafford, S. Overhage, S. Becker, and P. J. Schroeder, Eds., vol. 3712. Springer, 2005, pp. 271–288.

[3] L. Bulej, T. Kalibera, and P. Tuma, "Repeated results analysis for middleware regression benchmarking," *Performance Evaluation*, vol. 60, no. 1–4, pp. 345–358, May 2005.

[4] T. Kalibera, L. Bulej, and P. Tuma, "Generic environment for full automation of benchmarking," in *SOQUA/TECOS*, ser. LNI, S. Beydeda, V. Gruhn, J. Mayer, R. Reussner, and F. Schweiggert, Eds., vol. 58. GI, 2004, pp. 125–132.

[5] T. Kalibera, "Intelligent source dependency tool," in *Proceedings of 13th ALADIN workshop on ALADIN applications in very high resolution*. Prague, Czech Republic: Czech Hydrometeorological Institute, 2003, pp. 73–81.

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

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

[8] T. Kalibera, L. Bulej, and P. Tuma, "Benchmark precision and random initial state," in *Proceedings of the 2005 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS 2005)*. San Diego, CA, USA: SCS, July 2005, pp. 853–862.

[9] T. Kalibera and P. Tuma, "Precise regression benchmarking with random effects: Improving Mono benchmark results," in *Formal Methods and Stochastic Models for Performance Evaluation*, ser. Lecture Notes in Computer Science, A. Horvath and M. Telek, Eds., vol. 4054. Springer, June 2006, pp. 63–77.

[10] T. Kalibera and P. Tuma, "Distributed component system based on architecture description: The SOFA experience." in *CoopIS/DOA/ODBASE*, ser. Lecture Notes in Computer Science, R. Meersman and Z. Tari, Eds., vol. 2519. Springer, 2002, pp. 981–994.

[11] T. Kalibera, "SOFA support in C++ environments," http://nenya.ms.mff.cuni.cz/~kalibera/sofacxx, 2002.

[12] T. Kalibera, J. Lehotsky, D. Majda, B. Repcek, M. Tomcanyi, A. Tomecek, P. Tuma, and J. Urban, "Automated benchmarking and analysis tool," accepted as full paper at VALUETOOLS 2006 conference, to be included in the conference proceedings, Jun 2006.

References

[13] C. E. McCulloch and S. R. Searle, *Generalized, Linear and Mixed Models*. New York, NY, USA: Wiley–Interscience, 2001.

[14] R. E. Jeffries, A. Anderson, and C. Hendrickson, *Extreme Programming Installed*. Boston, MA, USA: Addison–Wesley Longman Publishing Co., Inc., 2000.

[15] K. Beck, *Test Driven Development: By Example*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[16] R. Jain, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. New York, NY, USA: Wiley–Interscience, Apr. 1991.

[17] J. Antoch, M. Hušková, and D. Jarušková, "Off-line statistical process control," in *Multivariate Total Quality Control: Foundations and Recent Advances*, ser. Contributions to Statistics, C. Lauro, J. Antoch, V. E. Vinzi, and G. Saporta, Eds. Heidelberg, Germany: Physica-Verlag, 2002, pp. 1–86.

[18] J. Antoch and D. Jarušková, "On-line statistical process control," in *Multivariate Total Quality Control: Foundations and Recent Advances*, ser. Contributions to Statistics, C. Lauro, J. Antoch, V. E. Vinzi, and G. Saporta, Eds. Heidelberg, Germany: Physica-Verlag, 2002, pp. 87–124.

[19] L. Horvath, "The maximum likelihood method for testing changes in the parameters of normal observations," *The Annals of Statistics*, vol. 21, no. 2, pp. 671–680, 1993.

[20] B. Boehm, "Get ready for agile methods, with care," *Computer*, vol. 35, no. 1, pp. 64–69, 2002.

[21] W. W. Royce, "Managing the development of large software systems: concepts and techniques," in *ICSE '87: Proceedings of the 9th international conference on Software Engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1987, pp. 328–338.

[22] B. W. Boehm, "A spiral model of software development and enhancement," *Computer*, vol. 21, no. 5, pp. 61–72, 1988.

[23] L. Layman, L. Williams, and L. Cunningham, "Exploring extreme programming in context: An industrial case study," in *ADC '04: Proceedings of the Agile Development Conference (ADC'04)*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 32–41.

[24] P. Abrahamsson, "Extreme programming: First results from a controlled case study," *euromicro*, vol. 00, p. 259, 2003.

[25] L. Williams, W. Krebs, L. Layman, A. I. Anton, , and P. Abrahamsson, "Toward a framework for evaluating extreme programming," in *Proceeings of the 8th Conference on Empirical Assessment in Software Engineering (EASE04)*, May 2004, pp. 11–20.

[26] M. K. Nakayama, "Two-stage stopping procedures based on standardized time series," *Manage. Sci.*, vol. 40, no. 9, pp. 1189–1206, 1994.

[27] D. Thain, T. Tannenbaum, and M. Livny, "Distributed computing in practice: the condor experience." *Concurrency - Practice and Experience*, vol. 17, no. 2-4, pp. 323–356, 2005.

[28] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall, "The paradyn parallel performance measurement tool," *Computer*, vol. 28, no. 11, pp. 37–46, 1995.

[29] B. A. Kingsbury, "The network queueing system," http://www.gnqs.org/oldgnqs/docs/papers/mnqs_papers/original_cosmic_nqs_paper.htm, 1992, Sterling Software.

[30] S. Herbert, "Generic NQS," http://www.gnqs.org/, 2006.

[31] J. A. Kaplan and M. L. Nelson, "A comparison of queueing, cluster and distributed computing systems," NASA, Tech. Rep., 1994.

[32] D. Diedrich, "Distributed queueing system," http://ftp.debian.org/debian/pool/non-free/d/dqs, GNU Debian/Linux.

[33] I. Marson, "Daily kernel performance testing called for by Torvalds," http://uk.builder.com/programming/unix/0,39026612,39241978,00.htm, Mar 2005, ZDNet UK.

[34] J. Andrews, "Linux: Benchmarking 2.6," http://kerneltrap.org/node/4940/print, Mar 2005.

[35] K. Chen and T. Chen, "Linux kernel performance," http://kernel-perf.sourceforge.net/, 2006.

[36] M. Mitchell, "Performance regression testing ?" http://gcc.gnu.org/ml/gcc/2005-11/msg01306.html, Nov 2005.

[37] Apple Computer, Inc., "Performance overview," http://developer.apple.com/documentation/Performance/Conceptual/PerformanceOverview/PerformanceOverview.pdf, Apr 2005, pp. 12–14.

[38] J. Clingan, "Performance regression testing," http://blogs.sun.com/roller/page/jclingan/20050330, Mar 2005, Sun Microsystems.

[39] Standard Performance Evaluation Corporation, "SPEC CPU 2000," http://www.spec.org/cpu2000/, 2006.

[40] University of Szeged, Dept. of Sw. Engineering, "GCC Code-Size Benchmark Environment (CSiBE)," http://www.inf.u-szeged.hu/csibe/, 2006.

[41] Sun Microsystems, "Overview of Solaris patch system testing and performance regression testing," http://sunsolve.sun.com/pub-cgi/show.pl?target=patches/sys-and-perf-test, 2006.

[42] D. Keskar and M. Leibowitz, "Speeding up openoffice - profiling, tools, approaches," http://ooocon.kiberpipa.org/media/Presentation_profiling_tools_approaches, 2006, OOCon 2005.

[43] D. Keskar and M. Leibowitz, "Automated profiling and performance regression (APPR)," http://wiki.services.openoffice.org/wiki/APPR, 2006.

[44] T. Ziade and S. Richter, "Performance regression tool," http://www.zope.org/Wikis/DevSite/Projects/ComponentArchitecture/PerformanceRegressionTool, 2006.

[45] Valgrind, "Project suggestions: Software infrastructure," http://valgrind.org/help/projects.html, 2006.

[46] Globus Alliance, "Globus toolkit 3.0 performance test page," http://www-unix.globus.org/ogsa/tests/gt3_perf_test.html, 2006.

[47] Globus Alliance, "Globus toolkit," http://www.globus.org/toolkit/, 2006.

[48] A. Beszédes, R. Ferenc, T. Gergely, T. Gyimóthy, G. Lóki, and L. Vidács, "CSiBE Benchmark: One Year Perspective and Plans," in *Proceedings of the 2004 GCC Developers' Summit*, Jun 2004, pp. 7–15.

[49] J. L. Henning, "SPEC CPU2000: Measuring CPU performance in the new millennium," *Computer*, vol. 33, no. 7, pp. 28–35, 2000.

[50] Standard Performance Evaluation Corporation, "SPEC CPU 95 Benchmarks," http://www.spec.org/cpu95/, 2006.

[51] Standard Performance Evaluation Corporation, "SPEC JVM 98 Benchmarks," http://www.spec.org/jvm98/, 2006.

[52] Purdue University, Dept. of C.S., "OVM predictability and performance benchmarking," http://ovmj.org/bench/, 2006.

[53] K. Beck and E. Gamma, "JUnit," http://www.junit.org, 2006.

[54] E. Cecchet and J. Marguerite, "RUBiS: rice university bidding system," http://rubis.objectweb.org, 2006.

[55] D. A. Wheeler, "More than a gigabuck: Estimating GNU/Linux's size," http://www.dwheeler.com/sloc/redhat71-v1/redhat71sloc.html, 2002.

[56] J. Spolsky, "The project Aardvark spec," http://www.joelonsoftware.com/articles/AardvarkSpec.html, 2005.

[57] J. Spolsky, "Daily builds are your friend," http://www.joelonsoftware.com/articles/fog0000000023.html, 2001.

[58] NASA Johnson Space Center, "Parametric cost estimating handbook: The waterfall model," http://www1.jsc.nasa.gov/bu2/PCEHHTML/pceh.htm, 2006.

[59] C. Fishman, "They write the right stuff," http://www.fastcompany.com/online/06/writestuff.html, 1996.

[60] ObjectWeb Consortium, "The CLIF project," http://clif.objectweb.org, 2006.

[61] N. Despres, "Automatic regression benchmark system," http://www.lrde.epita.fr/cgi-bin/twiki/view/Publications/20050608-Seminar-Despres-Transformers-RegressionBenchmark-Report, 2005, LRDE (EPITA Research and Development Laboratory).

[62] 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.

[63] E. Cecchet, J. Marguerite, and W. Zwaenepoel, "Performance and scalability of ejb applications," in *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York, NY, USA: ACM Press, 2002, pp. 246–261.

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

[65] ECMA, *ECMA-335: Common Language Infrastructure (CLI)*. Geneva, Switzerland: ECMA (European Association for Standardizing Information and Communication Systems), Dec. 2002. [Online]. Available: http://www.ecma-international.org/publications/standards/Ecma-335.htm

[66] Free Software Foundation, "The GNU compiler collection," http://gcc.gnu.org, 2006.

[67] Free Software Foundation, "Benchmarking GCC," http://gcc.gnu.org/benchmarks/, 2006.

[68] C. Re and W. Vogels, "SciMark – C#," http://rotor.cs.cornell.edu/SciMark/, 2004.

[69] R. Pozo and B. Miller, "SciMark 2.0 benchmark," http://math.nist.gov/scimark2/, 2005.

[70] Novell, Inc., "The Mono Project," http://www.mono-project.com, 2006.

[71] S.-L. Lo, D. Grisby, D. Riddoch, J. Weatherall, D. Scott, T. Richardson, E. Carroll, D. Evers, , and C. Meerwald, "Free high performance orb," http://omniorb.sourceforge.net, 2006.

[72] DOC Group, "TAO performance scoreboard," http://www.dre.vanderbilt.edu/stats/performance.shtml, 2006.

[73] M. de Icaza, "Mono news: Tracking performance," http://tirania.org/blog//archive/2005/Jan-19.html, Jan 2005.

[74] Novell, Inc., "Mono project news: Tracking performance in Mono," http://www.mono-project.com/news/archive/2006/Apr-05.html, Apr 2006.

[75] Novell, Inc., "Mono project: Performance testing," http://www.mono-project.com/PerformanceTesting, 2006.

[76] M. Courson, A. Mink, G. Marçais, and B. Traverse, "An automated benchmarking toolset." in *HPCN Europe*, ser. Lecture Notes in Computer Science, M. Bubak, H. Afsarmanesh, R. Williams, and L. O. Hertzberger, Eds., vol. 1823. Springer, 2000, pp. 497–506.

[77] National Institute of Standards and Technology, "Automated benchmarking tool," http://www.itl.nist.gov/div895/cmr/cluster/, 2006.

[78] A. Buble, L. Bulej, and P. Tuma, "CORBA benchmarking: A course with hidden obstacles." in *IPDPS*. IEEE Computer Society, 2003, p. 279.

[79] Distributed Systems Research Group, "Middleware benchmarking project," http://nenya.ms.mff.cuni.cz/benchmark, 2006.

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

[81] Distributed Systems Research Group, "Open CORBA benchmarking," http://nenya.ms.mff.cuni.cz/~bench, 2006.

[82] P. Hnetynka and F. Plasil, "Dynamic reconfiguration and access to services in hierarchical component models," in *accepted for Proceedings of the 9th International SIGSOFT Symposium on Component-Based Software Engineering (CBSE 2006)*, 2006.

[83] F. Plasil and S. Visnovsky, "Behavior protocols for software components." *IEEE Trans. Software Eng.*, vol. 28, no. 11, pp. 1056–1076, 2002.

[84] F. Plasil, D. Balek, and R. Janecek, "SOFA/DCUP: Architecture for component trading and dynamic updating," in *CDS '98: Proceedings of the International Conference on Configurable Distributed Systems*. Washington, DC, USA: IEEE Computer Society, 1998, p. 43.