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

Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'05)
1526-7539/05 $20.00 © 2005 IEEE

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

Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'05)
1526-7539/05 $20.00 © 2005 IEEE

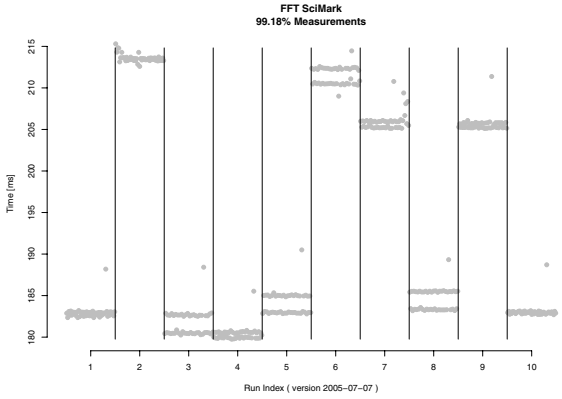


Figure 1. Durations of the same FFT computation in several benchmark runs.

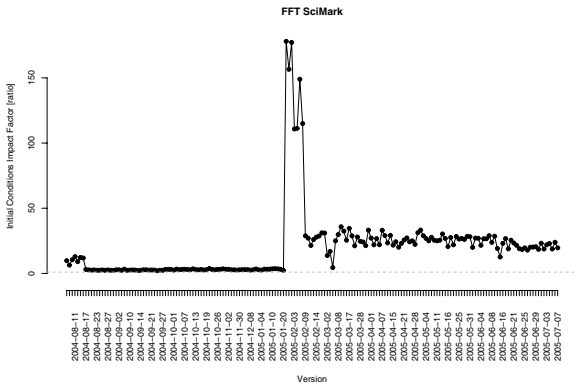


Figure 2. Impact factors of initial conditions for FFT benchmark in different Mono versions.

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-

Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'05)
1526-7539/05 $20.00 © 2005 **IEEE**

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,

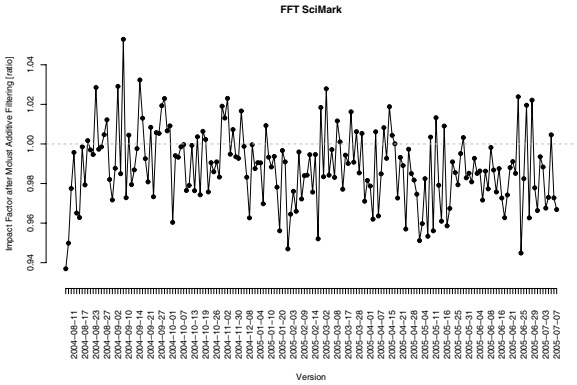


Figure 3. Impact factors of initial conditions after clustered additive filtering in FFT Scimark.

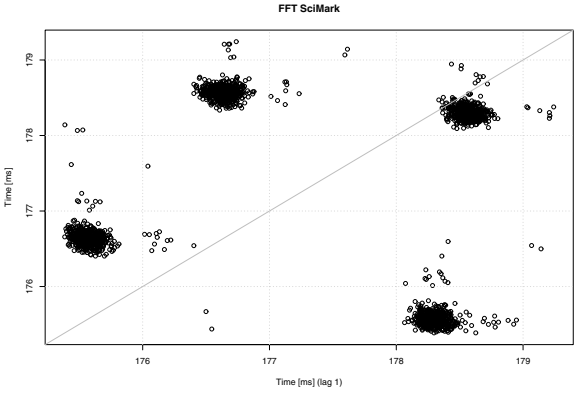


Figure 4. Lag-plot of FFT computation times in a single benchmark run.

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].

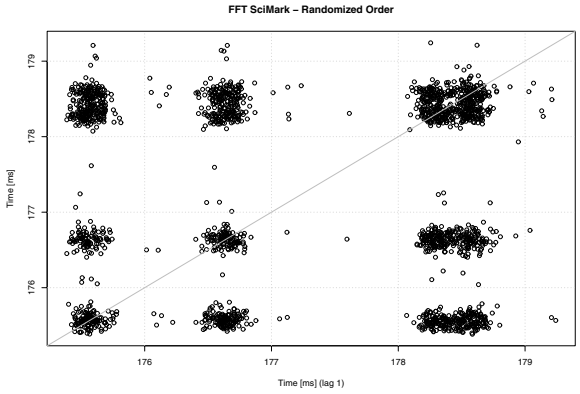


Figure 5. Lag-plot of randomly reordered FFT computation times in a single benchmark run.

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.

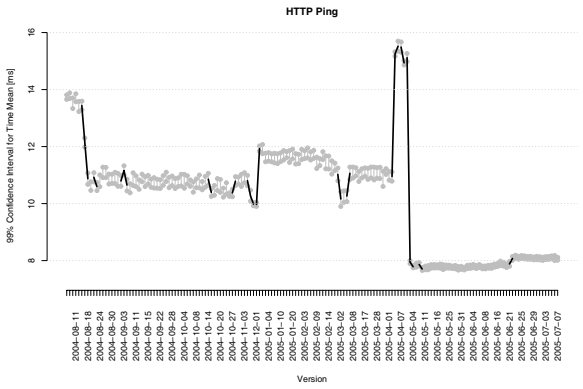
5 Automated Detection of Changes

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.