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 "$Y|\,[M_E = \mu_E]$":

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

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.

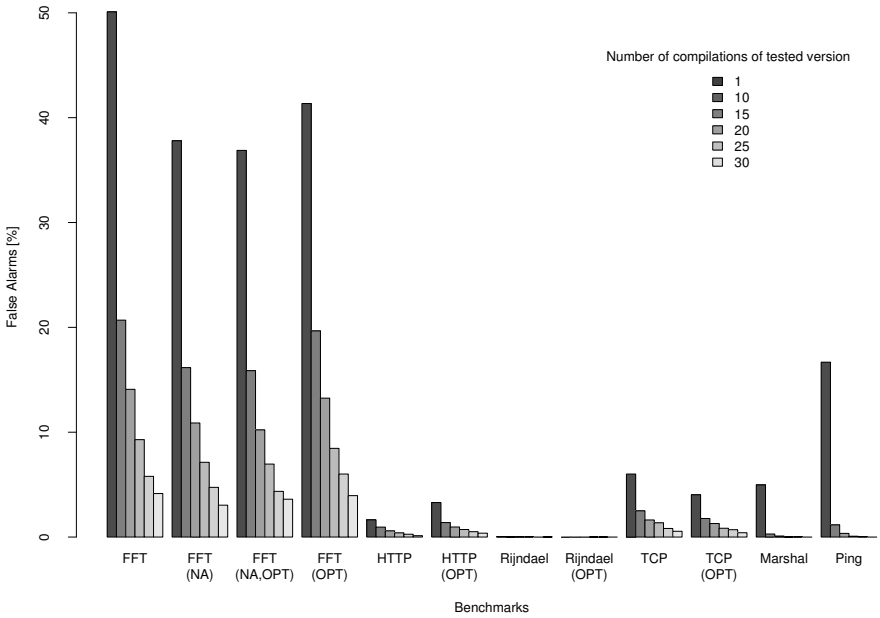
The evaluation is made difficult by the fact that deciding whether a change in observed performance corresponds to a change between software versions requires manual analysis of the software versions in question. Such an analysis becomes prohibitively expensive when enough data for a statistically significant 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].