Department of Distributed and Dependable Systems
Technical report no. D3S-TR-2014-04
December 18, 2014

SPL: Unit Testing Performance

Lubomír Bulej, Tomáš Bureš, Vojtěch Horký, Jaroslav Kotrč,
Lukáš Marek, Tomáš Trojánek and Petr Tůma

Abstract: Unit testing is an attractive quality management tool in the software development process, however, practical obstacles make it difficult to use unit tests for performance testing. We present Stochastic Performance Logic, a formalism for expressing performance requirements, together with interpretations that facilitate performance evaluation in the unit test context. The formalism and the interpretations are evaluated in multiple experiments, to demonstrate (1) the ability to reflect typical developer concerns related to performance, and (2) the ability to identify performance differences in realistic measurements.

This work was partially supported by the EU project ASCENS 257414 and Charles University institutional funding

1 Introduction

Software testing is an established part of the software development process. Depending on the target of the test, three broad testing stages are generally recognized [4]—*unit testing*, where the software components are tested in isolation, *integration testing*, where the focus is on component interactions, and *system testing*, where the entire software system is evaluated. This paper focuses on unit testing.

The limited testing scope makes unit testing flexible—where other testing stages may require specialized personnel and specialized installation to prepare and execute the tests, unit tests are often prepared by the same developers that write the tested units and executed by the same environment that builds the tested units. This is attractive for agile software development methodologies, which emphasize development with short feedback loops.

Typically, unit tests are employed for *functional testing*, where the correctness of the implementation with respect to functional requirements is assessed. Our goal is to employ unit tests for *performance testing*, that is, to assess the temporal behavior of the implementation. Compared to functional unit testing, performance unit testing presents additional challenges:

- The test preparation is sensitive to design errors. A performance unit test is essentially a microbenchmark. Seemingly innocuous mistakes in microbenchmark design can produce misleading results [7, 32].
- The test execution is sensitive to interference. A build environment can compile multiple tested units and execute multiple functional unit tests in parallel—doing the same with performance unit tests would likely produce invalid results due to disruptions from parallel execution.
- The test evaluation is context-dependent and probabilistic rather than context-independent and deterministic. With the possible exception of certain real-time systems, performance testing results very much depend on the execution environment and their evaluation against absolute timing requirements is not practical.

We work on a performance testing environment that addresses these challenges [33]. Central to the environment is Stochastic Performance Logic (SPL), a mathematical formalism for expressing and evaluating performance requirements [5]. The environment permits attaching performance requirement specifications to individual methods to define common performance tests [13].

Broadly, the goal of performance testing is making sure that the system under test executes fast enough. What exactly is fast enough depends on the test scope—system testing evaluates the end-to-end performance, where the performance requirements can be derived from the application function or from the user expectations. Examples of such requirements include the time limit for decoding a frame in a video stream, determined from the frame rate, or the time limit for serving a web page in an interactive application, determined from the studies of user attention span. Importantly, these requirements can be naturally expressed in terms of absolute time limits.

In unit testing, performance requirements expressed as absolute time limits are much less practical. Not only is it difficult to determine how fast an individual method should execute, it is also hard to scale the limits to reflect their platform dependence. SPL formulas therefore express performance requirements by comparing performance of multiple methods against each other, making it easy for the developer to express common performance related assertions—for example that a lookup in a hash map should be faster than a lookup in a list, or that a refactored method should not be slower than the earlier version.

Performance measurements require collecting multiple observations to provide sufficiently representative information. SPL formulas are therefore interpreted using statistical hypothesis testing on the collected observations. Multiple interpretations with different requirements on the measurement procedure are provided, the developer can also tune the test sensitivity by adjusting the test significance level.

When attached to individual methods, SPL formulas can be used for various purposes. These include regression testing, where the developer checks that the performance of a method does not degrade, but also documenting performance provided by the method or documenting performance expected from the environment. Our performance testing environment takes care of executing the measurements, evaluating the performance requirements and reporting the results.

This paper constitutes a comprehensive presentation of our performance testing environment. In Section 2, we introduce the software development context we target. Section 3 presents the basic definitions of the SPL formalism. Section 4 introduces three practical SPL interpretations, each tailored

for a different measurement procedure. The programming side of the performance test construction is described in Section 5. Section 6 provides experimental evaluation. Connections to related work are summarized in Section 7. Finally, Section 8 concludes the paper.

This is a paper that combines and extends previously published conference material. The basic definitions of the SPL formalism and the basic SPL interpretations were initially published in [5] and remain mostly the same here. The SPL interpretations that consider multiple measurement runs are new, except for some initial derivations that rely on [19]. The test construction is significantly extended compared to [5]. The experimental evaluation is partially new and partially from [13].

2 Software Development Context

To provide a comprehensive presentation of our performance testing environment, we present both the formal foundation and the important implementation aspects. Although we strive to keep these two parts of the presentation relatively independent, many elements of the formal foundation are necessarily tied to the intricacies of real world performance evaluation. We have therefore found it useful to first briefly outline the software development context we target, the relevant connections are then introduced gradually in the text.

We aim to support modern programming environments with potentially complex runtime platforms. Most of our work uses Java 7 as the programming language and OpenJDK 1.7 as the virtual machine. We assume the performance measurements are performed in presence of disruptive mechanisms associated with such environments, in our case especially the garbage collection and the just-in-time compilation.

The workloads we measure are benchmarks with granularity broadly similar to unit tests. We expect code with dependencies that can be mocked and can handle multiple measurement iterations. The measurement times must be compatible with the build process where unit tests typically execute, this means benchmarks cannot always run long enough to deliver entirely stable overall performance measurements. We focus on testing steady state performance that the measurement iterations should tend to, rather than transient performance effects that only some iterations exhibit, but we do expect such effects to always be present.

On the terminology side, we say measurements collect one *observation* per iteration, this observation is typically the execution time of the measured code. Observations collected between one start and one exit of a virtual machine are said to belong to the same *run*. A measurement may involve restarting the virtual machine between observations, such measurement consists of multiple runs.

In the experiments presented throughout the paper, we use the following platforms:

- An Intel Xeon E5-2660 machine with 2 sockets, 8 cores per socket, 2 threads per core, running at 2.2 GHz, 32 kB L1, 256 kB L2 and 20 MB L3 caches, 48 GB RAM, running 64 bit Fedora 20 with OpenJDK 1.7, here referred to as *Platform Alpha*.
- An Intel Xeon E5345 machine with 2 sockets, 4 cores per socket, 1 thread per core, running at 2.33 GHz, 32 kB L1 and 4 MB L2 caches, 8 GB RAM, running 64 bit Fedora 18 with OpenJDK 1.7, here referred to as *Platform Bravo*.
- An Intel Pentium 4 machine running at 2.2 GHz, 8 kB L1 and 512 kB L2 caches, 512 MB RAM, running 32 bit Fedora 18 with OpenJDK 1.7, here referred to as *Platform Charlie*.
- An Intel Atom machine running at 1.6 GHz, 24 kB L1 and 512 kB L2 caches, 1 GB RAM, running 32 bit Windows XP Service Pack 2 with Oracle HotSpot 1.7, here referred to as *Platform Delta*.

In some experiments, we use the benchmarks from the ScalaBench 0.1 suite, which incorporates the DaCapo 9.12 suite [30]. For performance unit testing, we use tests of the JDOM library, a software package "for accessing, manipulating, and outputting XML data from Java code" [15]. The tests are motivated by developer concerns inferred from the commit messages in [16] and focus on the SAX builder and the DOM converter as two essential high-level components of JDOM, and on the Verifier class as a low-level component whose performance is critical to JDOM [17]. The choice of JDOM is motivated by both code size and development history—with about 15000 LOC, it is of reasonable size for experiments that require frequent compilation and manual code inspection; with over 1500 commits spread across 14 years of development, it provides ample opportunity to observe performance regressions; it also has an open source license.

3 Stochastic Performance Logic

We formally define the performance of a method as a random variable representing the time it takes to execute the method with random input parameters. The nature of the random input is formally represented by a *workload class* and a *method workload*. The workload is parametrized by *workload parameters*, which capture the dimensions along which the workload can be varied, e.g. array size, matrix sparsity, number of vertices in a graph, etc.

Definition 3.1. Workload class *is a function* $\mathfrak{L} : P^n \to (\Omega \to I)$*, where for a given* $\mathfrak{L}$*,* P *is a set of* workload parameter *values,* n *is the number of parameters,* Ω *is a sample space, and* I *is a set of objects (method input arguments) in a chosen programming language.*

For later definitions we also require that there is a total ordering over P*.*

Definition 3.2. Method workload *is a random variable* $L^{p_1,\ldots,p_n}$ *such that* $L^{p_1,\ldots,p_n} = \mathfrak{L}(p_1, \ldots, p_n)$ *for a given workload class* $\mathfrak{L}$ *and parameters* $p_1, \ldots, p_n$*.*

Unlike conventional random variables, which map the individual observations to real numbers, method workload is a random variable mapping observations to object instances, which serve as random input parameters for the method under test. If necessary, the developer may adjust the underlying stochastic process to obtain random input parameters representing domain-specific workloads, e.g. partially sorted arrays.

To demonstrate the above concepts, let us assume we want to measure the performance of a method S, which sorts an array of integers. The input parameters for the sort method S are characterized by workload class $\mathfrak{L}_S : \mathbb{N}^+ \to (\Omega_S \to I_S)$. Let us assume that the workload class $\mathfrak{L}_S$ represents an array of random integers, with a single parameter determining the size of the array. The method workload returned by the workload class is a random variable whose observations are instances of random arrays of given size. For example, method inputs in the form of random arrays of size 1000 will be obtained from observations of random variable $L_S^{1000} : \Omega_S \to I_S = \mathfrak{L}_S(1000)$.

Note that without loss of generality, we assume in the formalization that there is exactly one $\mathfrak{L}_M$ for a particular method M and that M has just one input argument.

With the formal representation of a workload in place, we now proceed to define the method performance.

Definition 3.3. *Let* $M(in)$ *be a method in a chosen programming language and* $in \in I$ *its input argument. Then method performance* $P_M : P^n \to (\Omega \to \mathbb{R})$ *is a function that for given workload parameters* $p_1, \ldots, p_n$ *returns a random variable whose observations correspond to the execution duration of method* M *with input parameters obtained from the observations of* $L_M^{p_1,\ldots,p_n} = \mathfrak{L}_M(p_1, \ldots, p_n)$*, where* $\mathfrak{L}_M$ *is the workload class for method* M*.*

We can now define the Stochastic Performance Logic (SPL) that will allow us to make comparative statements about method performance under a particular method workload. To facilitate comparison of method performance, SPL is based on regular arithmetics, in particular on axioms of equality and inequality adapted for the method performance domain.

Definition 3.4. *SPL is a many-sorted first-order logic defined as follows:*

- *There is a set* $FunP$ *of function symbols for method performances with arities* $P^n \to (\Omega \to \mathbb{R})$ *for* $n \in \mathbb{N}^+$*.*
- *There is a set* $FunT$ *of function symbols for performance observation transformation functions with arity* $\mathbb{R} \to \mathbb{R}$*.*
- *The logic has equality and inequality relations* $=, \leq$ *for arity* $P \times P$*.*
- *The logic has equality and inequality relations* $\leq_{p(tl,tr)}, =_{p(tl,tr)}$ *with arity* $(\Omega \to \mathbb{R}) \times (\Omega \to \mathbb{R})$*, where* $tl, tr \in FunT$*.*
- *Quantifiers (both universal and existential) are allowed only over finite subsets of* P*.*

- *For $x, y, z \in P$ and $P_M, P_N \in FunP$, the logic has the following axioms:*

$$x \leq x \tag{1}$$
$$(x \leq y \wedge y \leq x) \leftrightarrow x = y \tag{2}$$
$$(x \leq y \wedge y \leq z) \rightarrow x \leq z \tag{3}$$
$$\begin{gathered} For\ each\ pair\ tl, tr \in FunT\ such\ that \\ \forall o \in \mathbb{R} : tl(o) \leq tr(o), there\ is\ an\ axiom \\ P_M(x_1, \ldots, x_m) \leq_{p(tl,tr)} P_M(x_1, \ldots, x_m) \end{gathered} \tag{4}$$
$$\begin{gathered} (P_M(x_1, \ldots, x_m) \leq_{p(tm,tn)} P_N(y_1, \ldots, y_n) \wedge \\ P_N(y_1, \ldots, y_n) \leq_{p(tn,tm)} P_M(x_1, \ldots, x_m)) \leftrightarrow \\ P_M(x_1, \ldots, x_m) =_{p(tm,tn)} P_N(y_1, \ldots, y_n) \end{gathered} \tag{5}$$

Axioms (1)–(3) come from arithmetics, since workload parameters (P) are essentially real or integer numbers. In an analogy to (1)–(2), axiom (4) can be regarded as generalised reflexivity, and axiom (5) shows the correspondence between $=_p$ and $\leq_p$. An analogy of transitivity (3) cannot be introduced for $=_p$ and $\leq_p$, because it does not hold for the sample-based interpretations of SPL as defined in Section 4.

Note that even though we currently do not make use of the axioms in our evaluation, they make the properties of the logic more obvious (in particular the performance relations $=_p$ and $\leq_p$). Specifically, the lack of transitivity for performance relations ensures that SPL formulas can only express statements that are consistent with hypothesis testing approaches used in the SPL interpretations.

Using the logic defined above, we would like to express assumptions about method performance in the spirit of the following examples:

Example 3.1. *"On arrays of 100, 500, 1000, 5000, and 10000 elements, the sorting algorithm A is at most 5% faster and at most 5% slower than sorting algorithm B."*

$$\begin{gathered} \forall n \in \{100, 500, 1000, 5000, 10000\} : \\ P_A(n) \geq_{p(id,\lambda x.0.95x)} P_B(n) \wedge P_A(n) \leq_{p(id,\lambda x.1.05x)} P_B(n) \end{gathered}$$

Example 3.2. *"On buffers of 256, 1024, 4096, 16384, and 65536 bytes, the Rijndael encryption algorithm is at least 10% faster than the Blowfish encryption algorithm and at most 200 times slower than array copy."*

$$\begin{gathered} \forall n \in \{256, 1024, 4096, 16384, 65536\} : \\ P_{Rijndael}(n) \leq_{p(id,\lambda x.0.9x)} P_{Blowfish}(n) \wedge \\ P_{Rijndael}(n) \leq_{p(id,\lambda x.200x)} P_{ArrayCopy}(n) \end{gathered}$$

For compact in-place representation of performance observation transformation functions, we use the lambda notation [1], with id as a shortcut for identity, $id = \lambda x.x$.

To ensure correspondence between SPL formulas in Examples 3.1 and 3.2 and their textual description, we need to define SPL semantics that provides the intended interpretation.

4 Performance Logic Interpretations

A natural way to compare random variables is to compare their expected values. Since method performance is a random variable, it is only natural to base SPL interpretation, and particularly the interpretation of equality and inequality relations, on the expected value of method performance. Other (valid) interpretations are possible, but for simplicity, we first define the *expected-value-based interpretation* and prove its consistency with the SPL axioms.

Each function symbol $f_P \in FunP$ is interpreted as a method performance, i.e. an n-ary function that for input parameter $p_1, \ldots, p_n$ returns a random variable $\Omega \rightarrow \mathbb{R}$, the observation of which corresponds to performance observation as defined in Definition 3.3.

Each function symbol $f_T \in FunT$ is interpreted as a performance observation transformation function, which is a function $\mathbb{R} \rightarrow \mathbb{R}$. In the context of equality and inequality relations between method performances, f_T represents transformation (e.g. scaling) of the observed performance – e.g. statement "*M is 2 times slower than N*" is expressed as $P_M =_{p(id,\lambda x.2x)} P_N$, where $f_{T_1} = id$ and $f_{T_2} = \lambda x.2x$.

The relational operators $\leq$ and $=$ for arity $P \times P$ are interpreted in the classic way, based on total ordering of P.

The interpretation of the relational operators $=_p$ and $\leq_p$ is defined as follows:

Definition 4.1. *Let $tm, tn : \mathbb{R} \to \mathbb{R}$ be performance observation transformation functions, P_M and P_N be method performances, and $x_1, \ldots, x_m, y_1, \ldots, y_n$ be workload parameters. Then the relations $\leq_{p(tm,tn)}$, $=_{p(tm,tn)}$: $(\Omega \to \mathbb{R}) \times (\Omega \to \mathbb{R})$ are interpreted as follows:*

$$P_M(x_1, \ldots, x_m) \leq_{p(tm,tn)} P_N(y_1, \ldots, y_n) \quad \textit{iff}$$
$$E(tm(P_M(x_1, \ldots, x_m))) \leq E(tn(P_N(y_1, \ldots, y_n)));$$

$$P_M(x_1, \ldots, x_m) =_{p(tm,tn)} P_N(y_1, \ldots, y_n) \quad \textit{iff}$$
$$E(tm(P_M(x_1, \ldots, x_m))) = E(tn(P_N(y_1, \ldots, y_n))),$$

where $E(X)$ denotes the expected value of the random variable X, and $tm(X)$ denotes a random variable derived from X by applying function tm on each observation of X.[1]

At this point, it is clear that the expected-value-based interpretation of SPL has the required semantics. However, we have yet to show that this interpretation is consistent with the SPL axioms.

The following lemma and theorem show that the interpretation of $=_p$ and $\leq_p$, as defined by Definition 4.1, is consistent with axioms (4) and (5). The consistency with other axioms trivially results from the assumption of total ordering on P.

Lemma 4.2. *Let $X, Y : \Omega \to \mathbb{R}$ be random variables, and $tl, tr, tx, ty : \mathbb{R} \to \mathbb{R}$ be performance observation transformation functions. Then the following holds:*

$$(\forall o \in \mathbb{R} : tl(o) \leq tr(o)) \to E(tl(X)) \leq E(tr(X))$$

$$(E(tx(X)) \leq E(ty(Y)) \wedge E(ty(Y)) \leq E(tx(X))) \leftrightarrow E(tx(X)) = E(ty(Y))$$

Proof. The validity of the first formula follows from the definition of the expected value. Let $f(x)$ be the probability density function of random variable X. Since $f(x) \geq 0$, it holds that

$$E(tl(X)) = \int_{-\infty}^{\infty} tl(x)f(x)dx \leq \int_{-\infty}^{\infty} tr(x)f(x)dx = E(tr(X))$$

The validity of the second formula follows naturally from the properties of total ordering on real numbers. □

Note that we assumed M to be a continuous random variable. The proof would be the same for a discrete random variable, except with a sum in place of the integral.

Theorem 4.3. *The interpretation of performance relations $\leq_p$ and $=_p$, as given by Definition 4.1, is consistent with axioms (4) and (5).*

Proof. The proof of the theorem naturally follows from Lemma 4.2 by substituting $P_M(x_1, \ldots, x_m)$ for X and $P_N(y_1, \ldots, y_n)$ for Y. □

While the above interpretation illustrates the idea behind SPL, it assumes that the expected value $E(tl(X))$ can be computed. Unfortunately, this assumption hardly ever holds, because the distribution function of X is typically unknown, and so is the expected value. While it is possible to measure durations of method invocations for the purpose of method performance comparison, the type of the distribution and its parameters remain unknown.

[1] Note that the effect of the performance observation transformation function on distribution parameters is potentially complex. We assume that in practical applications, the performance observation transformation functions will be limited to linear shift and scale.

4.1 Sample Based Mean Value Interpretation

To apply SPL when only measurements—that is, observations of the relevant random variables rather than the distribution functions—are available, we turn to sample based methods that work with estimates of distribution parameters derived from the measurements. This leads us to a *sample-based interpretation* of SPL that relies solely on the observations of random variables. Due to limited applicability of the expected-value-based interpretation, in the rest of the paper we will only deal with sample-based interpretations.

The basic idea is to replace the comparison of expected values in the interpretation of $\leq_p$ and $=_p$ by a statistical test. Given a set of observations of method performances (i.e. random variables), the test will allow us to determine whether the mean values of the observed method performances are in a particular relation (i.e. $\leq_p$ or $=_p$).

However, to formulate the sample-based interpretation, we first need to fix the set of observations for which the relations will be interpreted. We therefore define an *experiment*, denoted $\mathcal{E}$, as a finite set of observations of method performances under a particular method workload.

Definition 4.4. *Experiment $\mathcal{E}$ is a collection of $\mathcal{O}_{P_M(p_1,\ldots,p_m)}$, where $\mathcal{O}_{P_M(p_1,\ldots,p_m)} = \{P_M^1(p_1,\ldots,p_m),\ldots, P_M^V(p_1,\ldots,p_m)\}$ is a set of V observations of method performance P_M subjected to workload $L_M^{p_1,\ldots,p_m}$, and where $P_M^i(p_1,\ldots,p_m)$ denotes i-th observation of performance of method M.*

Having established the concept of an experiment, we can now define the sample-based interpretation of SPL. Note that the sample-based interpretation depends on a particular experiment.

The interpretation is the same as before, except for Definition 4.5, used to assign semantics to the method performance relations.

Definition 4.5. *Let $tm, tn : \mathbb{R} \to \mathbb{R}$ be performance observation transformation functions, P_M and P_N be method performances, $x_1,\ldots,x_m, y_1,\ldots,y_n$ be workload parameters, and $\alpha \in \langle 0, 0.5\rangle$ be a fixed significance level.*

For a given experiment $\mathcal{E}$, the relations $\leq_{p(tm,tn)}$ and $=_{p(tm,tn)}$ are interpreted as follows:

- $P_M(x_1,\ldots,x_m) \leq_{p(tm,tn)} P_N(y_1,\ldots,y_n)$ *iff the null hypothesis*

$$\mathrm{H}_0 : E(tm(P_M^i(x_1,\ldots,x_m))) \leq E(tn(P_N^j(y_1,\ldots,y_n)))$$

  cannot be rejected by one-sided Welch's t-test [37] at significance level α based on the observations gathered in the experiment $\mathcal{E}$;

- $P_M(x_1,\ldots,x_m) =_{p(tm,tn)} P_N(y_1,\ldots,y_n)$ *iff the null hypothesis*

$$\mathrm{H}_0 : E(tm(P_M^i(x_1,\ldots,x_m))) = E(tn(P_N^j(y_1,\ldots,y_n)))$$

  cannot be rejected by two-sided Welch's t-test at significance level 2α based on the observations gathered in the experiment $\mathcal{E}$;

where $E(tm(P_M^i(\ldots)))$ and $E(tn(P_N^j(\ldots)))$ denote the mean value of performance observations transformed by function tm or tn, respectively.

Briefly, the Welch's t-test rejects with significance level α the null hypothesis $\overline{X} = \overline{Y}$ against the alternative hypothesis $\overline{X} \neq \overline{Y}$ if

$$\left| \frac{\overline{X} - \overline{Y}}{\sqrt{\frac{S_X^2}{V_X} + \frac{S_Y^2}{V_Y}}} \right| > t_{\nu,1-\alpha/2}$$

and rejects with significance level α the null hypothesis $\overline{X} \leq \overline{Y}$ against the alternative hypothesis $\overline{X} > \overline{Y}$ if

$$\frac{\overline{X} - \overline{Y}}{\sqrt{\frac{S_X^2}{V_X} + \frac{S_Y^2}{V_Y}}} > t_{\nu,1-\alpha}$$

where V_i is the sample size, S_i^2 is the sample variance, $t_{\nu,\alpha}$ is the α-quantile of the Student's distribution with ν levels of freedom, with ν computed as follows:

$$\nu = \frac{\left(\frac{S_X^2}{V_X} + \frac{S_Y^2}{V_Y}\right)^2}{\frac{S_X^4}{V_X^2(V_X-1)} + \frac{S_Y^4}{V_Y^2(V_Y-1)}}$$

As before, we need to show that the sample-based interpretation of SPL is consistent with axioms (4) and (5).

Theorem 4.6. *The interpretation of relations $\leq_p$, and $=_p$, as given by Definition 4.5, is consistent with axiom (4) for a given fixed experiment $\mathcal{E}$.*

Proof. For sake of brevity, we will denote the sample mean of $tl(P^i_M(x_1, \ldots, x_m))$ as $\overline{M}_{tl}$ and the sample variance of the same as S^2_{tl}; $\overline{M}_{tr}$ and S^2_{tr} are defined in a similar way.

Assuming $\forall o \in \mathbb{R} : tl(o) \leq tr(o)$, we have to prove that the null-hypothesis

$$H_0 : E(tl(P^i_M(x_1, \ldots, x_m))) \leq E(tr(P^i_M(x_1, \ldots, x_m)))$$

cannot be rejected by the Welch's t-test.

Based on the formulation of the t-test, it means that the null-hypothesis can be rejected if

$$\frac{\overline{M}_{tl} - \overline{M}_{tr}}{\sqrt{\frac{S^2_{tl}}{V} + \frac{S^2_{tr}}{V}}} > t_{\nu,1-\alpha}$$

where V is the number of samples $P^i_M(x_1, \ldots, x_m)$ in the experiment $\mathcal{E}$.

Since the denominator is a positive number, the whole fraction is non-positive. However, the right hand side $t_{\nu,1-\alpha}$ is a non-negative number since we assumed that $\alpha \leq 0.5$. This means that the inequality never holds and thus the null-hypothesis cannot be rejected. □

Theorem 4.7. *The interpretation of relations $\leq_p$, and $=_p$, as given by Definition 4.5, is consistent with axiom (5) for a given fixed experiment $\mathcal{E}$.*

Proof. For sake of brevity, we will denote the sample mean of $tm(P^i_M(x_1, \ldots, x_m))$ as $\overline{M}$ and the sample variance of the same as S^2_M; $\overline{N}$ and S^2_N are defined in a similar way.

By interpreting axiom (5) according to Definition 4.5, we get the following statements:

$$\begin{aligned}
&P_M(x_1, \ldots, x_m) \leq_{p(tm,tn)} P_N(y_1, \ldots, y_n) \\
&\quad\longleftrightarrow \frac{\overline{M} - \overline{N}}{\sqrt{\frac{S^2_M}{V_M} + \frac{S^2_N}{V_N}}} \leq t_{\nu_{M,N},1-\alpha} \\
&P_N(y_1, \ldots, y_n) \leq_{p(tn,tm)} P_M(x_1, \ldots, x_m) \\
&\quad\longleftrightarrow \frac{\overline{N} - \overline{M}}{\sqrt{\frac{S^2_N}{V_N} + \frac{S^2_M}{V_M}}} \leq t_{\nu_{N,M},1-\alpha} \\
&P_M(x_1, \ldots, x_m) =_{p(tm,tn)} P_N(y_1, \ldots, y_n) \\
&\longleftrightarrow -t_{\nu_{M,N},1-\alpha} \leq \frac{\overline{M} - \overline{N}}{\sqrt{\frac{S^2_M}{V_M} + \frac{S^2_N}{V_N}}} \leq t_{\nu_{M,N},1-\alpha}
\end{aligned}$$

Thus, we need to show that

$$\begin{aligned}
&\frac{\overline{M} - \overline{N}}{\sqrt{\frac{S^2_M}{V_M} + \frac{S^2_N}{V_N}}} \leq t_{\nu_{M,N},1-\alpha} \wedge \frac{\overline{N} - \overline{M}}{\sqrt{\frac{S^2_N}{V_N} + \frac{S^2_M}{V_M}}} \leq t_{\nu_{N,M},1-\alpha} \\
&\quad\longleftrightarrow -t_{\nu_{M,N},1-\alpha} \leq \frac{\overline{M} - \overline{N}}{\sqrt{\frac{S^2_M}{V_M} + \frac{S^2_N}{V_N}}} \leq t_{\nu_{M,N},1-\alpha}
\end{aligned}$$

This holds, because $\nu_{M,N} = \nu_{N,M}$ and thus

$$\frac{\overline{N} - \overline{M}}{\sqrt{\frac{S^2_N}{V_N} + \frac{S^2_M}{V_M}}} \leq t_{\nu_{N,M},1-\alpha} \longleftrightarrow -t_{\nu_{X,Y},1-\alpha} \leq \frac{\overline{M} - \overline{N}}{\sqrt{\frac{S^2_M}{V_M} + \frac{S^2_N}{V_N}}}$$

□

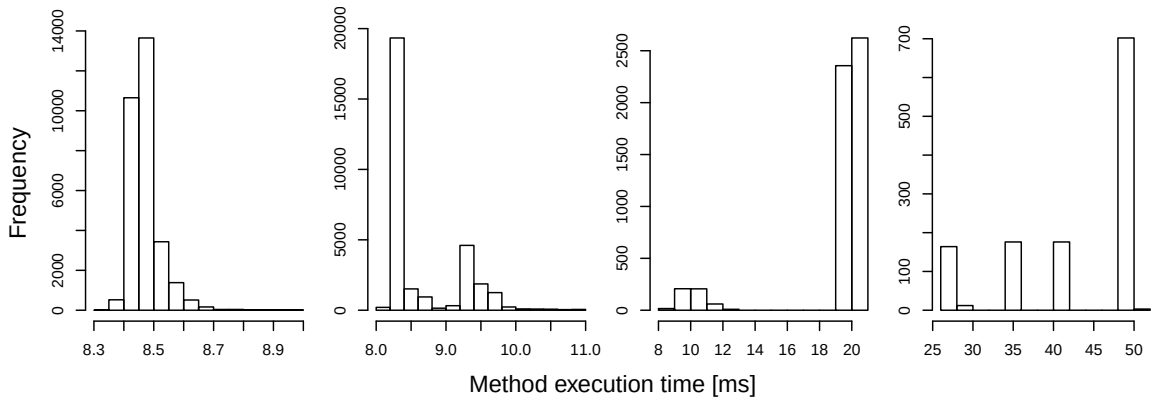

Figure 1: *Execution times of the four example functions.*

Note that, as indicated in Section 3, transitivity (i.e. $(P_X(\ldots) \leq_{p(tx,ty)} P_Y(\ldots) \wedge P_Y(\ldots) \leq_{p(ty,tz)} P_Z(\ldots)) \rightarrow P_X(\ldots) \leq_{p(tx,tz)} P_Z(\ldots)$) does not hold for the sample-based interpretation. This can be shown by considering the following observations and performing single-sided tests at significance level $\alpha = 0.05$: $\mathcal{O}_{P_X} = \{2, 4\}$, $\mathcal{O}_{P_Y} = \{-1, 1\}$, $\mathcal{O}_{P_Z} = \{-4, -2\}$.

Although Welch's t-test formally requires that X and Y are normally distributed, it is reasonably robust to violations of normality due to the Central Limit Theorem. We illustrate this behavior on four methods whose execution time distributions decidedly break the normality assumptions. The execution time histograms of the four methods are given on Figure 1—the distributions are unimodal with a tail, bimodal with small and large coefficient of variation, and quadrimodal.[2] For each of the four methods, we calculate the sensitivity of the performance test to a given change in execution time on a given number of measurements.

Given a set of measurements M of method m, we calculate the sensitivity to a change of scale $s > 1$ on n measurements as follows:

1. We use random sampling to split M in halves M_X and M_Y, $M = M_X \uplus M_Y$.
2. We use random sampling with replacement to create sets of measurements X and Y of size n, $x \in X \Rightarrow x \in M_X$, $y \in Y \Rightarrow y \in M_Y$.
3. We scale one of the sets of measurements by s, $Z = \{y \cdot s : y \in Y\}$.
4. We see whether one sided Welch's t-tests reject the null hypothesis $\bar{X} = \bar{Z}$ in favor of the alternative $\bar{X} > \bar{Z}$ and the alternative $\bar{Z} > \bar{X}$ with significance $\alpha = 0.01$.

The sets X and Z represent hypothetical measurements of m before and after a change of scale s in execution time. We repeat the steps enough times to estimate the probability that the tests correctly favor $\bar{Z} > \bar{X}$ and the probability that the tests incorrectly favor $\bar{X} > \bar{Z}$, which together characterize the test sensitivity.

Figure 2 plots the test sensitivity, expressed as the two probabilities, for $s = 1.01$. The results indicate that for the four methods, mere hundreds of measurements are enough to keep the probability of incorrectly concluding $\bar{X} > \bar{Z}$ close to zero, and tens of thousands of measurements are enough to make the probability of correctly concluding $\bar{Z} > \bar{X}$ reasonably high. To save space, we do not show results for other values of s. These results indicate a test would require an unreasonably high number of measurements to detect changes of 0.1%, while changes of 10% are easily detected even from a small number of measurements.

Welch's t-test also requires that the observations $\mathcal{O}_{P_M(p_1,\ldots,p_m)}$ are sampled independently. This matches the formalization of method performance in Definition 3.3, however, practical measurements are likely to introduce statistical dependence between observations. Two major sources of dependent observations are the initial transient conditions of each measurement run and the changes of conditions between individual measurement runs, both discussed next.

[2]The methods are two versions of SAXBuilder::build, used to build a DOM tree from a byte array stream, Verifier.checkAttributeName, used to check XML attribute name for syntactical correctness, and Verifier.checkCharacterData, used to check XML character data for syntactical correctness, all from the JDOM library, executing on Platform Bravo. The selection is ad hoc, made to illustrate various practical behaviors.

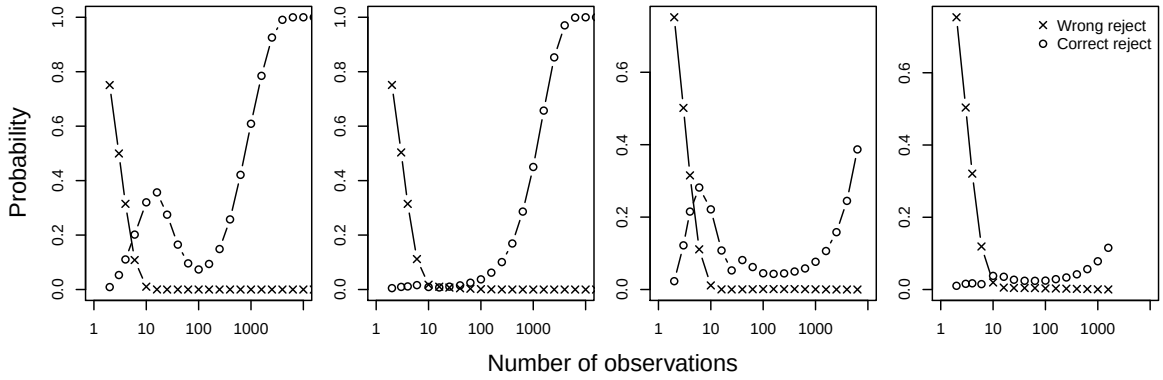

Figure 2: *Sensitivity to 1% execution time change on the four example methods.*

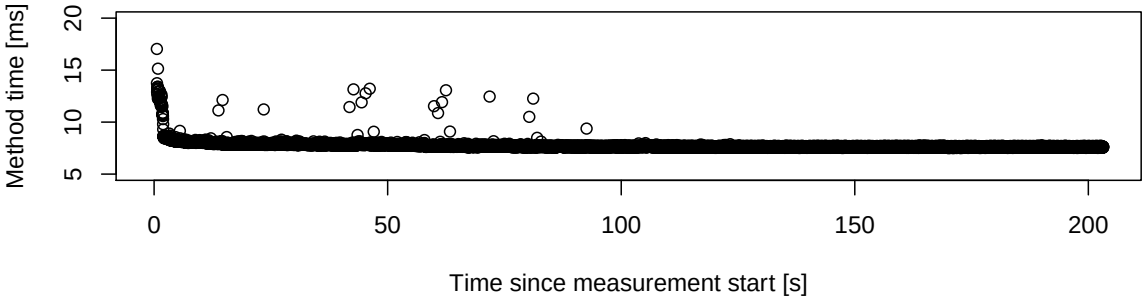

Figure 3: *Example of how just-in-time compilation influences method execution time.*

4.2 Handling Initial Transient Conditions

As an inherent property of our experimental environment, each measurement run is exposed to mechanisms that may introduce transient execution time changes. Measurements performed under these conditions are typically denoted as warmup measurements, in contrast to steady state measurements.

One well known mechanism that introduces warmup is just-in-time compilation. With just-in-time compilation, the method whose execution time is measured is initially executed by an interpreter or compiled into machine code with selected optimizations based on static information. During execution, the same method may be compiled with different optimizations based on dynamic information and therefore exhibit different execution times. This effect is illustrated on Figure 3.[3]

Many other common mechanisms can introduce warmup. To cite three more HotSpot virtual machine examples, the virtual machine defers biased locking for some time after start [9, 8]; the garbage collector automatically adjusts the heap size to reflect the application behavior [26]; the garbage collector needs some time to recognize and promote long lived objects. Until then, the presence of these objects in the young generation can increase both the garbage collection frequency and the cost per garbage collection cycle [21].

Warmup measurements are not necessarily representative of steady state performance and are therefore typically avoided. Often, this can be done by configuring the relevant mechanisms appropriately—a good example is the heap size adjustment, which can be simply disabled.

Unfortunately, the configuration changes that help reduce the initial transient conditions sometimes also impact the steady state performance. We illustrate this on just-in-time compilation, which uses a configurable invocation count threshold to identify the methods to compile. By adjusting this threshold, we can make some methods compile sooner, hoping to reduce the transient effects. Figure 4 shows the impact of this change on the steady state performance of several common benchmarks.[4] Obviously,

[3]The method is SAXBuilder::build, used to build a DOM tree from a byte array stream, from the JDOM library, executing on Platform Alpha. The selection is ad hoc, made to illustrate practical behavior.

[4]The benchmarks are actors, scalac, specs, sunflow, tomcat and xalan, all from the ScalaBench suite, executing on Platform Alpha.

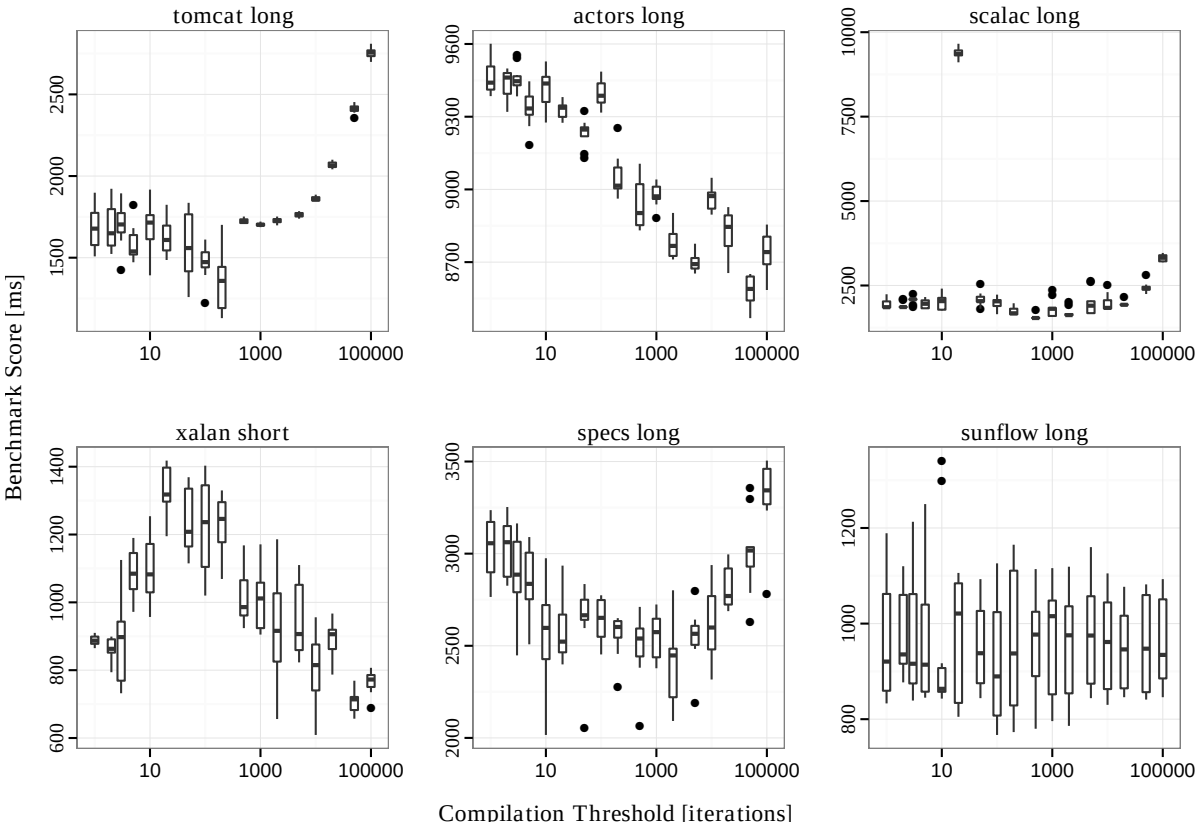

Figure 4: *Effect of compilation threshold on steady state performance.*

the impact is almost arbitrary, suggesting that attempts to avoid the initial transient conditions have practical limits.

Warmup measurements can sometimes be identified by analyzing the collected observations. Intuitively, long sequences of observations with zero slope (such as those on the right side of Figure 3) likely originate from steady state measurements, in contrast to initial sequences of observations with downward slope (such as those on the left side of Figure 3), which likely come from warmup. This intuition is not always reliable, because the warmup measurements may exhibit very long periods of apparent stability between changes. These would look like steady state measurements when analyzing the collected observations. Furthermore, the mechanisms that introduce warmup may not have reasonable bounds on warmup duration. As one example, just-in-time compilation can be associated with events such as change in branch behavior or change in polymorphic type use, which may occur at any time during measurement.

Given these obstacles, we believe that warmup should not be handled at the level of logic interpretation. Instead, knowledge of the relevant mechanisms should be used to identify and discard observations collected during warmup. For the HotSpot virtual machine examples listed above, this would entail disabling the heap size adjustment, consulting the virtual machine configuration to discard observations taken before the biased locking startup delay expires, and using the just-in-time compilation logs or the just-in-time compilation notifications delivered by JVMTI [25] to discard observations taken before most method compilations complete. Afterwards, the logic interpretation can assume the observations do not suffer from transient initial conditions.

In addition to the transient initial conditions, the logic interpretation has to cope with changing conditions between individual measurement runs. In contemporary computer systems, the measurement conditions include factors that stay relatively stable within each measurement run but differ between runs—for example, a large part of the process memory layout on both virtual and physical address level is determined at the beginning of each run. When these factors cannot be reasonably controlled, as is the case with the memory layout example, each measurement run will execute with possibly different

The selection is ad hoc, made to illustrate various practical behaviors.

conditions, which can affect the measurements. The memory layout example is one where a significant impact was observed in multiple experiments [20, 23]. Therefore, no single run is entirely representative of the observable performance.

A common solution to the problem of changing conditions between runs is collecting observations from multiple runs. In practice, each measurement run takes some time before performing steady state measurements, the number of observations per run will therefore be high but the number of runs will be low. In this situation, the sample variance S^2 (when computed from all the observations together) is not a reliable estimate of the population variance σ^2 and the sample-based logic interpretation becomes more prone to false positives, rejecting performance equality even between measurements that differ only due to changing conditions between runs. In [13], we solve the problem by introducing a sensitivity limit. Here, we improve on [13] with two logic interpretations that explicitly consider runs.

4.3 Parametric Mean Value Interpretation

From the statistical perspective, measurements taken within a run have a conditional distribution depending on a particular run. This is typically exhibited as a common bias shared by all measurements within the particular run [19]. Assuming that each run has the same number of observations, the result statistics collected by a benchmark can be modeled as the sample mean of sample means of observations per run (transformed by tm as necessary):

$$\overline{M} = \frac{1}{ro}\sum_{i=1}^{r}\sum_{j=1}^{o} tm(P_M^{i,j}(x_1,\ldots,x_m))$$

where $P_M^{i,j}(x_1,\ldots,x_m)$ denotes the j-th observation in the i-th run, r denotes the number of runs and o denotes the number of observations in a run.

From the Central Limit Theorem, $\overline{M}$ and the sample means of individual runs

$$\overline{M}_i = \frac{1}{o}\sum_{j=1}^{o} tm(P_M^{i,j}(x_1,\ldots,x_m))$$

are asymptotically normal. In particular, a run mean converges to the distribution $N(\mu_i, \sigma_i^2/n)$. Due to the properties of the normal distribution, the overall sample mean then converges to the distribution

$$\overline{M} \sim N\left(\mu, \frac{\rho^2}{r} + \frac{\overline{\sigma^2}}{ro}\right)$$

where $\overline{\sigma^2}$ denotes the average of run variances and ρ^2 denotes the variance of run means [19].

This can be easily turned into a statistical test of equality of two means, used by the interpretation defined below. Note that since the variances are not known, they have to be approximated by sample variances. That makes the test formula only approximate, though sufficiently precise for large r and o [19].

Definition 4.8. *Let $tm, tn : \mathbb{R} \to \mathbb{R}$ be performance observation transformation functions, P_M and P_N be method performances collected over r_M, r_N runs, each run having o_M, o_N observations respectively, $x_1,\ldots,x_m$, $y_1,\ldots,y_n$ be the workload parameters, and $\alpha \in \langle 0, 0.5\rangle$ be a fixed significance level.*

For a given experiment $\mathcal{E}$, the relations $\leq_{p(tm,tn)}$ and $=_{p(tm,tn)}$ are interpreted as follows:

- $P_M(x_1,\ldots,x_m) \leq_{p(tm,tn)} P_N(y_1,\ldots,y_n)$ *iff*

$$\overline{M} - \overline{N} \leq z_{(1-\alpha)}\sqrt{\frac{o_M R_M^2 + \overline{S_M^2}}{r_M o_M} + \frac{o_N R_N^2 + \overline{S_N^2}}{r_N o_N}}$$

where $z_{(1-\alpha)}$ is the $1-\alpha$ quantile of the normal distribution,

$$\overline{S_M^2} = \frac{1}{r_M(o_M-1)}\sum_{i=1}^{r}\sum_{j=1}^{o}\left(tm(P_M^{i,j}(x_1,\ldots,x_m)) - \frac{1}{o}\sum_{k=1}^{o} tm(P_M^{i,k}(x_1,\ldots,x_m))\right)^2$$

$$R_M^2 = \frac{1}{r_M - 1} \sum_{i=1}^{r} \left[\left(\frac{1}{n} \sum_{j=1}^{o} tm(P_M^{i,j}(x_1, \ldots, x_m) \right) - \overline{M} \right]^2$$

and similarly for $\overline{S_N^2}$ *and* R_N^2.

- $P_M(x_1, \ldots, x_m) =_{p(tm,tn)} P_N(y_1, \ldots, y_n)$ *iff*

$$\left|\overline{M} - \overline{N}\right| \leq z_{(1-\alpha)} \sqrt{\frac{o_M R_M^2 + \overline{S_M^2}}{r_M o_M} + \frac{o_N R_N^2 + \overline{S_N^2}}{r_N o_N}}$$

The following theorem shows that this interpretation of SPL is consistent with axioms (4) and (5).

Theorem 4.9. *The interpretation of relations* $\leq_p$, *and* $=_p$, *as given by Definition 4.8, is consistent with axioms (4) and (5) for a given fixed experiment* $\mathcal{E}$.

Proof. The proof is very similar to the proofs of Theorem 4.6 and Theorem 4.7. To prove consistency with axiom (4), it is necessary to show that

$$\overline{M_{tl}} - \overline{M_{tr}} \leq z_{(1-\alpha)} \sqrt{\frac{o_M R_{tl}^2 + \overline{S_{tl}^2}}{r_M o_M} + \frac{o_M R_{tr}^2 + \overline{S_{tr}^2}}{r_M o_M}}$$

where M_{tl} denotes the sample mean of run means of $tl(P_M^{i,j}(x_1, \ldots, x_m))$, $\overline{S_{tl}^2}$ and R_{tl}^2 denote for the same the average of variances of samples per run and the variance of run means, respectively. M_{tr}, $\overline{S_{tr}^2}$, and R_{tr}^2 are similarly defined for $tr(P_M^{i,j}(x_1, \ldots, x_m))$. This inequality is obviously true because $\overline{M_{tl}} \leq \overline{M_{tr}}$ and because the right-hand side of the inequality is non-negative.

The consistency with axiom (5) directly comes from comparing the first part of Definition 4.8 with the second part of the definition. □

4.4 Non Parametric Mean Value Interpretation

The interpretation given by Definition 4.8 requires a certain minimal number of runs to work reliably, as illustrated in Section 6. This is because the distribution of run means

$$\overline{M}_i = o^{-1} \sum_{j=1}^{o} tm(P_M^{i,j}(x_1, \ldots, x_m))$$

is not normal even for relatively large values of o—illustrated on Figure 5.[5] Again, for a small number of runs this typically results in a high number of false positives, we therefore provide an alternative interpretation that uses the distribution of $\overline{M}_i$ directly. It works reliably with any number of runs (including only one run), however, the price for this improvement is that the test statistics has to be learned first (e.g. by observing performance across multiple runs of "similarly behaving" methods).

We assume that all observations $P_M^{i,j}(x_1, \ldots, x_m)$ in a run i are identically and independently distributed with a conditional distribution depending on a hidden random variable C. We denote this distribution as $B_M^{C=c}$, meaning the distribution of observations in a run conditioned by drawing some particular c from the hidden random variable C.

We further define the distributions of the test statistics as follows:

- $B_{\overline{M},r_M,o_M}$ is the distribution function of $(r_M o_M)^{-1} \sum_{i=1}^{r_M} \sum_{j=1}^{o_M} tm(\dot{P}_M^{i,j}(x_1, \ldots, x_m))$, where $\dot{P}_M^{i,j}(x_1, \ldots, x_m)$ denotes a random variable with distribution $B_M^{C=c}$ for c drawn randomly once for each i. In other words, $B_{\overline{M},r_M,o_M}$ denotes a distribution of a mean computed from r_M runs of o_M observations each.

- $B_{\overline{M},r_M,o_M-\overline{N},r_M,o_M}$ is the distribution function of the difference $\tilde{M} - \tilde{N}$, where $\tilde{M}$ is a random variable with distribution $B_{\overline{M},r_M,o_M}$ and $\tilde{N}$ is a random variable with distribution $B_{\overline{N},r_N,o_N}$.

[5] Each run collects $o = 20000$ observations after a warmup of 40000 observations. The method is SAXBuilder::build, used to build a DOM tree from a byte array stream, from the JDOM library, executing on Platform Alpha. The selection is ad hoc, made to illustrate practical behavior.

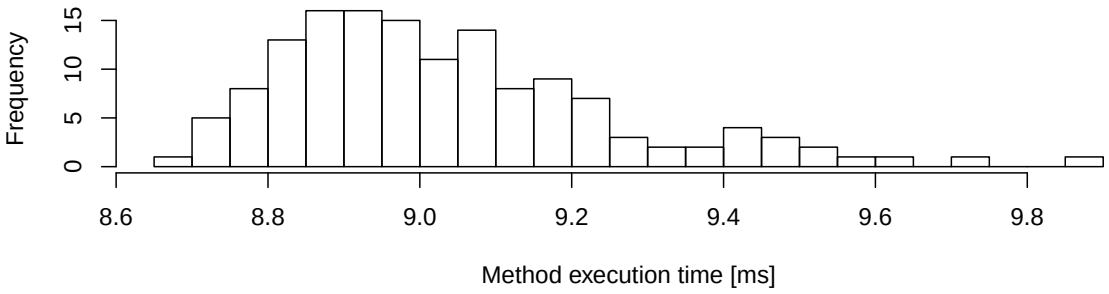

Figure 5: *Example histogram of run means from multiple measurement runs of the same method and workload.*

After adjusting the distributions $B_{\overline{M},r_M,o_M}$ and $B_{\overline{N},r_N,o_N}$ by shifting to have an equal mean, the performance comparison can be defined as:

- $P_M(x_1,\ldots,x_m) \leq_{p(tm,tn)} P_N(y_1,\ldots,y_n)$ iff

$$\overline{M} - \overline{N} \leq B^{-1}_{\overline{M},r_M,o_M-\overline{N},r_N,o_N}(1-\alpha)$$

  where $\overline{M}$ denotes the sample mean of $tm(P_M(x_1,\ldots,x_m))$, $\overline{N}$ is defined similarly, and $B^{-1}_{\overline{M},r_M,o_M-\overline{N},r_N,o_N}$ denotes the inverse of the distribution function $B_{\overline{M},r_M,o_M-\overline{N},r_N,o_N}$ (i.e. for a given quantile, it returns a value).

- $P_M(x_1,\ldots,x_m) =_{p(tm,tn)} P_N(y_1,\ldots,y_n)$ iff

$$B^{-1}_{\overline{M},r_M,M-\overline{N},r_N,o_N}(\alpha) \leq \overline{M} - \overline{N} \leq B^{-1}_{\overline{M},r_M,o_M,-\overline{N},r_N,o_N}(1-\alpha)$$

An important problem is that the distribution functions $B_{\overline{M},r_M,o_M}$, $B_{\overline{N},r_N,o_N}$, and consequently $B_{\overline{M},r_M,o_M-\overline{N},r_N,o_N}$ are unknown. To get over this problem, we approximate the B-distributions in a non-parametric way by bootstrap and Monte-Carlo simulations [31]. This can be done either by using observations $P^{i,j}_M(x_1,\ldots,x_m)$ directly, or by approximating from observations of other methods whose performance behaves similarly between runs.

The basic idea of the bootstrap is that the distribution of a sample mean of an i-th run $\hat{\theta}_{i,o} = o^{-1}\sum_{j=1}^{o}\dot{P}^{i,j}_X(x_1,\ldots,x_m)$ of some method X is estimated by sampling its "bootstrap version" $\theta^*_{i,o} = o^{-1}\sum_{j=1}^{o}P^{*i,j}_X(x_1,\ldots,x_m)$, where samples $P^{*i,j}_X(x_1,\ldots,x_m)$ are randomly drawn with replacement from samples $P^{i,j}_X(x_1,\ldots,x_m)$.

Extending this line of reasoning to the mean of run means $\overline{X}_{r,o} = r^{-1}\sum_{i=1}^{r}\hat{\theta}_{i,o}$, we estimate the distribution of $\overline{X}_{r,o}$, i.e. $B_{\overline{X},r,o}$, by sampling its "bootstrap version" $\overline{X}^*_{r,o} = r^{-1}\sum_{i=1}^{r}\theta^{**}_{i,o}$, where $\theta^{**}_{i,o}$ is randomly drawn with replacement from $\theta^*_{i,o}$. We denote $B^*_{\overline{X},r,o}$ as the distribution of $\overline{X}^*_{r,o}$.

The exact computation of the distribution of the bootstrapped estimator (e.g. the mean of run means) requires traversal through all combinations of samples. This is computationally infeasible, thus Monte-Carlo simulation is typically employed to approximate the exact distribution of the estimator. In essence, the Monte-Carlo simulation randomly generates combinations of samples, evaluates the estimator on them (e.g. $\overline{X}_{r,o}$) and constructs an empirical distribution function $F_n(\cdot) \equiv \frac{1}{n}\sum_{i=1}^{n}\mathbf{1}(X_i \leq \cdot)$ (e.g. of $B_{\overline{X}^*_{r,o}}$ in our case), where $\mathbf{1}(A)$ denotes the indicator function of a statement A.

The whole aparatus of the bootstrap and the Monte-Carlo simulation can then be used to create the bootstrapped distributions $B^*_{\overline{X},r_X,o_X}$, $B^*_{\overline{Y},r_Y,o_Y}$ and their difference $B^*_{\overline{X},r_X,o_X-\overline{Y},r_Y,o_Y}$. To obtain the desired test distribution $B_{\overline{M},r_M,o_M-\overline{N},r_N,o_N}$, we use the approximation $B^*_{\overline{X},r_X,o_X-\overline{Y},r_Y,o_Y}$, where X, Y stand for M, N or other methods whose performance behaves similarly between runs.

Having this theory in place, we define a non-parametric interpretation of the logic as follows:

Definition 4.10. *Let $tm, tn : \mathbb{R} \to \mathbb{R}$ be performance observation transformation functions, P_M and P_N be method performances, $x_1, \ldots, x_m, y_1, \ldots, y_n$ be workload parameters, $\alpha \in \langle 0, 0.5\rangle$ be a fixed significance level, and let X, Y be methods (including M and N) whose performance observations are used to approximate the distributions of P_M and P_N, respectively.*

For a given experiment $\mathcal{E}$, the relations $\leq_{p(tm,tn)}$ and $=_{p(tm,tn)}$ are interpreted as follows:

- $P_M(x_1, \ldots, x_m) \leq_{p(tm,tn)} P_N(y_1, \ldots, y_n)$ *iff*

$$\overline{M} - \overline{N} \leq B^{*-1}_{\overline{X}, r_X, o_X - \overline{Y}, r_Y, o_Y}(1 - \alpha)$$

- $P_M(x_1, \ldots, x_m) =_{p(tm,tn)} P_N(y_1, \ldots, y_n)$ *iff*

$$B^{*-1}_{\overline{X}, r_X, o_X - \overline{Y}, r_Y, o_Y}(\alpha) \leq \overline{M} - \overline{N} \leq B^{*-1}_{\overline{X}, r_X, o_X - \overline{Y}, r_Y, o_Y}(1 - \alpha)$$

When assuming pure bootstrap (without the Monte-Carlo simulations) and placing some restrictions on tl, tr, this interpretation of SPL is again consistent with axioms (4) and (5).

Theorem 4.11. *Assuming that tl and tr are linear functions, the interpretation of relations $\leq_p$, and $=_p$, as given by Definition 4.10, is consistent with axiom (4) for a given fixed experiment $\mathcal{E}$.*

Proof. To prove the consistency with axiom (4), we have to show that it is not true that

$$B_{\overline{X}_{tl}, r_X, o_X - \overline{X}_{tr}, r_X, o_X}(\overline{M}_{tl} - \overline{M}_{tr}) < \alpha$$

We will show that this is not true even for the maximum permitted $alpha$, i.e. 0.5.

$$B_{\overline{X}_{tl}, r_X, o_X - \overline{X}_{tr}, r_X, o_X}(\overline{M}_{tl} - \overline{M}_{tr}) \geq 0.5$$

For sake of brevity, we will denote $\overline{M}_{tl} - \overline{M}_{tr}$ as d. Further, we denote $B'_{\overline{X}_{tl}, r_X, o_X}$ as b_{tl} (the probability density function, the derivative of the cumulative distribution function). We define b_{tr} analogously.

Assuming that tl and tr are linear, we can find a, b, such that $tr(x) = a \cdot tl(x) + b$. The relation between b_{tl} and b_{tr} is then $b_{tr}(x) = b_{tl}\left(\frac{x-b+d}{a}\right)$. The bias d is added because the interpretation works with shifted distributions of equal mean when building the test on equality of means.

Given b_{tl} and b_{tr} and the relation between them, the distribution function

$$B_{\overline{X}_{tl}, r_X, o_X - \overline{X}_{tr}, r_X, o_X}(\overline{M}_{tl} - \overline{M}_{tr})$$

which is the distribution function of $X_{tl} - X_{tr}$, is defined as

$$B_{\overline{X}_{tl}, r_X, o_X - \overline{X}_{tr}, r_X, o_X}(x) = \int_0^\infty b_{tl}(y) \int_{y-x}^\infty b_{tr}(z) dz dy$$

Computing $B_{\overline{X}_{tl}, r_X, o_X - \overline{X}_{tr}, r_X, o_X}$ for d, we get the following equations:

$$B_{\overline{X}_{tl}, r_X, o_X - \overline{X}_{tr}, r_X, o_X}(d) = \int_0^\infty b_{tl}(y) \int_{y-d}^\infty b_{tr}(z) dz dy =$$
$$= \int_0^\infty b_{tl}(y) \int_{y-d}^\infty b_{tl}\left(\frac{z-b+d}{a}\right) dz dy = \int_0^\infty b_{tl}(y) \int_y^\infty b_{tl}\left(\frac{z-b}{a}\right) dz dy$$

Because $tr(x) \geq tl(x)$, it must hold that $b \geq 0$ and $a \geq 1$. Consequently $(z-b)/a \leq z$ and thus

$$\int_0^\infty b_{tl}(y) \int_y^\infty b_{tl}\left(\frac{z-b}{a}\right) dz dy \geq \int_0^\infty b_{tl}(y) \int_y^\infty b_{tl}(z) dz dy$$

The formula $\int_0^\infty b_{tl}(y) \int_y^\infty b_{tl}(z) dz dy$ corresponds to probability of observing a non-positive value in the distribution $B_{\overline{X}_{tl}, r_X, o_X - \overline{X}_{tl}, r_X, o_X}$ of $X_{tl} - X_{tl}$. This distribution is by its nature completely symmetric around 0 and as such $B_{\overline{X}_{tl}, r_X, o_X - \overline{X}_{tl}, r_X, o_X}(0) = 0.5$. Substituting this to the equation completes the proof. □

Note that it would be possible to prove the theorem even without requiring tl, tr to be linear. In that case, we would need another way of expressing tr in terms of tl.

Theorem 4.12. *The interpretation of relations $\leq_p$, and $=_p$, as given by Definition 4.10, is consistent with axiom (5) for a given fixed experiment $\mathcal{E}$.*

Proof. Similar to the proof of consistency with axiom (5) in Theorem 4.9, the consistency here also directly comes from comparing the first part of Definition 4.10 with the second part of the definition. □

5 Programming Language Integration

With unit testing generally accepted as best practice for software projects of almost any size, we assume that developers interested in SPL-based performance unit testing will be familiar with unit testing frameworks, writing unit test cases, and integrating unit testing into the build system. We aim to leverage these competences, and strive to make SPL integration into development process similar to that of commonly used unit testing frameworks.

Consequently, we introduce the concept of a SPL-based performance unit test, which consists of a decision component, represented by an SPL formula associated with code, and a workload component, which provides workload needed to collect method performance data, which is in turn needed to evaluate the formula.

In the following, we use Java in most of the code listings to illustrate integration of SPL into a particular programming language. In general, however, SPL does not have any special language requirements, and can be mapped to any programming language. Obviously, the level of integration and comfort will vary between languages, depending on the features they provide.

5.1 Decision Part of a Performance Unit Test

We first consider the decision part of a performance unit test—an SPL formula capturing a performance assumption. Based on experience outlined in Section 6.2, we have identified the following primary use cases for developers wanting to capture a performance assumption.

UC1: Regression testing. A developer is convinced that a new version of a particular method has better performance, or wants to make sure that a new version of a method does not degrade performance. In both cases, the developer wants to make these assumptions explicit and automatically verifiable.

UC2: Performance documentation. A developer wants to describe the performance of a particular method by comparing it to performance of another method with a well-understood (performance) behavior, such as memory copying.

UC3: Assumption about third party code. A developer assumes certain specific (relative or absolute) performance from a third-party component and wants to make this assumption explicit in the code that relies on the third-party component.

In general, it is desirable to keep SPL formulas capturing performance assumptions close to the code they refer to. This is motivated by the well-known observation that externally maintained programming documentation tends to become outdated, in contrast to documentation that is integrated with the source code, where a developer is more likely to update it to reflect changes in the code. In case of performance assumptions, we aim to achieve the same to prevent false alarms that could result from outdated performance assumptions.

The Java language supports annotations as a means to associate meta data with various program elements. Consequently, when we want to keep SPL formulas as close to the code as possible, we attach them to relevant methods using the `@SPL` annotation. Due to limited expressive power of Java annotations (and limited support for domain-specific languages in general), we can only attach SPL formulas to methods as plain strings. This forces us to perform SPL syntax checking outside the IDE, which reduces the level of comfort provided to a developer, but it is still preferable to using special comments, which clutter the source code. Moreover, annotations are retained in the class file, and can be processed using a convenient API.

An example demonstrating UC2 and the use of the `@SPL` annotation is shown in Listing 1. The method `getProperty()` is annotated with an SPL formula capturing the assumption that accessing named property in a collection of properties has the same performance as accessing a hash table.

```
class Properties {
  @SPL("THIS = HashMap.get()")
  public String getProperty(String name) {
    /* ... */
  }
}
```

Listing 1: *Annotation to document expected performance of a method*

Performance assumptions intended to prevent performance regressions (UC1) can be also attached to code using annotations, especially when they refer to the current version of a method. However, SPL formulas in the context of UC1 can also refer to method versions that no longer exist in the active code base, e.g. when a new version of a method is introduced.

We believe that the performance assumption refering to past versions of the method should be kept as a documentation of implementation decisions during project development, but because it refers to code that is no longer active, it should not clutter the active code base. To this end, SPL formulas can be also stored in a separate file with a descriptive name.

An example demonstrating UC1 with an SPL formula in an external file is shown in Listing 2. The formula captures the assumption that version 2 of myMethod() is faster than version 1. Because using fully qualified method names would result in formulas that are very difficult to read, we first define aliases for the two versions of the method we want to compare, and then use the aliases in the SPL formula. To refer to a particular method version, we attach a VCS revision identifier to the name.[6]

```
ver1 = myMethod@ver1
ver2 = myMethod@ver2

ver2 <= ver1
```

Listing 2: *Documenting improvements of a specific method*

Generally, performance assumptions of this kind should not fail during subsequent development. If they do, it may be due to changes in the environment, e.g. updating a third-party library or switching to a new version of a virtual machine. In such cases, it is useful to know about the impact of the change, and perhaps modify the method if the change in the environment is there to stay.

5.2 Workload Part of a Performance Unit Test

For a performance unit test to be complete, it must specify the workload for the methods referenced by the SPL formula in the decision part of the test. This allows the SPL formula to be evaluated using method performance data collected by executing the methods repeatedly with the test-specific workload and measuring the duration of the execution. Although it is technically possible to collect performance data at run-time by instrumenting the application, here we focus solely on collecting method performance data at build time.

To leverage the developer experience with writing unit tests, we provide support for producing performance data in a way that resembles writing small benchmarks or unit tests. A unit test generally consists of the setup, execution, validation and cleanup phases [2, 14]. The execution phase contains code to drive the operation under test, while the validation phase ensures that the operation under test did what it was supposed to do.

Our workload implementation adopts the same basic structure, with some modifications. The execution phase contains both code that prepares the workload data and code that performs the measured operation. The measured operation needs to be explicitly marked by the developer—this is useful for measuring larger parts of code than just single method invocation.

[6] In many versioning systems, the commit identification is not known before the actual commit. Therefore the act of *improving method X and writing the corresponding SPL formula* may require two commits, so that the SPL formula can reference the identifier of the previous commit.

In contrast to unit testing, the measured operation needs to be executed multiple times to collect a representative amount of performance data. The number of iterations needed for the data to be considered representative is not known statically, the developer therefore needs to put the measured operation inside a loop with the loop termination condition controlled externally by the SPL evaluation framework. Following Section 4.2, the framework can, for example, detect compilation events and discard warm-up data to obtain steady-state performance results.

An example of such a test-like performance data generator is shown in Listing 3. The saxBuilderTest() method prepares input data for the operation under test (SAXBuilder.build()) and executes the data collection loop. In the loop body, the input data is reinitialized before executing the measured operation, which is delimited by the calls to the SPLController.start() and SPLController.end() methods. The calls to these methods can be omitted if the loop body contains only code of the measured operation.

```
import org.jdom2.input;
import org.jdom2.Document;
import java.nio.file.*;
import java.io.*;

void saxBuilderTest(SPLController spl, String filename) {
  byte[] data = Files.readAllBytes(Paths.get(filename));
  InputStream is = new ByteArrayInputStream(data);
  SAXBuilder sax = new SAXBuilder();
  Document xml = null;

  while (spl.needsMore()) {
    is.reset();

    spl.start();
    xml = sax.build(is);
    spl.end();
  }
}
```

Listing 3: *Performance data generator for SAXBuilder.build()*

The need for an explicit loop in the performance data generator method deserves attention, because the compiler can, for example, move an invariant outside of the loop, thus leaving the measured code smaller and rendering the measurements useless. We believe that it is responsibility of the developer to take care of this issue – one remedy is moving the critical code into a separate function and indicating to the virtual machine to not inline the function, as implemented in the Java Microbenchmark Harness.[7]

Unlike a unit testing framework, which executes all test methods using a single virtual machine, the SPL evaluation framework executes each performance data generator in a clean virtual machine, i.e. in a separate JVM process. This is to prevent pollution of the virtual machine by the activity of previously executed generators, which may influence the performance measurement. The setup and cleanup phases of the generator are therefore intended only for source-level sharing of code.

The workload implementation can be turned into a complete performance unit test if we include a validation phase that actually triggers the evaluation of an SPL formula. However, this only makes sense if the formula relates solely on data measured in the workload implementation body. In case of more complex formulas, the SPL evaluation framework collects performance data on demand, based on the SPL formulas it needs to evaluate. The order of execution of individual workload implementations is outside the developer control. Explicitly requesting evaluation of an SPL formula at the end of a workload implementation body could therefore fail due to the SPL evaluation framework lacking performance data for other methods referenced in the formula (unless the assertion were actually a blocking function). Performance tests comparing different methods should therefore associate SPL formulas either directly with the corresponding methods, or place them in separate files, as described earlier in this section.

The workload implementation shown in Listing 3 is the most generic approach that can be used in most common use cases. As such, it does not necessarily fit some special situations, such as when a

[7] <http://openjdk.java.net/projects/code-tools/jmh>

test compares different methods with identical signatures. This includes comparing different versions of the same method to verify performance improvement or prevent performance regressions. It also includes comparing different implementations of the same interface methods, for example, alternative implementations of collection interfaces.

Such situations are frequent enough to justify a special mapping, where the generation of the workload (i.e. arguments for the measured operation) is separated from the measured operation. In addition, the allocation and initialization of the instance on which the measured operation will be called is separated into an instance generator. The measurement is entirely controlled by the SPL evaluation framework, which uses the workload and instance generators to prepare arguments for the measured operations.

The workload part of a performance unit test illustrating this separation is shown in Listing 4. The individual factory*() methods allocate and initialize the instance object using the prepare() method, and the workload generator represented by the randomSearch() method prepares arguments for the measured method (Collection.contains()) – the measured operation represents a search for a random integer in a collection. The two generators and the measured method are bound together in an SPL formula representing the decision part of the test, shown in Listing 5.

```
Object prepare(Collection<Integer> col, int size) {
  for (int i = 0; i < size; i++) {
    col.add(i);
  }
  return col;
}

Object factoryLinkedList(int size) {
  return prepare(new LinkedList<Integer>(), size);
}

Object factoryArrayList(int size) {
  return prepare(new ArrayList<Integer>(), size);
}

void randomSearch(MethodArguments args, int size) {
  args.set(0, Random.nextInt(size));
}
```

Listing 4: *Workload part of a performance unit test with separated instance and workload generation*

```
arraylist = ArrayList.contains[randomSearch, factoryArrayList]
linkedist = LinkedList.contains[randomSearch, factoryLinkedList]
for i in 10, 100, 500, 1000
  linkedlist(i) > arraylist(i)
```

Listing 5: *An SPL formula binding the workload and instance generators to the measured method*

Compared to the performance data generator shown in Listing 3, the disadvantage of this mapping is that the code is more scattered and more difficult to understand – the measured operation is not immediately obvious and is a result of combining the generators and the measured method. On the other hand, a performance unit test employing this separation can be easily extended to include additional types (e.g. instances of the Vector class for the example in Listing 4), without disrupting the existing code base.

5.3 Mapping to Other Languages

The mapping of SPL to code very much depends on the features provided by the target programming language. For illustration purposes, the examples presented so far focused on the Java language. However, we have also developed a mapping of SPL into C# [35], which exploits advanced features of the language to provide a mapping that is more code-oriented.

An example of a complete performance unit test is shown in Listing 6. The IFunctor instances serve as pointers to the measured method. The workload is an array of randomly generated integers, prepared using the generator interfaces. The performace unit test (instance of the Experiment class) is composed from the functor and the generator, and an evaluation strategy, i.e. comparison of performance data (the $<$ operator in the lambda function passed to the constructor of the Experiment class performs a statistical hypothesis test on the two means). The validation phase of the test consists of a standard assert which checks the value of the formula encoded in the Experiment instance.

```
[TestMethod]
public void InsertionSort_FasterThan_BubbleSort() {
  IFunctor insSort = FunctorFactory.Create(new InsertionSort(),
    insertion => insertion.Sort(Arg.Of<IList<int>>()));

  IFunctor bubSort = FunctorFactory.Create(new BubbleSort(),
    bubble => bubble.Sort(Arg.Of<IList<int>>()));

  Random rand = new Random();
  ArrayComposer <int> composer = new ArrayComposer <int>(1000,
    () => rand.Next());
  IArgumentGenerator gen = new CompositeGenerator() {
    () => composer.GetNext()
  };

  Experiment test = new Experiment(
    (insertion , bubble , mean)
    => mean(insertion) < mean(bubble));

  test.Methods["insertion"] = new MethodOperand(insSort, gen);
  test.Methods["bubble"] = new MethodOperand(bubSort, gen);

  bool result = Engine.Evaluate(test, true);
  Assert.IsTrue(result);
}
```

Listing 6: *Performance test of insertion and bubble sort in C#*

The implementation of our performance testing environment includes additional features not described here, for example support for the Git and Subversion version control systems and integration with Eclipse and Hudson. Some of the features described here are work in progress. The implementation is available as open source at [33].

6 Overall Evaluation

It is our experience that observations collected from real measurements provide few guarantees and many surprises. Although we have derived many properties of the SPL formalism and the SPL interpretations theoretically, we still need to evaluate their practical application. To begin with, we evaluate the sensitivity of the SPL interpretations. Next, we use the SPL formalism to express and validate developer assumptions about performance. Finally, we look at the potential for portability of performance requirements expressed in terms of relative comparison.

6.1 Interpretation Sensitivity

A practical concern related to the SPL interpretations is how many false positives (situations where the interpretation reports a significant performance difference even when none truly exists) and false negatives (situations where the interpretation does not report a difference even when one does exist) can be expected. In practical scenarios, it is difficult to define this criterion exactly – measurements always exhibit some differences and deciding whether the differences are essential or incidental is often a matter of perspective.

To evaluate the ability of the SPL interpretations to distinguish essential and incidental differences in performance, we construct an evaluation scenario around a method that builds a DOM data structure from an XML input.[8] We measure the method in multiple runs with multiple input sizes and use the individual SPL interpretations to decide whether the measured performance differs for various combinations of input size and run count. In detail, for interpretation i, input size s and run count r:

1. We use random sampling to select r runs with input size s into a set of measurements M_s, to represent the observations of the method performance on input size s, denoted $P(s)$.
2. We use random sampling to select r runs with base input size b into a set of measurements M_b, to represent the observations of the method performance on base input size b, denoted $P(b)$.
3. We use the interpretation i on M_s and M_b to decide whether $P(s) \geq_{p(id,id)} P(b)$ and $P(s) \leq_{p(id,id)} P(b)$ at significance $\alpha = 0.01$.

We repeat the steps enough times to estimate the probability of the individual decisions. The results are available on Figure 6, with the interpretations from Sections 4.1, 4.3 and 4.4 denoted as *Welch*, *Parametric* and *Non-Parametric*, respectively. Each run collects $o = 20000$ observations after a warmup of 40000 observations, the input size ranges from about 5000 XML elements in a file of 282 kB to about 5600 XML elements in a file of 316 kB (base input size). In total 130 runs were used to derive the distribution $B_{\overline{M},r_M,o_M-\overline{N},r_N,o_N}$, approximated by $B^*_{\overline{X},r_X,o_X-\overline{Y},r_Y,o_Y}$ for $X = Y = M(s)$. For each input size, 10 runs were available for random sampling.

To interpret the results on Figure 6, we first look on the measurements with zero file size difference, that is, the measurements where the interpretation was asked to decide whether there is a discernible difference between performance of the same method under the same workload. Any rejection for zero file size difference constitutes an evident false positive. We see that the interpretation from Section 4.1 is incorectly rejecting the null hypothesis very often regardless of the number of runs used. The parametric interpretation from Section 4.3 is only incorrectly rejecting the null hypothesis for a small number of runs. The non-parametric interpretation from Section 4.4 is almost never rejecting the null hypothesis.

Next, we look on the measurements with non-zero file size difference. Here, chances are that the file size difference is also reflected in the performance – to provide a rough estimate, it takes on average 9.04 ms to execute the method for the largest input size and 7.81 ms to execute the method for the smallest input size. We therefore want the interpretation to reject the hypothesis that $P(s) \geq_{p(id,id)} P(b)$ – in other words, to identify that there actually is a performance difference ; but this is only useful when the interpretation also never rejects $P(s) \leq_{p(id,id)} P(b)$ – in other words, it should identify the performance difference reliably. We see that the interpretation from Section 4.1 is very willing to identify performance differences, however, it is not really reliable until the difference is very large and the number of runs is sufficient. The parametric interpretation from Section 4.3 is reasonably reliable except when given a small number of runs, and the sensitivity to performance differences depends on the number of runs available. The non-parametric interpretation from Section 4.4 is reliable even when used with just one run.

To summarize, the interpretation from Section 4.1 worked reasonably well for relatively large performance differences. The more sophisticated parametric and non-parametric interpretations from Sections 4.3 and 4.4 exhibited similar sensitivity to performance differences and were more reliable – the parametric interpretation only with more runs, the non-parametric interpretation even with fewer runs. This is not counting the runs used to build the necessary distribution of the test statistics. In practical settings, new runs would be measured and assessed as machine time permits. The non-parametric interpretation could be used with the distribution built from the historical measurements to quickly assess the initial runs, and the parametric interpretation could refine the assessment after more runs become available.

6.2 Expressing and Validating Developer Assumptions

To evaluate the ability of the SPL formalism to express and validate developer assumptions about performance, we perform a retroactive case study – we take the JDOM library, augment it with tests and evaluate the results. By looking for keywords such as "performance", "refactor", "improve" or "faster" in the commit log, we have identified commits which the developers consider relevant to performance.

[8] The method is SAXBuilder::build, used to build a DOM tree from a byte array stream, from the JDOM library, executing on Platform Alpha.

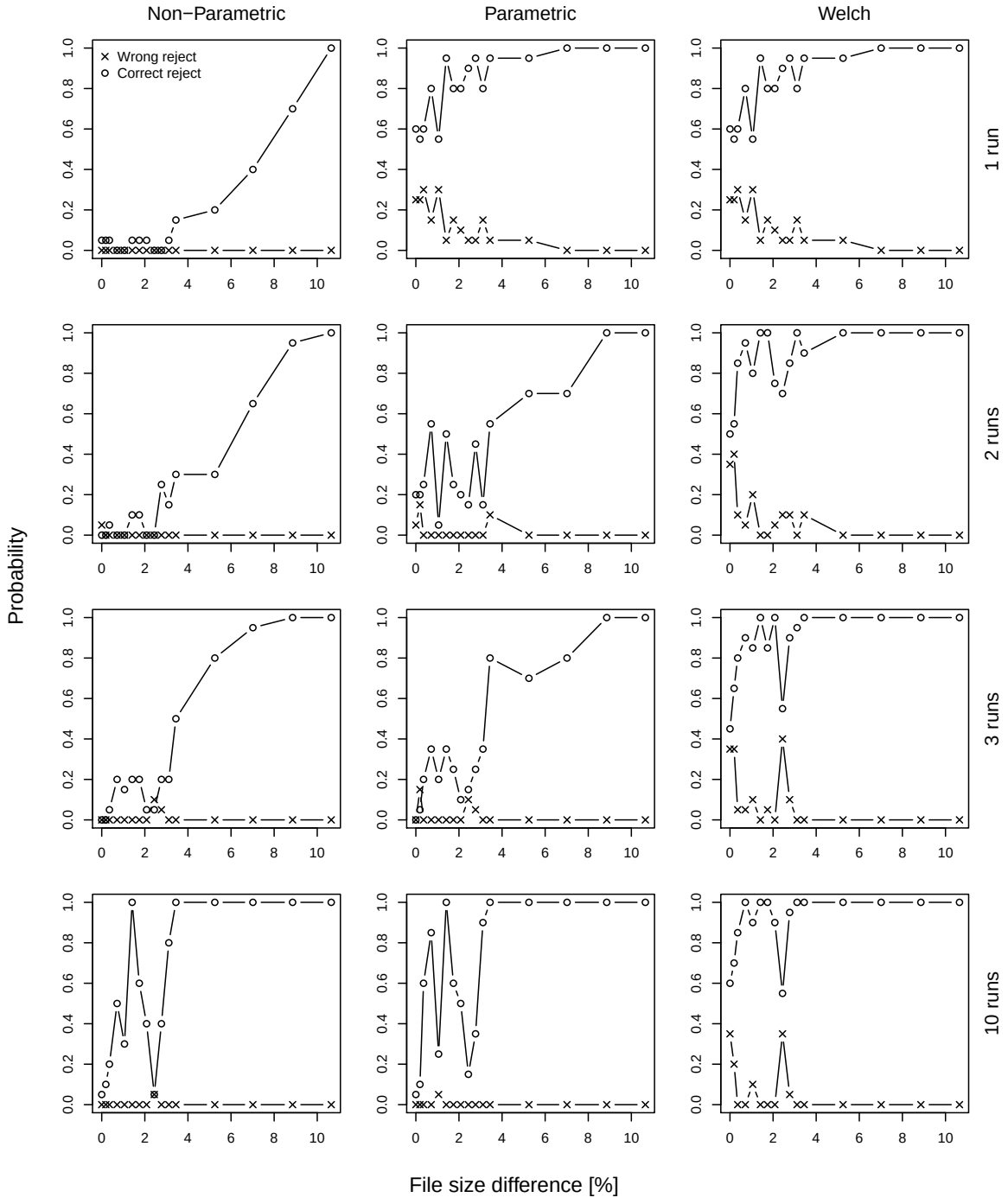

Figure 6: *Sensitivity to input size change for combinations of interpretation and run count.*

Method	Commit	Median	10% Q	90% Q
SAXBuilder.build	6a49ef6	9.2 ms	9.1 ms	9.3 ms
SAXBuilder.build	4e27535	11.2 ms	11.1 ms	11.3 ms
Verifier.checkAttributeName	500f9e5	22.5 ms	22.5 ms	22.6 ms
Verifier.checkAttributeName	4ad684a	20.0 ms	19.9 ms	20.1 ms
Verifier.checkAttributeName	e069d4c	21.4 ms	21.3 ms	21.5 ms
Verifier.checkAttributeName	1a05718	25.2 ms	25.2 ms	25.4 ms
Verifier.checkElementName	e069d4c	31.6 ms	28.6 ms	31.7 ms
Verifier.checkElementName	1a05718	42.6 ms	41.3 ms	42.8 ms

Table 1: *Selected measurement results. The 10% Q and 90% Q columns show the 10% and 90% quantiles.*

We have then written tests that express the developer assumptions about performance, using data from the performance study [17] in the workload generators. In all, we have 103 performance comparisons over 102 measurements in 58 tests across 46 commits. All SPL formulas had the same general form, simply comparing two methods.

In the following sections, we select three examples to illustrate the typical developer assumptions we test. The measurement results for the examples are given in Table 1, collected on Platform Bravo.

6.2.1 Case I: Negative Improvement

The first example shows a situation where the developers believed a change will improve performance significantly, when the opposite was actually true. The change was introduced with this commit 4e27535 message: "instead of using the slow and broken PartialList to make lists live, we'll be using a *faster and smarter* FilterList mechanism ... it *should be faster* and consume fewer resources to traverse a tree" [16].

Table 1 shows the performance from this commit and the preceding commit 6a49ef6 for the build() method of SAX builder. Instead of the expected performance improvement, the change actually increased the median execution time by 22%.

6.3 Case II: Confirmed Improvement

The second example shows a successful performance improvement confirmed by the performance test. Commit 4ad684a focused on improving performance of the Verifier class after the developers made their own performance evaluation [17]. Comparison with the preceding commit 500f9e5 in Table 1 indicates the method execution time decreased by 11%.

6.4 Case III: Measurement

The last example shows a somewhat creative use of a performance test. We focus on a situation where the developers actually expect a performance problem and want to assess the magnitude. Such situations can arise for example when the code is refactored for readability, possibly assuming that performance optimizations would be applied later, or when more complex code replaces previous implementation.

In the JDOM project, this happened for example between commit e069d4c and commit 1a05718, where modifications bringing better standard conformance were introduced: "bringing the letter/digit checks in line with the spec ... following the BNF productions more closely now" [16]. By adding a test that refers to the execution time of a particular method in both versions, the developers obtain a report that helps assess the magnitude of the performance change. Table 1 shows the execution times of the checkAttributeName() and the checkElementName() methods, with the median execution time increasing by 18% and 35% respectively. Note that the execution times refer to multiple consecutive invocations, since one method invocation would be too short to measure accurately.

Our experience with these and other examples suggests that unit testing of performance would help the developers confront their assumptions with measurements with relatively little effort. We have observed six cases where the developer assumptions were refuted by the tests, which is about one tenth of all examined assumptions – the retroactive character of our case study does not allow to guess

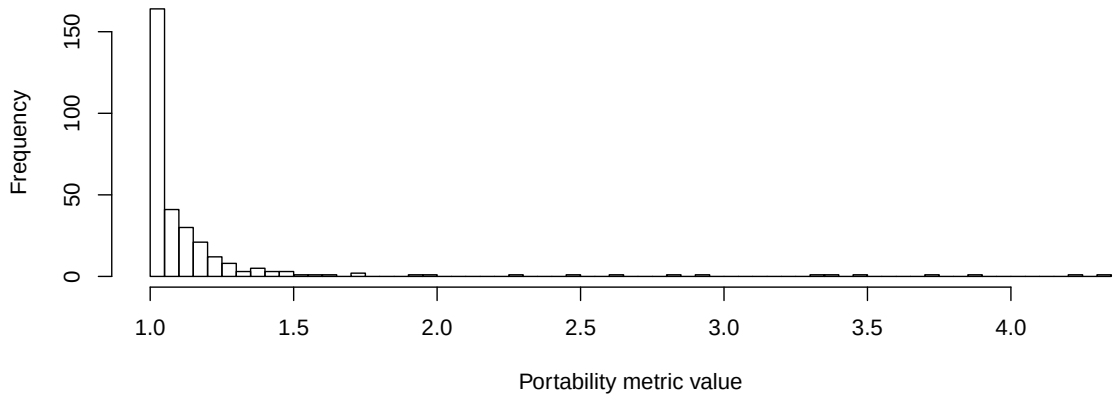

Figure 7: *Histogram of portability metric values.*

the developer reaction, however, we can still assume the developers would find the feedback helpful. For interested readers, we have made the complete source of our tests and the complete measurement results available on the web at [33].

6.5 Platform Portability

Among the features of SPL is the ability to compare performance of multiple functions against each other. This feature is motivated by the need to make the test criteria reasonably portable – while it is generally not possible to make accurate conclusions about performance on one platform from measurements on another, running the same test on similar platforms should ideally lead to similar conclusions. As the last evaluation experiment, we therefore look at how the test results differ between three different platforms, here labeled Bravo, Charlie, Delta.

We define a metric that describes the relative difference between results of a single test on two different platforms. Assuming a test condition that compares performance of functions M and N on platforms 1 and 2, we compute the ratio $(\bar{M}_1/\bar{N}_1)/(\bar{M}_2/\bar{N}_2)$ or its reciprocal, whichever is greater, where $\bar{X}_i$ denotes the mean execution time of method X on platform i. A perfectly portable test condition would preserve the ratio $\bar{M}/\bar{N}$ on all platforms, giving the portability metric value of one.

Figure 7 shows a histogram of the portability metric values for all test and platform pairs in our case study. Most portability metric values are very close to one, with 96% of values smaller than two and no value greater than five. This leads us to believe most tests in our case study are indeed reasonably portable.

7 Related Work

PSpec [27] is a language for expressing performance assertions that targets goals similar to ours, namely regression testing and performance documentation. The performance data is collected from application logs and checked against the expected performance expressed as absolute limits on performance metrics (such as execution time).

Performance assertions based on the PA language are introduced in [36]. The PA language provides access to various performance metrics (both absolute and relative) as well as key features of the architecture and user parameters. Similar to ours, the assertions themselves are part of the source code. The assertions are checked at runtime and support local behavior adaptations based on the results, however, use for automated performance testing is not considered.

PIP [29] describes a similar approach, exploiting declarative performance expectations to debug both functional and performance issues in distributed systems. In contrast to SPL, PIP includes a specification of system behavior and the expected performance is declared in the context of this behavioral specification. PIP uses application logs to obtain the performance data and uses absolute performance metrics (such as processor time or message latency) in performance expectations.

Complementary to our method, [22] describes system-level performance expectations imperatively, using programmatic tests of globally measured performance data. The measurements are performed at runtime via injected probes and the data is analyzed continuously.

Similarly, [34] employs the A language to express validation programs concerning both business logic and performance characteristics (such as balanced processor load) of distributed services. The method focuses mainly on runtime validation of operator actions and static configuration.

Among the tools targeted at performance unit testing, JUnitPerf [6] and ContiPerf [3] are notable extensions of the JUnit [18] functional unit testing framework. Both tools use absolute time limits to specify performance constraints evaluated by the tests. Both JUnitPerf and ContiPerf support execution of the function under test by multiple threads in parallel. Thanks to integration with JUnit, both tools are also supported by environments that support JUnit itself, including Eclipse and Hudson.

Also related to our work are projects for continuous testing. Among those, the Skoll project [28] is a decentralized distributed platform for continuous quality assurance. The execution is feedback-driven – each quality assurance activity is represented by a task that is executed and once its results are analyzed, other tasks are scheduled as necessary. Multiple strategies for executing dependent tasks are used to isolate individual problems.

The DataMill project [24] offers a heterogeneous environment for running tests on different operating systems or on different hardware platforms. DataMill is not concerned with analysis – instead, the goal is to allow testing software on different platforms. This makes DataMill an important complement to many performance testing approaches to resolve the issues related to repeatability of performance measurements.

On the system testing level, Foo et al. [10] stress the need for automated approach to discover performance regressions. The proposed approach is based on monitoring a set of performance metrics, such as the system load, and comparing the measurements across different releases. Data-mining and machine-learning techniques are used to correlate the collected metrics, creating performance patterns that can indicate performance regressions.

Also on the system testing level, Ghaith et al. [11] propose to use transaction profiles to identify performance regressions. The profiles are constructed from resource utilization and do not depend on workload. Comparison of transaction profiles reveals the regression. In contrast, our approach captures workload dependent performance and expects the developers to provide application specific workload generators.

Beyond the performance testing itself, the problem of identifying the change that caused a performance regression is tackled in [12]. The authors use functional unit tests as a basic for monitoring performance regressions. Commit history bisection is used to identify a particular revision, measurement on individual levels of the call tree are used to locate the regression in code.

8 Conclusion

Pursuing the goal of applying unit tests for performance testing activities, we have presented Stochastic Performance Logic (SPL), a mathematical formalism for expressing and evaluating performance requirements, and an environment that permits attaching performance requirement specifications to individual methods to define common performance tests.

As a distinguishing feature, SPL formulas express performance requirements by comparing performance of multiple methods against each other, making it easy for the developer to express common performance related assertions – we have evaluated the ability of the SPL formulas to express real developer assumptions by retroactively implementing and evaluating performance tests based on the development history of the JDOM project.

The evaluation of SPL formulas relies on a formally defined logic interpretation. Three such interpretations were developed and evaluated – one for simple scenarios where large performance differences need to be detected, two for more demanding scenarios where the measurements fluctuate due to factors outside experimental control and where the measurement procedure can collect observations from multiple runs.

Our work is backed by an open source implementation that supports writing and evaluating performance unit tests in the Java environment, complete with Git and Subversion support and Eclipse and Hudson integration, and an additional prototype implementation for the C# environment. The implementations and complete experimental data presented in the paper are available from [33].

References

[1] H. P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*. North-Holland, Amsterdam, 1984.

[2] Kent Beck. *Simple Smalltalk Testing*. Cambridge University Press, 1997.

[3] Volker Bergmann. *ContiPerf 2*, 2013. <http://databene.org/contiperf.html>.

[4] P. Bourque and R.E. Fairley. *Guide to the Software Engineering Body of Knowledge, Version 3.0*. IEEE Computer Society, 2014.

[5] Lubomir Bulej, Tomas Bures, Jaroslav Keznikl, Alena Koubkova, Andrej Podzimek, and Petr Tuma. Capturing Performance Assumptions using Stochastic Performance Logic. In *Proc. ICPE 2012*. ACM, 2012.

[6] Mike Clark. *JUnitPerf*, 2013. <http://www.clarkware.com/software/JUnitPerf>.

[7] Cliff Click. The Art of Java Benchmarking. <http://www.azulsystems.com/presentations/art-of-java-benchmarking>.

[8] D. Dice, M.S. Moir, and W.N. Scherer. Quickly reacquirable locks, October 12 2010. US Patent 7,814,488.

[9] Dave Dice. *Biased Locking in HotSpot*, 2006. <https://blogs.oracle.com/dave/entry/biased_locking_in_hotspot>.

[10] King Foo, Zhen Ming Jiang, B. Adams, A.E. Hassan, Ying Zou, and P. Flora. Mining performance regression testing repositories for automated performance analysis. In *Proc. QSIC 2010*. IEEE, 2010.

[11] S. Ghaith, Miao Wang, P. Perry, and J. Murphy. Profile-based, load-independent anomaly detection and analysis in performance regression testing of software systems. In *Proc. CSMR 2013*. IEEE, 2013.

[12] Christoph Heger, Jens Happe, and Roozbeh Farahbod. Automated root cause isolation of performance regressions during software development. In *Proc. ICPE 2013*. ACM, 2013.

[13] Vojtěch Horký, František Haas, Jaroslav Kotrč, Martin Lacina, and Petr Tůma. Performance regression unit testing: A case study. In MariaSimonetta Balsamo, WilliamJ. Knottenbelt, and Andrea Marin, editors, *Computer Performance Engineering*, volume 8168 of *Lecture Notes in Computer Science*, pages 149–163. Springer Berlin Heidelberg, 2013.

[14] IEEE standard for software unit testing. *ANSI/IEEE Std 1008-1987*, 1986.

[15] *JDOM Library*, 2013. <http://www.jdom.org>.

[16] *hunterhacker/jdom [Git]*, 2013. <https://github.com/hunterhacker/jdom>.

[17] *hunterhacker/jdom: Verifier performance*, 2013. <https://github.com/hunterhacker/jdom/wiki/Verifier-Performance>.

[18] *JUnit Tool*, April 2013. <http://junit.org>.

[19] Tomas Kalibera, Lubomir Bulej, and Petr Tuma. Automated detection of performance regressions: the mono experience. In *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2005. 13th IEEE International Symposium on*, pages 183–190, Sept 2005.

[20] Tomáš Kalibera, Lubomír Bulej, and Petr Tůma. Benchmark Precision and Random Initial State. In *Proc. SPECTS 2005*. SCS, 2005.

[21] Peter Libič, Lubomír Bulej, Vojtěch Horky, and Petr Tůma. On the limits of modeling generational garbage collector performance. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*, ICPE '14, pages 15–26, New York, NY, USA, 2014. ACM.

[22] Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. D3S: Debugging Deployed Distributed Systems. In *NSDI'08*. USENIX, 2008.

[23] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong. In *Proceedings of ASPLOS 2009*, New York, NY, USA, 2009. ACM.

[24] Augusto Oliveira, Jean-Christophe Petkovich, Thomas Reidemeister, and Sebastian Fischmeister. Datamill: Rigorous performance evaluation made easy. In *Proc. ICPE 2013*. ACM, 2013.

[25] Oracle. *JVM Tool Interface*, 2006. <http://docs.oracle.com/javase/6/docs/platform/jvmti/jvmti.html>.

[26] Oracle. *Garbage Collector Ergonomics*, 2014. <http://docs.oracle.com/javase/7/docs/technotes/guides/vm/gc-ergonomics.html>.

[27] Sharon E. Perl and William E. Weihl. Performance assertion checking. *SIGOPS Oper. Syst. Rev.*, 27, 1993.

[28] Adam Porter, Cemal Yilmaz, Atif M. Memon, Douglas C. Schmidt, and Bala Natarajan. Skoll: A process and infrastructure for distributed continuous quality assurance. *IEEE Trans. Softw. Eng.*, 33(8):510–525, 2007.

[29] Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vahdat. Pip: Detecting the Unexpected in Distributed Systems. In *NSDI'06*. USENIX, 2006.

[30] Andreas Sewe, Mira Mezini, Aibek Sarimbekov, and Walter Binder. Da capo con scala: Design and analysis of a scala benchmark suite for the java virtual machine. In *Proceedings of OOPSLA 2011*, OOPSLA '11, pages 657–676, New York, NY, USA, 2011. ACM.

[31] David J. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures*. CRC Press, 2011.

[32] Aleksey Shipilev. Assorted performance presentations. <http://shipilev.net>.

[33] *SPL Tool*, 2013. <http://d3s.mff.cuni.cz/software/spl>.

[34] A. Tjang, F. Oliveira, R. Bianchini, R.P. Martin, and T.D. Nguyen. Model-Based Validation for Internet Services. In *Proc. 28th IEEE Intl. Symp. on Reliable Distributed Systems*, 2009.

[35] Tomas Trojanek. Capturing performance assumptions using stochastic performance logic. Master's thesis, Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic, 2012.

[36] Jeffrey S. Vetter and Patrick H. Worley. Asserting Performance Expectations. In *Proc. 2002 ACM/IEEE Conf. on Supercomputing*, Supercomputing '02. IEEE CS, 2002.

[37] B. L. Welch. The Generalization of 'Student's' Problem when Several Different Population Variances are Involved. *Biometrika*, 34, 1947.