Vojtěch Horký

Performance Awareness in Agile Software Development

Department of Distributed and Dependable Systems

Supervisor of the doctoral thesis: prof. Ing. Petr Tůma, Dr.
Study programme: Informatics
Specialization: Software Systems

Prague 2018
I declare that I carried out this doctoral thesis on my own (the papers included in Part II have been written in cooperation with their respective co-authors), and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague, March 9, 2018
I would like to thank my advisor Petr Tůma for his encouragement, support and guidance through my study. A big thanks also goes to all my colleagues from our department and especially to my colleagues from the room 205. They all made the study a pleasant and interesting experience. And it was fun too.

Above all I would like to thank my family, especially my wife Naď’a for their continuous support.
Performance Awareness in Agile Software Development

Vojtěch Horký
vojtech.horky@d3s.mff.cuni.cz

prof. Ing. Petr Tůma, Dr.
petr.tuma@d3s.mff.cuni.cz

Department of Distributed and Dependable Systems
Faculty of Mathematics and Physics
Charles University
Malostranské nám. 25, 118 00 Prague, Czech Republic

Abstract

Broadly, agile software development is an approach where code is frequently built, tested and shipped, leading to short release cycles. Extreme version is the DevOps approach where the development, testing and deployment pipelines are merged and software is continuously tested and updated.

In this context, our work focuses on identifying spots where the participants should be more aware of the performance and offers approaches and tools to improve their awareness with the ultimate goal of producing better software in shorter time. In general, the awareness is raised by testing, documenting, and monitoring the performance in all phases of the development cycle.

In this thesis, we (1) show a framework for writing performance tests for individual components (e.g. libraries). The tests capture and codify assumptions about the performance into runnable artifacts that simplify repeatability and automation. For evaluation of the performance tests, we (2) propose new methods, which can automatically detect performance regressions. These methods are designed with inherent variation of performance data in mind and are able to filter it out in order to detect true regressions. Then we (3) reuse the performance tests to provide the developers with accurate and up-to-date performance API documentation that steer them towards writing efficient solutions from the beginning. Finally, we (4) explore the practical limits of using runtime monitoring to gather performance data from production environments. This allows us to assess whether it is possible to use production performance data in the performance tests and thus validate that the assumptions also hold outside – inherently simplified – testing environment.

Keywords

performance evaluation, agile software development, performance testing, software documentation, automated software testing
Anotace

Název Výkon softwaru jako faktor při agilních metodách vývoje
Autor Vojtěch Horký
vojtech.horky@d3s.mff.cuni.cz
Školitel prof. Ing. Petr Tůma, Dr.
petr.tuma@d3s.mff.cuni.cz
Katedra Katedra distribuovaných a spolehlivých systémů
Matematicko-fyzikální fakulta
Univerzita Karlova v Praze
Malostranské nám. 25, 118 00 Praha 1, ČR

Abstrakt
Za agilní metody vývoje softwaru jsou obecně považovány přístupy, kdy jsou programy často sestavovány, testovány a nasazovány. Výsledkem je tak kratší vývojový cyklus. Přístupy typu DevOps pak dovádí tuto koncepci do extrému, kdy jsou setřeny rozdíly mezi vývojovým a produkčním prostředím a nasazený software průběžně aktualizují.
V tomto kontextu se tato práce zaměřuje na nalezení míst, kde by jednotliví účastníci měli mít větší povědomí o výkonu vyvíjeného softwaru. Práce nabízí přístupy a nástroje jak toto povědomí zvýšit; hlavním cílem je vytvářet lepší (rychlejší) software v kratším čase. Zlepšení je dosaženo pomocí testování, dokumentace a sledování výkonu během všech fází vývoje software.
V této práci ukážeme (1) nástroje pro psaní testů výkonu pro jednotlivé komponenty (např. knihovny). Tyto testy zachycují a kodifikují předpoklady o výkonu a převádí je do spustitelných entit, které zjednodušují automatizaci a opakovatelnost. Pro vyhodnocení testů výkonu jsme (2) navrhlí nové metody které dokáží automaticky nalézt regrese. Tyto metody jsou navrženy tak, aby braly v úvahu variabilitu dat pocházejících z měření výkonu softwaru a dokázaly odlišit skutečné regrese od šumu. Testy výkonnosti pak také (3) zužitkujeme pro vytvoření aktuální a přesné API dokumentace výkonu, která vývojářům usnadní psaní efektivních programů od začátku vývoje. Závěrem také (4) zkoumáme praktická omezení metod, která měří výkon softwaru při produkčním nasazení. Znalost těchto omezení nám umožní zjistit, zda-li je možné použít tato data v původních testech a tím ověřit, zda dostatečně věrně reprezentují skutečné zatížení systému.

Klíčová slova vyhodnocování výkonnosti, agilní metody vývoje softwaru, testování výkonu, softwarová dokumentace, automatizované testování softwaru
Contents

1 Introduction and Contribution Overview 3

1 Performance Awareness in Software Development 5
   1.1 Performance Awareness .......................... 7
   1.2 Software (Code) Documentation .................. 8
   1.3 Build-Time Testing ............................... 10
   1.4 Automated Regression Detection .................. 12
   1.5 Runtime Software Monitoring ...................... 13
   1.6 Robust Data Comparison ........................... 14

2 Goals in Detail 17

3 Overview of Contribution 19
   3.1 Performance Unit Testing .......................... 19
      3.1.1 Stochastic Performance Logic .................. 20
      3.1.2 Deployment in Java ............................. 21
      3.1.3 Framework Evaluation ......................... 23
   3.2 Robust Regression Detection ....................... 25
      3.2.1 SPL Interpretations ............................. 25
      3.2.2 Evaluation – Interpretation Sensitivity ........ 26
   3.3 Performance Documentation ........................ 31
      3.3.1 Reusing Performance Tests ...................... 31
      3.3.2 Performance-extended Javadoc .................. 32
      3.3.3 Evaluation – performance-extended Javadoc ...... 34
   3.4 Analysis of Performance Monitoring Overhead .... 39
      3.4.1 Framework for Overhead Measurement .......... 39
      3.4.2 Evaluation: Overhead of Monitoring SPECjbb2015 40

II Collection of Papers 45

4 Preface 47

5 Unit testing performance with Stochastic Performance Logic 49

6 Analysis of Overhead in Dynamic Java Performance Monitoring 99
### Utilizing Performance Unit Tests To Increase Performance Awareness

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 Utilizing Performance Unit Tests To Increase Performance Awareness</td>
<td>113</td>
</tr>
</tbody>
</table>

### Related Work and Conclusion

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>III Related Work and Conclusion</td>
<td>127</td>
</tr>
</tbody>
</table>

#### 8 Related Work

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1 Performance Expectations Description</td>
<td>129</td>
</tr>
<tr>
<td>8.2 Performance Benchmarking</td>
<td>134</td>
</tr>
<tr>
<td>8.3 Anomaly Detection</td>
<td>138</td>
</tr>
<tr>
<td>8.4 Accuracy and Overhead of Performance Monitoring</td>
<td>140</td>
</tr>
</tbody>
</table>

#### 9 Conclusion

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.1 G1 (Performance Testable Code)</td>
<td>145</td>
</tr>
<tr>
<td>9.2 G2 (Robust Regression Detection Mechanism)</td>
<td>145</td>
</tr>
<tr>
<td>9.3 G3 (Performance Documentation)</td>
<td>146</td>
</tr>
<tr>
<td>9.4 G4 (Limits of Runtime Monitoring)</td>
<td>147</td>
</tr>
</tbody>
</table>

#### 10 List of Publications

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>149</td>
</tr>
</tbody>
</table>

#### 11 Bibliography

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>151</td>
</tr>
</tbody>
</table>
Part I

Introduction and Contribution
Overview
Software performance is important. This simple statement becomes obvious when looking at recent examples of major performance related problems. The failure of the Czech vehicle registry system in 2012 [21], the collapse of the HealthCare.gov website in 2013 [86], or the more recent crash of the ABS Census system in 2016 [10] are prominent examples of situations where issues with software performance caused serious problems. In fact, the magnitude of the malfunctions was big enough to lower public confidence in the entire systems.

The importance of software performance is also reflected in our understanding of software quality – for example, performance efficiency is listed as one of eight software quality characteristics in the ISO/IEC 25010: Standard for Systems and Software Quality Requirements and Evaluation [45]. Performance efficiency is defined as dealing with timing, capacity, and utilization requirements, and is put on par with other major characteristics such as functionality.

Software quality is managed throughout the software development process. Activities ranging from requirements specification to test plan preparation and execution, together with the associated tools, are designed specifically to make sure the software under development meets the quality requirements. Agile software development in particular pushes the software quality management activities close to the developer, aiming to shorten the development cycle and ideally prevent, rather than only detect and then contain, possible quality issues.

This thesis focuses at the push to manage software quality close to the developer, which is very much evident in practice today. For some quality characteristics, such as functionality or maintainability, there are well known tools such as JUnit [47] or CheckStyle [17]. However, there is a marked imbalance where performance is concerned – as our recent study shows,
from nearly 100 thousand open-source Java projects on GitHub about one third uses some kind of (functional) unit testing, but less than half percent (!) contains any kind of performance test [85].

Our goal is to address this imbalance and explore the possibilities of enhancing performance awareness in agile software development processes. In short, we want to make the developers aware of the eventual performance of their projects from early stages of the software lifecycle and help build the software from components that have known performance characteristics.

Following university regulations, the thesis is structured as a collection of published research papers accompanied by a motivating introduction and a concluding discussion. In Part I, we introduce the research problem – Chapter 1 outlines the context, Chapter 2 refines the research questions, and Chapter 3 concludes the introduction with a summary of the thesis contributions. Part II contains the introduction relevant to the topic of the thesis. Part III discusses related work and directions of future work.

A publication bears the names of people that contributed to it but it misses people that came afterwards and improved on the work or maintained it in some way. Many of the papers mentioned here are accompanied by a software implementation, often done by students whose thesis and projects the author advised. Therefore, the acknowledgements must be given to all the authors of the publications and also to the following students. The initial implementation of the SPL tools [37] was done by a team of the following students: František Haas, Jaroslav Kotrč and Martin Lacina. Jaroslav Kotrč then continued the work by implementing a prototype that bound together compile-time tests with performance data collected at runtime [51]. Extension of Javadoc with performance data was done by Jakub Náplava [67].

Although the following terms have a broad meaning in the computer science community, we will be using them in the following manner. Component refers to any well-defined part of a software, ranging from a runtime library to a one-line function (method). Performance of a component then refers to the speed (i.e. execution time or throughput) of an operation on that component. We use execution time as it is a basic metric available on virtually any platform and also because it retains its intuitive meaning regardless of the underlying system.

1.1 Performance Awareness

To introduce the concept of performance awareness, we return to the examples of expensive system failures that were caused by performance issues [21].

---

1Even in simple cases, the overall performance of software composed from multiple components is seldom a simple sum of its parts [66]. However, we believe that building from better components (e.g. performance tested ones) is more effective (i.e. less expansive) in the long run for achieving the required overall quality.
Although the reasons behind these failures are typically complex and often difficult to discern, let us assume that some had cause in the code— that is, the software contained a performance bug. We compare this situation with a similar hypothetical situation where a functional bug was introduced:

- When designing the functional aspects of code, developers rely on extensive functional documentation of the building blocks involved. They may also consult the code directly when the documentation is lacking. In contrast, the performance of the building blocks involved is rarely documented.

- In agile development, it is common practice to accompany code with unit tests of functionality, which the developer may even execute during development to assess progress. In contrast, unit tests of performance are rare [85, 55].

- The development environment can even automatically block attempts to commit or deploy code that fails functional tests. Again, automated blocking of code that fails performance tests is rare and may require significant engineering effort [69, 30].

Our points show that while the developer has ample opportunity to acquire awareness about code functionality during development, the same is not true where awareness about code performance is concerned. This is despite the fact that the importance of performance testing is known for a very long time [62, 8] and even the already mentioned ISO/IEC 25010 standard [45] describes performance related qualities in great detail. In practice, performance is still lagging behind functionality as a first-class citizen of the development lifecycle.

In the following text, we will draw more parallels between validating performance and other software quality attributes. We will be using the following hypothetical scenario that is simple enough to be tackled by one developer yet allows us to demonstrate the decisions taken during the implementation.

A developer is tasked with the following objective: merge results from multiple tools, where each output is in a separate XML file. There are multiple options how to solve this task, we will focus on a subset of the task which is the extraction of information from the XML file. The original motivation for this example comes from [41] (see Section 3.3.3 for an overview).

Following options are generally available regardless of the environment (programming language, libraries etc.).

- Use SAX-based API and extract the data in the callback routines.
- Use DOM-based API and iterate over the XML tree to extract the data.
- Use XPath API to extract the data.

We now look at tooling and approaches that are available and that would steer the developer towards the final implementation. The process of weight-
ing individual options would be described for both performance and non-
performance related characteristics of the solution.

We split the reasoning about the options into four big groups, ordered
by how the developer advances from design over implementation to deploy-
ment. In Section 1.2 we will describe challenges related to performance doc-
umentation; in Section 1.3 we discuss how to make software testable from
performance point of view. We dedicate the whole Section 1.4 to the issues
of automated regression detection as identifying a performance regression
usually requires more complex analysis than identification of a functional
bug. Section 1.5 is dedicated to the challenges of continuous software moni-
toring in the production environment. As a standalone topic, in Section 1.6
we discuss the specialities of performance data comparison.

In each section we first describe what tools and approaches are com-
monly available and used for non-performance characteristics (correctness,
maintainability, etc.), describe performance-related challenges and gaps and
also envision how the ideal state would look like. We also discuss potential
connections between the tools and the individual quality characteristics in
ISO/IEC 25010 [45], summarized in Table 1.1. While our main focus is on
the Performance efficiency characteristics, we believe a somewhat broader look
on the possible relationships between the discussed approaches and gener-
ally accepted software quality characteristics is useful.

We note that there are many aspects that would affect the decision in our
example we described above. Some of them the developer cannot influence
(licensing etc.) and we silently ignore these. We assume that responsible
developer would construct a software in such way to optimize the code for
small resource consumption while also balancing other needs such as main-
tainability of the code. We acknowledge that it is up to the developer to
balance these decisions and that the solution is often a compromise.

1.2 Software (Code) Documentation

One of the aspects that would steer the developer from our example is the
available documentation. The developer would not use a library that is
poorly documented. This includes a missing API documentation but also
missing examples or deployment notes (e.g. Maven snippets to include). Similarly, the developer would need to add documentation of the new fea-
ture that is being implemented.

Most tools related to API documentation adhere to principles described
by Kramer [52] that prefer to keep documentation close to the source code
to keep it up-to-date. Javadoc [72], Doxygen [53] or phpDox [6] are typical
examples of extractors of in-code documentation. And some managed envi-
ronments (e.g. Python) even make comments available at runtime, subject to
reflection operations.

Challenges Information about performance of individual operations comes from two
main sources: previous experience and documentation. While previous ex-
perience is a valuable asset it may require years to achieve. Beginners need
Table 1.1: Main software quality characteristics of ISO/IEC 25010 [45]. The word *product* refers to a whole system or a single component; by *environment* is meant both hardware and software environment.

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional suitability</td>
<td>Degree to which a product provides functions that meet stated and implied needs.</td>
</tr>
<tr>
<td>Performance efficiency</td>
<td>Represents the performance relative to the amount of resources used.</td>
</tr>
<tr>
<td>Compatibility</td>
<td>Degree to which a product can exchange information with other products, and/or perform its required functions, while sharing the same environment.</td>
</tr>
<tr>
<td>Usability</td>
<td>Degree to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction.</td>
</tr>
<tr>
<td>Reliability</td>
<td>Degree to which a product performs specified functions under stated conditions for a specified period of time.</td>
</tr>
<tr>
<td>Security</td>
<td>Degree to which a product protects information and data so that persons or other products have the appropriate degree of data access.</td>
</tr>
<tr>
<td>Maintainability</td>
<td>Degree of effectiveness/efficiency with which a product can be modified to improve it, correct it or adapt it to changes in environment, and in requirements.</td>
</tr>
<tr>
<td>Portability</td>
<td>Degree of effectiveness/efficiency with which a product can be transferred from one operational or usage environment to another.</td>
</tr>
</tbody>
</table>

more guidance. Performance documentation is something that can be available all the time and, with proper information, can steer developers towards more efficient solutions.

In our example with extraction of data from XML files the developer needs to decide which library and which method is the fastest. If the libraries would be accompanied by a performance documentation, for example stating that “parsing XML document with 500 elements into a DOM tree takes 1600 ms”, filtering-out slow libraries would be simple. Furthermore, if individual methods would be annotated with concrete numbers, the developer would immediately see that using XPath is slower than manual iteration [41], and would be able to take this information into account.

Availability of API documentation improves code maintainability (the analysability subcharacteristics as documentation is close to the code), usability (the learnability subcharacteristics as the intent of function is clearly stated) and partially also functional suitability (the completeness subcharacteristics as it may simplify testing against requirements specification).

Performance documentation improves the learnability and the analysability subcharacteristics too because it states the expected workload and corre-

ISO/IEC 25010
sponding performance. Indirectly, such documentation improves also performance efficiency as the requirements are clearly stated close to the code.

**Ideal State**

A perfect performance documentation would be ubiquitous and context sensitive – the developer would hover the mouse over a function call in IDE and together with API documentation would see how long the method takes or how much memory it needs. For his own methods, IDE would compute their execution times from the number of calls and their length, giving the developer information about performance even before the code would be compiled and executed for the first time. However, such permeating performance documentation is beyond reach for practical reasons.

There are several contending aspects. With more information available, the developer can make better decisions; but too much information can overload the developer. With more knowledge about the expected workload, the modeling of the performance will be more accurate; but describing the workload can be time-expensive or may not be known in advance. And as we already mentioned, performance is not the only concern the developer has to take care of.

### 1.3 Build-Time Testing

Availability of (functional) tests is another aspect that affects what library the developer would choose. The presence of such tests usually indicates that the developers have not considered the library as a *write once, use once* project. Similarly, the developers would probably add their own suites to test that the code works as expected.

Depending on the development process used, the developer may even start with writing the tests firsts to better understand the task itself [5]. The advantages of test-driven development were already empirically proved [63, 28] and could be considered fundamentals of any agile development process. The developer can use plethora of tools for testing the code. JUnit [47] is a prominent example of a unit-testing framework. On top of JUnit the developer can use Cobertura [19] to assess how much of the code is actually covered by the tests.

The importance of the tests is also captured in ISO/IEC 25010 [45] that considers testability – “degree of effectiveness with which test criteria can be established for a product and tests can be performed to determine whether those criteria have been met” – as a separate characteristics of software quality.

**Challenges**

We can safely assume that developers do not want to make new versions of software slower. To check that, the software needs to be tested and the results evaluated. Optimally, testing is fully automated and tests are executed at each level (i.e. unit, system, . . . ) to catch errors as early as possible. To accomplish that we need to make software components testable and accompany them with suitable performance tests.

In our example with XML, we assume the developer already selected one particular library that seemed to be the fastest based on available doc-
umentation. The developer also prepares a performance test. If the results would be unconvincing, it would be possible to change the implementation and re-run the test to have immediate comparison. Such performance test also serves as a documentation: the developer codified what is considered a typical workload.

We already mentioned that testability is a separate subcharacteristics in ISO/IEC 25010. But (functional) tests improve other characteristics too. Obviously the functional correctness as that is what the tests actually check. If we consider tests injecting faults we can improve fault tolerance and recoverability too (subcharacteristics of reliability characteristics). And indirectly tests also improve maintainability characteristics; for example, the code is more decoupled (as we want to test in isolation) or the tests serve as a documentation of a typical usage.

Obviously, performance tests improve the performance efficiency characteristics as it is possible to automatically check it. Maintainability is also indirectly improved, generally for the same reasons as with functional tests.

In a perfect world there would be a performance test for every single method of every class, for each interface and connection point of a component, etc. Ideally the tests would be automatically generated and would measure the typical workload as well as corner cases. Practically that is not possible and having too many performance tests can be a hindrance rather than an advantage.

We intuitively expect that more important things are tested more thoroughly. From that perspective, having a performance test for everything (consider a battery of performance tests for a simple getter that is rarely needed as an example of wasted effort) obscures the difference between important and less-important functionality. On the other hand, if only several methods are accompanied by a performance test it sends a clear message: the developer considered these methods to be important performance-wise.

1.4 Automated Regression Detection

While in the previous section we focused on mere existence of tests we should also take into account how and when they are executed. Not long ago, the use of continuous-integration service such as Jenkins \[46\] was considered a hallmark of professionals. Today, widespread cloud-based solutions such as Travis \[91\] allow that even one-man projects can afford to run tests on each commit. While having a green icon “build passing” is becoming a norm on open-source software hubs such as GitHub, agile development processes are pushing the continuous testing approach even further. DevOps practices of infrastructure-as-a-code \[44, 65\] require that even deployment has to be fully automated to allow completely unattended release of a new version. In extreme cases, software is released several times per day.
Such automation in the release procedure would not be possible without fully automated tests that push the software to next stage (e.g. deployment) only when all tests are passing. Because the development, testing and deployment are tightly bound together, the developer is notified of the issue virtually immediately and can fix it while still having the mental picture of the code in mind. The importance of fixing bugs early to reduce costs is well-known \cite{63, 28, 7, 97} but DevOps movement brought it to the extreme.

### Challenges

If we return to our XML example, the developer has already implemented the solution and accompanied it by a test. The test becomes a part of a test suite of low-level (wrt. size of tested code) performance tests. First question is, where such test would be executed?

When the application is running on a homogeneous grid, executing the test on single platform would be sufficient. For application targeting more heterogeneous environment (e.g. end-user installations) we would like to test on multiple platforms.

Next thing that needs to be considered is the amount of time the test needs to be executed. Managed environments may require several minutes to properly “warm up” before reasonable measurements can be done. This time cannot be easily shortened and running performance tests in parallel has its own caveats too \cite{2}. And typically one run is not enough to reliably quantify variability of the data \cite{14}.

Considering three restarts, each three minutes long and 10 performance tests we would need an hour and half only to execute the tests. Even for mid-size projects this number could mean that it would not be possible to run every test on every commit and smarter scheduling of the tests would be needed.

Another challenge of its own is how to decide that the two sets of measurements (e.g. of current and of previous revision) are the same, i.e. that no regression occurred. We discuss this in more detail in Section 1.6.

### ISO/IEC 25010

Continuous-integration alone does not affect software quality. But its presence forces the developers to think more (and earlier) about the quality. In this sense, automation of performance and functional tests improve the functional suitability and performance efficiency characteristics as it reduces the window of time when the software is “broken” and the developers are not aware of it.

### Ideal State

Fully automated and integrated regression testing platform would only need the implementation of the performance tests. The tests would be executed, evaluated and issues would be reported. The platform could offer possible solutions or offer an annotated trace of execution flow leading to the regression. The developer would know which commit caused the regression and which components contributed the most to the performance degradation.

Because performance tests generally take more time than functional ones, the platform would also need to schedule the tests according to their priority. The priority would be inferred from previous runs – tests that are known to
show more significant differences could be run earlier to shorten the feedback time.

1.5 Runtime Software Monitoring

Testing in isolation has many advantages – usually the setup and configuration is simple and the execution time is short. Collecting enough data for evaluation could be matter of minutes even for rare operations that would require hours of execution in production environment. However, testing in isolation means that we intentionally ignore any complex interactions between components. That could be remedied by more complex benchmarks but they have issues of their own as happened with SPECjbb2013 [81]. Furthermore, it may not be even possible to test open systems in isolation and some tests can be executed only after deployment.

DevOps principles embraced the approach where “everything is a production” and prefer short release cycle, usually combined with Canary testing [69] on production over thorough tests delaying the deployment on pre-production environment. Supporting this paradigm are full-stack monitoring solutions such as Kieker [36] or Dynatrace [24]. Specialized frameworks such as SystemTap [20] can be used for finer-grained monitoring.

In our running example, the developer’s code is running on production. To collect performance data, the developer also configures the monitoring framework to measure the new code too.

The first challenge is how to compare the data from production with the data from tests to check that the performance test is realistic enough. First of all, the performance data from production environment has to be classified, for example by size. Then it would be possible to compare them with data from tests executing similarly-sized workload (e.g. number of nodes in the XML document). Comparing the data sets is discussed in Section 1.6 in more detail.

When an issue is found in the deployed code, the developer/operator needs to find the root cause. That may require collecting more information without interfering with the running system. The monitoring frameworks supports turning on/off the measurement at runtime but we need to consider what would be the overhead of the monitoring. While it is technically not a problem to measure virtually every instruction, the slowdown shall not influence the availability and reliability of the system.

The presence of runtime monitoring features improves almost all of the software quality characteristics of ISO/IEC 25010, though often indirectly. Analysability (subcharacteristics of maintainability) is improved as the running system is monitored and the operator sees actual interactions. The monitoring also collects data that can be used to better assess usability (e.g. learnability or operability).

Performance efficiency can be actually quantified as there are data from the running system. Reliability of the system can be assessed better too.
Analysability is also improved from the performance point of view. Indirectly – in connection with Canary testing – portability and usability characteristics are improved too as it is possible to compare what works better at runtime.

**Ideal State**

Ideally, every component would be fully monitored – all possible metrics ranging from execution time or memory consumption to length of disk seeks would be collected – yet there would be no overhead to the running application. While it is practically possible to collect this kind of information, the overhead is rarely negligible. Thus the monitoring system should inform the user what is the overhead of the monitoring (wrt. the whole system) and also adjust the reported values to remove the bias caused by the monitoring itself.

On the development side, this information would become part of the documentation to be available when needed; but also not overloading the developer with unasked-for information.

### 1.6 Robust Data Comparison

As we already mentioned earlier in Section 1.4 an inseparable part of the automated testing process is the actual detection that there was some regression. For functional testing with equality assertions this is trivial – failed assertion means a failed test, a regression. Testing of user experience or performance is more fuzzy as it is often not possible to provide 100% confident answers [5, 48]. Therefore we would like to quantify the issue in some way to allow the developer focus on most critical regressions first.

In this sense, we have performance data of two versions and we need to decide whether there is a difference and quantify the difference. The versions can be two consecutive revisions, alternative implementations etc. Here we focus on “dealing with the numbers” part regardless how we obtained the data.

**Challenges**

Returning to our XML example for the last time, the testing infrastructure runs the test regularly. To better assess the variability in the performance, the test is executed several times. Therefore, the comparison is between two vectors of performance measurements. That is a typical input of traditional statistical tests (e.g. Welch’s test). But since there is generally no guarantee of the distribution of the samples, non-parametric tests – that are generally less powerful – has to be used.

Furthermore, if the test is restarted (e.g. restart the JVM) we obtain a different set of data. Individual restarts of the whole test then capture the variability of the platform itself. Instead of comparing two vectors, the comparison is between two sets of vectors. Merging the data in each set (i.e. considering a restart as a continuation of previous run) loses information about one kind of variability. Therefore, more sophisticated statistical methods has to be devised.
Quantifying the regression has its own pitfalls too. Trivial ratio of means hides the differences of variability but is easy to compute. Using medians might be less intuitive although it is more stable (single outlier can shift the arithmetic mean a lot). Often it is even difficult to describe – at least, semi-formally – required properties of a ranking algorithm.

Our requirements for a robust and reliable comparison are seemingly intuitive but in practice they can be contradictory.

Our first requirement is that the comparison must be robust against noise – i.e. ignore small differences – caused by outside factors such as OS scheduler, garbage collection cycles dynamic frequency scaling of the processor etc. Second requirement is that the comparison must detect even a small differences reliably. These two requirements are competing over the size of “small” difference yet both are reasonable and a good method must take them into account.

Our last requirement takes into account ranking of the results – we need to be able to distinguish small and big differences to allow developers prioritize their work.
In the previous chapter we provided an overview of different aspects influencing performance-aware development. But the broadness of the topic forces us to focus in this work on selected subtopics only as described in this chapter.

This work focuses on the software performance engineering aspects of the performance-aware development. With focus on the software engineering aspects of the topic and the summary provided in Chapter \[1.1\] we can expand our initial goal as follows.

The primary goal is to improve developers awareness of performance of the software they are working on. The following list covers key areas of our interest.

\[(G_1)\] Design method for writing performance-testable code where performance tests will be integrated as a natural part of the development cycle.

\[(G_2)\] Design method for robust detection of performance regressions.

\[(G_3)\] Design method for documenting performance of individual components that provides platform-relevant information and is readily available.

\[(G_4)\] Explore limits of runtime performance monitoring of individual components in the terms of precision and overhead.

Most of the topics described earlier in Chapter \[1.1\] are environment-independent (hardware, operating system, ...) We decided to use Java as the evaluation platform. Also, all examples in this work that require source code snippets would be in Java.

As Java ecosystem has the following properties we believe it is a reasonable choice for evaluation in the domain of software performance.

- The language is widely used and the runtime environment is supported by major operating systems.
– The execution platform (JVM) supports also dynamic languages and thus the results are more easily applicable to non-Java software too.

– The environment heavily uses just-in-time compilation or garbage collection. This ensures that we exercise more the robustness of our evaluation because of higher volatility of the performance data.

– Java has a well-defined interface for invoking operations not directly available in the VM (JNI and JVMTI) that might be required for certain types of analysis/evaluation.
In this chapter we summarize contribution that is in detail presented in Part II, Collection of Papers.

This chapter provides a unifying view on the publication contents without going into technical detail. We omit or simplify technical details when they are not crucial for understanding the core concepts. This is especially true for several source code examples and schema diagrams. The details can be found either in the text of the publications or directly in the implementation of the relevant tools.

The chapter is structured as follows. First, we introduce performance unit testing as an approach for creating performance-testable components in Section 3.1. Such components can be easily measured and the measurements can be used to detect performance regressions – we describe several methods for evaluating the measurements in Section 3.2. In Section 3.3, we show our approach for documenting performance of individual components as another aspect of writing testable code. Finally, in Section 3.4, we analyze the actual overhead of collecting performance measurements to understand the practical limits of the precision we can obtain when we are monitoring performance in live systems.

### 3.1 Performance Unit Testing

Systematic performance testing can help create software with high-quality performance. While load testing and other types of system-wide performance tests are well established approaches, they happen too late to benefit from the “test early, test often” motto of test-driven development. Therefore we want to test individual components even before they are integrated and detect regressions much earlier in the development lifecycle.
We are motivated by a large body of studies that document the costs and benefits of functional tests \cite{63,28,7,97}. Our initial experiments and surveys \cite{28,85} suggest our work is heading in a promising direction. Because of the similarity between our approach and (functional) unit testing, we decided to use the term performance unit testing. These tests focus on small parts of the system and therefore are relatively short (compared to load tests, for example). Therefore they can be executed in isolation and early in the development lifecycle.

In the following sections we describe two important aspects of performance unit testing contributed by this thesis – how to reasonably capture the performance requirements of individual components, and how to integrate the tests in the source code.

### 3.1.1 Stochastic Performance Logic

One of the most important aspects of any automated testing approach is capturing the test condition in a concise, elegant manner. Clearly stated – and in a machine readable form – testing conditions represent a form of requirement specification that helps prevent ambiguities concerning correctness of both the tested code and the test condition itself.

Functional tests often use assertions that check the correctness by comparing a result computed by the test with a (statically) defined expected output. Requirements for performance test are similar, the comparison would deal with performance metrics rather than computed results, for example by asserting that an operation performed by the test has to complete within some time limit:\footnote{We will use “lower is better” semantics in the following description and use wall-clock time as a typical metric.}

$$\text{metric}_{\text{operation}} \leq \text{limit}$$ \hspace{2cm} (3.1)

The idea of a simple performance comparison encounters multiple practical challenges. One is the question of the exact value for \textit{limit}. For a high-level operation, \textit{limit} can be derived from software requirements specification, often backed by user-behaviour studies (e.g. acceptable waiting times) or physical principles (e.g. sensor limitations). Quantifying the value for lower-level operations is much more difficult. As an example, let us consider the limit for sorting 100 integers. The actual \textit{limit} depends on the hardware we use – calibrating against low-end cell phone would make the test too tolerant on a high-end mainframe and vice versa. Yet, a test checking the speed of a sorting function makes sense.

As an alternative to fixed limits, we contribute an approach to specify the test condition as a comparison of performance between similar routines, where one serves as a baseline. Our initial formula \cite{3.1} shall be thus transformed into

$$\text{metric}_{\text{operation}} \leq \text{metric}_{\text{baseline \ operation}}$$ \hspace{2cm} (3.2)
We have formalized this approach in Stochastic Performance Logic (SPL) – a many-sorted first-order logic with operators for comparing performance of multiple functions against each other.

As a more elaborate example we consider a specialized sorting method that shall be faster on small arrays of integers than the standard library sorting method. This assertion can be captured in our formalism in the following manner:

$$\forall n \in \{10, 20, 30, \ldots, 100\} : \text{Perf}_{\text{specializedSort}}(n) \leq \text{Perf}_{\text{librarySort}}(n)$$

where $\text{Perf}$ is a random variable representing the performance of a given component. Practically, it is a set of measurements collected during test execution.

It may seem that by using a baseline operation for comparison we have not solved the issue of finding the limit – now we need to find a proper baseline operation. We believe that would be usually simpler and furthermore there would be always one obvious candidate: previous version of the same operation (e.g. previous commit). That also clearly captures an apparent requirement: the software could not be slower in new versions.

Note that our notation intentionally works with specific problem sizes and avoids $O$ and similar asymptotic notations. The $O$ notation is useful for specification of algorithmic complexity, however it is too generic for capturing assumptions about the detailed performance of a concrete implementation. As another distinction from $O$, SPL can explicitly state the multiplication constants to accurately quantify the performance relationships. Constants are not present in $O$ formulas.

SPL is an axiomatic logic that serves as a syntactic vehicle for describing the test conditions. The logic has multiple interpretations defining how each SPL formula is evaluated. These interpretations affect the sensitivity of the assertion: an overview of several interpretations is described in Section 3.2.

3.1.2 Deployment in Java

As our next contribution, we develop support for deploying SPL-based performance unit tests in Java. The overall structure of a performance unit test in Java is depicted in Figure 3.1. We provide more details on the structure while discussing function selection, workload selection and build process integration.

To employ SPL-based performance unit tests, the developer first has to decide what functions have to be tested and write SPL formulas for them. This decision is associated with the perception of test coverage. Although having all functions covered with performance tests may be appealing, it is often impractical – writing a good performance test is typically more difficult than writing a good functional test and takes more time. Testing some functions in isolation may also make very little sense, for example, when the tested function is very short (and therefore may be inlined and otherwise
Listing 3.1: Performance unit test in Java. On line 1 we attach the @SPL annotation to the tested method to keep the performance assertion close to the implementation.

The workload generator code starts on line 7. On line 10 we prepare the array to be sorted. We use the convention of the Java reflection API and wrap the array into Object[] on line 15. For very short operations, the workload generator may choose to provide multiple input arguments to be used inside single measurement. This is why the parameters are wrapped in an iterable on line 17.

```java
1   @SPL("for i in { 10, 20, 30, 40, 50, 60, 70, 80, 90, 100 }"
2       + "SELF[getData](i) <= java.util.Arrays#sort[getData](i)"
3   public static void specializedSort(long[] data) {
4       // Special sorting implementation optimized for small inputs.
5   }
6
7   public static Iterable<Object[]> getData(int size) {
8       Random rand = new Random();
9
10      long[] data = new long[size];
11      for (int j = 0; j < size; j++) {
12          data[j] = rand.nextLong();
13      }
14
15      Object[] params = new Object[] { data };
16
17      ArrayList<Object[]> result = new ArrayList(1);
18      result.add(params);
19
20      return result;
21   }
```

optimized in a way that distorts the test) or rarely used (and therefore of little importance from the application perspective). Here, we leave it to the developer to decide which components require performance tests.

In order to measure the tested component, we need to execute it with a relevant workload – the second step in writing a performance test is therefore the implementation of a workload generator, responsible for preparing data that can be fed to the tested component. We can parametrize the workload generator to e.g. provide data of different sizes while reusing the same source code.

A simplified test using the assertion on specialized sort from Eq. 3.3 can be seen in Listing 3.1. It shows both the use of the @SPL annotation (with the test condition) as well as the implementation of a workload generator.

Finally, performance unit testing must be integrated into the build process. We contribute the design and implementation of a tool that locates the test annotations, executes the workloads and measures the tested methods, with results provided in a form similar to JUnit test reports. When regression testing is required, the tool can also access the project version control system [37].
Figure 3.1: Performance unit test structure. From the performance assertion formula we extract the name of the workload generator and its parameters and the tested method. The configured workload generator prepares the workload that is fed to the tested method and the method is measured. Once enough data are collected, the test condition can be evaluated and the results are reported to the user.

### 3.1.3 Framework Evaluation

To validate our contribution in a practical context, we have incorporated performance tests into an existing open-source project. The performance requirements were based on comments in the source code and on commit messages in the versioning system. The performance tests were applied retroactively to multiple versions of the project, which allowed us to approximate performance testing during project development.

Our project of choice was JDOM [43], a library “for accessing, manipulating, and outputting XML data from Java code”. The library has reasonable size – about 15 000 LOC – and over 1 500 commits spread across more than 14 years of development. As such, the library represents a medium-sized open-source project where performance is of evident concern.

For regression testing, we have selected 46 versions that – according to the commit messages – changed performance significantly. Our validation focused on one high-level component – the SAX builder (XML parser) – and one low-level component – the Verifier class (utility class). For these, we have created over 50 performance unit tests that codified the developer assumptions as stated in the commit messages. We have collected results on three different hardware platforms, described in detail in [38], and for certain revisions we have even partially emulated older software environments. Besides demonstrating the functionality of our framework, the results show that in approximately one tenth of the test cases, the developer assumptions about performance were wrong, i.e. commits supposed to improve performance actually caused regressions.
Figure 3.2: Performance unit test portability. The histogram shows portability metric $PM$ of all the tests across three different hardware platforms. The metric of a perfectly portable test (workload) has value of 1.0 therefore ideally we should see a single bar at 1.0 in the histogram.

As another practical outcome of the validation experiments, we conclude that exhaustive testing of all methods with the goal of achieving full code coverage – akin to what functional testing might aim for – is not feasible. Our performance tests covered about one fifth of the library and the duration of the performance tests for single commit was still approximately 30 min. Increasing the coverage would lead to higher test duration and for larger projects, the duration could easily increase past typical commit interval. $^2$

Finally, we have also used the experiment to assess performance test portability. Formal definitions are omitted here but the reader can find them in $[38]$. We define a portability metric $PM$, which indicates how much the performance of a particular pair of methods – forming the performance comparison (Eq. 3.2) – changes between platforms. Values close to 1 indicates that the test is portable, i.e. after running the test on a different platform the test result remained the same (and with similar confidence).

The values of the portability metric, displayed on Figure 3.2, are mostly close to 1, suggesting that performance assertions based on relative comparisons (recall Eq. 3.2) are indeed portable. For most of the tests, the portability metric value was smaller than 1.25, i.e. the relative differences were mostly smaller than 25%, which is an excellent result given that we have used hardware platforms with manufacturing dates 5 years apart.

Details of our experiment are described in $[38]$. Robustness and reliability of our evaluation is described in Section 3.2.

### 3.2 Robust Regression Detection

So far, we have focused on how our contribution facilitates the construction and deployment of performance unit tests. Next, we introduce the com-
putations required for carrying out the performance comparison operations described by SPL formulas. The comparisons are performed on data that is naturally distorted by noise and other measurement artifacts, we therefore focus on computations that take these into account to deliver robust results.

3.2.1 SPL Interpretations

When using the $\leq$ operator in Eq. 3.3 in Section 3.1 for performance comparison, we have suggested that we use the “lower is better” semantics. Formally, we define the semantics by providing an interpretation for the SPL logic, which in this case takes the form of a formula for computing the Boolean result of the operator from the measurement data. To reflect different scenarios, we provide four different interpretations which we discuss individually. Mode details of the interpretations together with proofs of their conformance with the axiomatic base of SPL are described in [14, 92].

Our interpretations assume measurement data describe a steady state of the application, represented as observations of a random variable and denoted $\text{Perf}$. Detection and handling of initial transient conditions (e.g. program load from disk or just-in-time compilation) is covered in more detail in [14] and [39].

We first present a theoretical interpretation that assumes that we know the expected value of $\text{Perf}$ of both sides of the comparison. Comparing performance measurements $\text{Perf}_A$ and $\text{Perf}_B$ is then performed simply by comparing the corresponding expected values.

This interpretation is not intended for practical use, because the expected value of a random variable is a theoretical concept that measurement can only estimate. However, it serves to provide an intuitive understanding of the comparison operator behavior.

In Quick Mean-value interpretation we replace the comparison of expected values with an evaluation of a statistical test. In layman terms, we evaluate the comparison as true when we do not observe sufficient evidence for the opposite verdict.

Let $P^i_X$ denote $i$th observed performance of component $X$; let $\alpha$ be a fixed significance level. Note that we ignore restarts and merge all samples into vector $P_X$.

Then $\text{Perf}_A \leq \text{Perf}_B$ iff we cannot reject the null hypothesis

$$H_0 : \mathbb{E}(P^i_A) \leq \mathbb{E}(P^j_B) \quad (3.4)$$

with a one-sided Welch’s $t$ test at significance level $\alpha$ with observations $P^i_A$ and $P^j_B$.

Note that Welch’s test assumes several properties of the data it works with. Generally, data collected for performance measurement violate these assumptions. Later in Section 3.2.2 we show that certain violations can be practically tolerated.
So far, the interpretations have not distinguished data collected across measurement restarts. In practice, it is necessary to restart the measurement process several times to assess variability between restarts. This variability is typically caused by the volatility of the environment (e.g. randomized memory layout or practical non-determinism of just-in-time compilation).

To account for both variability between measurements separated by a restart and variability inside measurement sequences without a restart, we have contributed a new statistical test based on Welch’s t-test. This test, which accounts for restarts explicitly, is described in detail in \[14\]. Practical experiments, presented in Section 3.2.2, indicate that the Parametric Mean-value interpretation is more robust in environments where restarts introduce significant additional variability, such as in Java.

In their statistical derivation, our earlier interpretations rely on (possibly asymptotic) normality of the observations. To remove this shortcoming, we also contribute the Non-parametric Mean-value interpretation that constructs an empirical probability distribution function of the sample means, rather than approximating it with a Gaussian distribution. The probability distribution function is constructed using a bootstrap procedure with a Monte-Carlo simulation \[79\], making the interpretation more computationally intensive, but less sensitive to violations of the normality assumption.

In more detail, deciding whether \(\text{Perf}_A \leq \text{Perf}_B\) involves constructing an empirical distribution function for the difference of (normalized) bootstrapped means of \(\text{Perf}_A\) and \(\text{Perf}_B\). The actual difference of means is compared with the quantiles of this distribution at given significance level to determine whether this difference provides strong evidence for rejecting \(\text{Perf}_A \leq \text{Perf}_B\). Details and proof of correctness is described in \[14\]. Evaluation is again presented in Section 3.2.2.

### 3.2.2 Evaluation – Interpretation Sensitivity

Our evaluation of the interpretations from previous sections focuses on two aspects. One is the ability to detect an actual change – i.e. how big a difference is detected. The other is the ability to disregard incidental changes – i.e. how often a false difference is reported due to random noise, restarts, environment, etc. In addition, we also look at how the Welch’s t-test behaves with measurements that violate its assumptions. We evaluate all three interpretations that are designed to work with actual measurements, i.e. the Quick, Parametric and Non-parametric mean value interpretations.

To evaluate the robustness of the interpretations against incidental changes, we have collected a high number of measurements of the same code in the same conditions – over a hundred of restarts, with several thousands of measurements between each restart, applying a long-enough warm-up to ensure we collect as stable data as possible. Then, we have used the interpretations to compare random subsets of the measurements and record any difference reported. By definition, the measurements we have collected
Figure 3.3: Probability of reporting an incidental difference with Quick mean-value interpretation. The histograms show how likely is it that the Quick mean-value interpretation would report a difference between data coming from restarts of the same workload (i.e. probability of false alarm). Using higher tolerance increases the robustness against false positives but obviously also reduces the sensitivity for detecting a regression in the data. The plot is based on more than 100 different workloads; with setting $\alpha = 5\%$.

should be “the same” – since we measure the same code in the same conditions, any observed difference is incidental. The relative count of reported differences therefore describes the robustness of the interpretation against incidental changes.

In more detail, each step of the evaluation consisted of (1) selecting two random subsets of all the data we had and (2) comparing these subsets against each other using the SPL interpretation of choice. Each step thus resembles one hypothetical performance test execution. By repeating these steps many times (i.e. doing a Monte-Carlo simulation) we can assess how often the interpretation would decide incorrectly (i.e. when the interpretation marked the two subsets as “differing”). In the following, we call this the probability of wrong reject.

For the Quick mean-value interpretation we extended the above procedure with a “tolerant” variant where, instead of testing whether $\text{Perf}_A = \text{Perf}_B$, we test that $\text{Perf}_A \in \text{Perf}_B \pm 5\%$. This is done by multiplying all measurements on one side of the comparison by 0.05 and then performing two one-sided tests.

Complete results of the evaluation, where we evaluate all three interpretations with various number of restarts and significance levels $\alpha \in \{5\%, 0.1\%\}$, are available in [14]. Here, Figure 3.3 shows the probability of wrong re-
Figure 3.4: Probability of reporting an incidental difference with Parametric and Non-parametric mean-value interpretations
The histograms show how likely is it that the Parametric and Non-parametric mean-value interpretations would report a difference between data coming from restarts of the same workload (i.e. probability of false alarm). The plot uses data with $\alpha = 5\%$.

ject for the Quick mean-value interpretation with different tolerance levels. The first (left-most) column demonstrates that as-is, this interpretation flags incidental differences quite often, which makes it unsuitable for our measurements. From the remaining columns of the figure we can conclude that for practical purposes multiple runs and at least 1% tolerance is needed for reasonable results with the Quick mean-value interpretation.

Figure 3.4 summarizes results of the Parametric and Non-parametric mean-value interpretations. These interpretations have similar probabilities of wrong reject as the Quick mean-value interpretation with 5% tolerance level. They are more robust against marking an incidental change but they also require more data to be collected.

To summarize, the Quick mean-value interpretation should be used with higher level of tolerance when one or only few restarts are available. Once enough data is collected we should use the Parametric and Non-parametric mean-value interpretations. Because the Parametric and Non-parametric mean-value interpretations provide similar probabilities of wrong rejects, we would typically prefer the Parametric mean-value interpretation as the Non-parametric one is much more computationally demanding.
In the second experiment we evaluate how sensitive the interpretations are to actual changes in performance. To provide measurements with realistic changes in performance, we have collected measurements from a single data processing benchmark of linear complexity, which we have run with different workload sizes. As a drawback, the change in workload size does not necessarily correspond to the same change in performance (although the complexity is linear, the dependency of performance on workload size may not be strictly linear). On the other hand, we believe this approach provides more realistic data than e.g. synthetic multiplication or measurements from single workload size by a small factor (which would also provide us with data that, technically, represent changing performance).

Our experiment again uses the Monte-Carlo simulation, this time to select random subsets from measurements with different workload sizes. We use one-sided comparison variants of the SPL interpretations to decide which subset is “faster”. We perform the comparison in both directions (i.e. \( \text{Perf}_A \leq \text{Perf}_B \) and \( \text{Perf}_B \leq \text{Perf}_A \)) to derive ratios of wrong rejects and correct rejects.

Assuming \( \text{Perf}_A \geq \text{Perf}_B \) (i.e. \( B \) refers to bigger workload that is, by definition, slower), wrong reject means that the statistical test rejects \( H_0 : B \geq A \), correct reject means that the statistical test rejects \( H_0 : B \leq A \).

We repeat this procedure for the three interpretations, various number of runs and significance level \( \alpha \in \{5\%, 0.1\%\} \).

The results of the evaluation are shown in Figure 3.5. Each subgraph shows the ratio of both wrong and correct rejects for a single interpretation. The experiment shows the Quick mean-value interpretation identifies performance differences early (i.e. little difference between workload sizes) but also has high rate of wrong rejects – from practical perspective the interpretation is rather unstable. Again, Parametric and Non-parametric mean-value interpretations behave similarly – they almost never wrongly reject and with high number of runs they can detect changes smaller than 1%.

Finally, we have also evaluated how robust the Quick mean-value interpretation is when Welch’s t-test assumptions are not met.

We have picked three methods from our JDOM experiment that produce different shapes of the execution time distribution functions. Shown in Figure 3.6 the distributions are unimodal with a tail, bimodal distribution and a quadrimodal one.

To test the robustness of the interpretation we collected data with multiple restarts of each method and computed the ratios of wrong and correct rejects for each method. Again, we use the Monte-Carlo simulation, but rather than changing the workload size as in the previous experiment, we introduce performance changes by multiplying individual samples by a constant close to one (e.g. 101%). Random subsets from the original data should ideally test as smaller than random subset of the modified data.

Results of the evaluation are in Figure 3.7. We can see that with large number of observations the likelihood of wrong reject is very low. With increasing number of observations the number of correct rejects is increasing for all tested distributions. The plots show results for difference of 1%. For bigger difference (e.g. 10%) the number of observations needed for high
Figure 3.5: Sensitivity to input size change. The plots show how quickly the interpretations are able to detect a change in performance. Horizontal axis shows relative input size difference, vertical axis represents probability. Wrong rejects represent probability of rejecting a correct relation (ideal value is 0%) correct rejects represent probability of rejecting an incorrect relation (ideally at 100%). Usually we strive for low number of false alarms as these had to be verified by other means, typically manually. $\alpha$ was set to 5%.

likelihood of correct verdict is much lower while the chance of wrong reject is practically nil.

Our results indicate that Welch’s t-test is sufficiently robust against normality violations when we have enough data. What exactly is “enough data” strongly depends on the properties of the data, which cannot really be derived from the experiment. Because our earlier experiments have shown the other interpretations to be both more robust and more sensitive, use of the Quick-mean value interpretation should remain confined to situations where simple computation is preferred to accurate results.

Summary

To conclude, our experiments show that the interpretations contributed by the thesis are practically viable, achieving a negligible false alarm rate while detecting changes in workload size as low as 1% to 3%, on measurements taken in a standard Java environment. Furthermore, our measurements contribute some insight into the relationship between the achievable
change detection sensitivity and the number of measurements executed, essential for practical deployment of performance tests.

### 3.3 Performance Documentation

Following the introduction of performance unit tests in Section 3.1, we have also explored the possibility of reusing the tests to provide documentation of performance of individual software components. This work is motivated by the need to provide information about performance throughout the software development process – where the performance tests are retrospective in nature, attracting attention only when problems are detected, performance documentation is more interactive, providing performance information on software components during the coding activities.

#### 3.3.1 Reusing Performance Tests

To document performance of a software component, we rely on the fact that a performance unit test of the component implements a workload generator (recall Section 3.1.2) that can drive the component in measurement experiments. We can reuse the workload generators in a development environment that performs measurement experiments on demand, leveraging the fact that they are clearly separated from the methods they test.

To generate performance documentation, we add several annotations to the workload generator. These serve to describe its behavior and explain the meaning of its parameters. An example of an annotated workload generator is available in Listing 3.2. Because the annotations are retained in the

---

**Figure 3.6**: Execution times of three example methods. Notice that the plots are clipped and do not display outliers.
Figure 3.7: Sensitivity to 1% execution time change on the three example methods. See Figure 3.6 for distributions of the measurement samples.

Listing 3.2: Performance documentation in Java. A workload generator accompanied by Java annotations that describe the workload (lines 30 and 32) and bind the documented method with the generator (line 24). A special annotation can be used to list methods whose performance is to be related with the documented one (line 25).

```
22 // class my.pkg.Utils
23
24 @Workload("my.pkg.Utils.getData(int)"")
25 @SeeAlso("java.util.Arrays.sort(long[])"")
26 public static void specializedSort(long[] data) {
27    // Special sorting implementation optimized for small inputs.
28 }
29
30 @Generator("Creates unsorted array of longs with given size")
31 public static Iterable<Object[]> getData(
32    @Param("Array size", min=0) int size
33 ) {
34    Random rand = new Random();
35    long[] data = new long[size];
36    for (int j = 0; j < size; j++) {
37        data[j] = rand.nextLong();
38    }
39
40    Object[] params = new Object[] { data };
41    ArrayList<Object[]> result = new ArrayList(1);
42    result.add(params);
43    return result;
44 }
```
binary form of the generator and available at runtime, the documentation can be part of a binary distribution and can be queried anytime. Another annotation provides for a simple binding between the workload generator and the documented method, useful when the method is not mentioned in any performance assertion.

With the annotated workload generator in place, we have sufficient means to collect information on component performance. More details about the technical side of workload generator reuse are described in [41], here we look next on how to present the information to the developer.

3.3.2 Performance-extended Javadoc

On the Java platform, the standard tool to generate software documentation is Javadoc. We employ the fact that the developers are used to consulting Javadoc in their development environments, and add performance documentation as interactive plots and tables with the performance measurements directly into Javadoc. Modified Javadoc – PerfJavadoc – is shown in Figure 3.8 where the user queries the performance of standard Java class ArrayList.

To use the performance extended documentation, a measurement server is started on the background of the development environment. The server is responsible for measuring the performance of methods that the developer looks up in Javadoc, and for caching the measured data.

The measurement process is initiated by the developer when browsing the Javadoc documentation. The documentation includes a form where the parameters for the workload generator can be adjusted (recall Listing 3.2). Once selected, the information on the method to be measured and the parameters is sent as a request to the measurement server.

In response, the measurement server either measures the requested method or sends back results from the cache when available. The measurement is done in several phases – in the first phase, the measurement server uses reflection to run the workload generator, yielding measurements that are delivered quickly but are possibly less accurate. Later, specialized measurement code is generated on the fly and executed in a separate JVM, providing higher measurement accuracy. On the client side, the measurement display is updated accordingly, yielding some results quickly and gradually refining as more measurements are done.

The schema of the communication between the client and the server is depicted in Figure 3.9: Figure 3.10 illustrates how much data can be available within a specific time slot, i.e. how long the user needs to wait when data is not cached. More details are available in [41], technical description of the current implementation is available in Náplava’s thesis [67].
Figure 3.8: Screenshot of performance-extended Javadoc for the `contains()` method of `ArrayList` class. The user can select the required workload (here a search for element that does not exist in the collection) and size of the collection. The graph depicts mean execution time together with standard deviation. Note that it is possible to also display median or to switch to a tabular view.

### 3.3.3 Evaluation – performance-extended Javadoc

When evaluating the contribution of performance documentation to development of efficient software, we consider two questions. The first question looks at whether a deliberate application of information provided by performance documentation can help speed up an existing software project. The second question is whether a software developer would benefit from having the performance documentation available while writing new software, without incentives to apply the documentation deliberately.

To evaluate whether deliberate use of performance documentation can help change an existing program and improve the overall performance, we use our measurements of the JDOM library from Section 3.1.3 and attempt to optimize other projects that use JDOM. We note that the result is not at all predictable. When consulting the performance documentation, we must implicitly assume that the workload used to derive the documentation is similar enough to the eventual application workload to provide relevant information. Furthermore, performance is not an easily composable property, and even if our changes lead to local performance improvements, it is not
User selected method and workload \( f(w) \)

Browser

\[ \text{GET } f(w) \]

Server

\[ \text{Measure } f(w) \]

\[ \text{(imprecise)} \]

None

Random delay

Display to user

\[ \text{DATA } Q=0 \]

Measure \( f(w) \)

\[ \text{(precise)} \]

\[ \text{GET } f(w) \]

\[ \text{DATA } Q=1 \]

Display to user

\[ \text{DATA } Q=0 \]

\[ \text{DATA } Q=1 \]

\[ \text{DATA } Q=0 \]

\[ \text{DATA } Q=1 \]

**Figure 3.9:** Communication schema between Javadoc and the measurement server. The \( Q \) attribute denotes the precision of the data returned by the server. An example of plots with different \( Q \)s is in Figure 3.10. The current implementation uses more levels of precision (\( Q \) attribute in the DATA answer) and thus the client typically sends more requests until obtaining the most precise result.

**Figure 3.10:** Improving the precision over time. The data displays time needed to sort a Java collection. Each subplot shows how may data points can be measured within specified time. For example, “0.1 s (5)” means that within 0.1 s we measured sorting of 5 different collection sizes. The boxplots show the relation between stability of the data and the total duration of measurement.

Because the time in the first plot is too short for just-in-time compiler to fully optimize the code, the results are not accurate. The subsequent plots do not suffer from this effect and collect many more data points. The plot does not show outliers. The seemingly increasing variance in the last plot is due to drawing too many overlapping boxes rather than error within the data.
Table 3.1: Improvements to the Buildhealth utility. The Repeated line denotes average execution time of multiple invocations inside single JVM (for this case we were also able to measure the effect of caching parsed XPath query). Because Buildhealth can be run as a standalone application or as Ant plugin, we report execution times for both.

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>No XPath</th>
<th>Cached XPath</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repeated</td>
<td>938.3 ms</td>
<td>908.4 ms</td>
<td>929.8 ms</td>
</tr>
<tr>
<td>Ant task</td>
<td>2.23 s</td>
<td>2.16 s</td>
<td>–</td>
</tr>
<tr>
<td>Standalone</td>
<td>2.52 s</td>
<td>2.40 s</td>
<td>–</td>
</tr>
</tbody>
</table>

clear whether and to what degree these will be reflected in the overall performance.

In our modifications of existing projects, we have focused on the iteration across XML node children as one essential JDOM feature. One iteration approach is to use XPath, other options involve several different methods with slightly different semantics and slightly different performance. In particular, unless an XPath expression is reused many times, the XPath approach is roughly an order of magnitude slower \[^41\] than the alternatives.

By browsing the Ohloh project tracking site, we have identified three projects that heavily use JDOM and iterate across XML node children using one of the methods mentioned above. The list of projects was considerably narrowed down by filtering out projects that we were not able to easily build. Also most of the projects use JDOM at startup only to read initial configuration: we have not considered that a reasonable target for our optimizations. We have modified the source code so that the iteration method with the best performance was used. Such change typically affected only several lines of source code. We have then measured the performance of the programs before and after the change.

Our results suggest that even this simple change can improve performance significantly. As an example, we take Buildhealth \[^22\] – a utility for aggregating build results – where XPath was used to iterate through result subtrees in a XML report. We replaced XPath with a simple loop over all children and instantly improved the performance by several percent on a realistic build log. Table 3.1 summarizes the results. Details and results for the remaining two programs are available in \[^41\].

Given the small scope of the changes needed to get the performance improvement, we believe the developers might have used the faster iteration approaches from the very beginning, had they access to the performance documentation. While the performance improvement in a single case is not very high – 5% in the most optimistic scenario we had – a combination of several similar optimizations has the potential to affect the overall application performance significantly. Importantly, with the contributions of this thesis, similar optimizations can be backed by actual measurements provided by a repeatable test, rather than by ad hoc experimentation or educated guesses.
Deliberately consulting the performance documentation with particular optimization in mind is different from simply coding with the documentation available alongside other information sources consulted only as needed. We have therefore also conducted an experiment where the controlled parameter was the availability of performance documentation and observed the impact on the performance of the implemented software.

To set up the experiment, we have asked the students enlisted in the Advanced Java Programming course at our university to finish one assignment relying only on a specialized version of the API documentation provided together with the assignment. Unknown to the students, two versions of the documentation were distributed – one a standard Javadoc, the other an extended Javadoc where some methods (those the students were likely to need in their assignment) were accompanied with information on their performance. To be less conspicuous, we have not used the interactive documentation and instead inserted statically generated content.

The assignment itself was to print links between sections in a DocBook document [95]. Our reference solution had about 200 LOC. After the solutions were submitted, we have asked the students to respond to an additional survey, which helped us collect information that can not be derived from the submitted solutions.

Our study did not make it possible to decide whether any of the two student groups submitted more efficient solutions – the differences were not statistically significant. In particular, there was a surprisingly large variance between the performance delivered by individual solutions. We show that in a sample of similar variance, thousands of student assignments would need to be collected to decide. That is an interesting fact by itself as it suggests that resulting software performance is not managed very well during development (the course is typically taken by students with 2+ years of programming, it is not clear how this applies to more or less experienced developers).

The survey results revealed another interesting phenomenon – the students were often able to state the algorithmic complexity of their solution (e.g. $O(n^3)$), but they were unable to translate this knowledge into reasonable estimates of the actual solution performance. For solutions more complex than $O(n)$, the authors tended to overestimate the actual execution time of their solution, sometimes even by several orders of magnitude. This shows that the theoretical understanding of algorithmic complexity does not translate into practical feeling for actual software performance.

We have repeated the experiments for two more years. We have extended the task description with information about typical input size and emphasized that the solution has to have a reasonable complexity. But the variance we see in the submitted solution hides any performance improvement we could observe; our own experiments have shown that the improvements are in order of several percents.

More details of our experiments are available in [41].
3.4 Analysis of Performance Monitoring Overhead

In this section we describe our analysis of dynamic runtime performance monitoring overhead in Java. By dynamic performance monitoring we mean a solution where the application performance is monitored on demand and the actual measurement probes are dynamically inserted into the application at runtime, as opposed to statically linked at compile time. In Java, dynamic performance monitoring is facilitated by class reloading, where the class definition is changed at runtime – monitoring probes are inserted before measurement and removed once enough data is collected.

We focus on questions related to precision of the measurements and effects on the running application. From practical perspective, we seek the limits of dynamic runtime performance monitoring – e.g. what is the shortest operation we can measure and how long it takes the application to return to a stable state after disruption caused by dynamically configuring the monitoring probes.

Our approach is described in more detail in [39]. Here we focus on three topics that we consider the most interesting. First, we look at the behavior of the just-in-time compiler when the dynamic probes are added, as that typically triggers massive recompilation of already compiled (and running) code. Next, we examine the accuracy of the collected data – i.e. how much we can trust the observed performance to represent what happens inside the application when no monitoring is performed. And finally we study how runtime bytecode modifications affect the steady-state performance of the monitored application.

3.4.1 Framework for Overhead Measurement

To assess the overhead of the dynamic monitoring, we use two independent measurement infrastructures. The first infrastructure is static – the measurement probes are inserted into the application at compilation time. The second infrastructure is dynamic – the probes are inserted (removed) on demand at runtime.

From the perspective of the dynamic infrastructure, the static infrastructure is part of the monitored application. It can therefore provide us with information about baseline performance. The dynamic infrastructure provides us with the “observed” performance as we would see it when using a dynamic monitoring framework.

To collect the data for our analysis, the application is run with the static infrastructure in place. During this run, the dynamic infrastructure is used to monitor individual components, emulating practical use of the dynamic performance monitoring. By comparing the data coming from the static and dynamic infrastructures, we can determine the overhead of the dynamic monitoring and also the “quality” of the collected data.

Figure [3.11] depicts a simplified scheme of the measurement framework. Our technical solution uses AspectJ [25] for the static modification of the
application classes and DiSL \cite{61} for the dynamic code modification. We use a heavily optimized C-agent and JNI for storing the information from the static probes. Dynamic probes are fully implemented in Java but only with functionality required for capturing wall-clock time of the measured component. We strive to have the measurement probes very small, with as little impact on the original code as possible, to assess the accuracy and overhead in the most optimistic scenario.

\subsection*{3.4.2 Evaluation: Overhead of Monitoring SPECjbb2015}

We have run our experiment with the SPECjbb2015 benchmark \cite{82}. The benchmark emulates operations of a market chain (supply orders, revenue statistics etc.) and in general emulates a distributed Java application.

We added the static collection probes at method entry and exit points of over 1200 methods. We then run the benchmark at a constant injection rate to have as stable workload as possible (at about 75\% of CPU utilization). At most one method at a time was monitored dynamically.

Figure \ref{fig:overhead_analysis} summarizes the course of our experiment. We have run the benchmark for about two hours before restarting it, during each restart we collected data from dynamic probes of about 20 methods. We always left enough time after the (dynamic) probe insertion and removal for the just-in-time compiler to stabilize before collecting data. During the measurement, we were also monitoring the CPU utilization and JIT compiler events to help us better detect transient states. To prevent inadvertent synchronizations, we inserted randomized pauses between individual actions.

Next we discuss the behavior of just-in-time compiler during class reload. When the Java Virtual Machine reloads a class, the JIT compiler has to discard all compiled versions of the class, including inlined code. The reloaded class is typically compiled into the native code again eventually.

Because our monitoring selected frequently called methods, we were interested especially in how long it typically takes before the class is recom-
Figure 3.12: Course of our experiment for collecting data for overhead analysis. Process of dynamic monitoring is captured in the shaded rectangle. Unlike in real-life deployment, we also collect data from static probes to gather baseline performance.

Figure 3.13: Distribution of time from start of code manipulation to end of immediately following compilations.

Figure 3.13 summarizes our results. It shows that recompilation is typically finished within 10 s. When we are inserting a probe, 50% of compilations are finished in 7 s, for probe removal it takes only 4 s. Only under 2% of compilations required more than 60 s to complete. As a rule of thumb, we can conclude that a delay of about half a minute is appropriate before actually recording the data from dynamic probes.
To assess the accuracy of the collected data we depended on our ability to measure the same code by two independent measurement infrastructures. During our experiment, each of the selected methods was at some time measured by both the static and the dynamic infrastructure. The static probe wraps the original code and provides the baseline performance information. The dynamic probe wraps the original code together with the static code and provides the observed performance. Obviously, for individual (paired) samples, the times collected by the dynamic probes are larger than the ones from the static probes. Ideally, the difference would be constant and equivalent to the time spent between reading the clock source in both probes.

The actual difference, aggregated across all methods, is depicted in Figure 3.14 as a ratio between the observed and the baseline performance. An ideal measurement framework would have all the points on the 1.0 line regardless of the method duration – both probes would collect (almost) the same values. In the figure, we can see that for shorter methods the difference is higher. For methods longer than 1 ms the difference is negligible. For methods with duration in the order of tens of microseconds, the ratio rises up to factor of one and half. Methods with execution time in units of microseconds have ratio as high as a factor of 3. For extremely short methods, the ratio can rise up to hundreds.

We can conclude that the practical limit of the accuracy of dynamic monitoring is in the order of tens of microseconds. Once the dynamic probes are reporting values under 50 μs, we need to interpret the results carefully, or possibly employ alternative means of performance monitoring.

Class reloading can cause a cascade of recompilations of seemingly unrelated code. We have investigated these effects, as caused by the dynamic probes, by comparing the baseline method performance when the application was not dynamically monitored and when the method was surrounded by a dynamic probe.
Figure 3.15: Ratio between measurements reported by static probes on methods during and after dynamic monitoring. For each method we computed average execution time when being dynamically monitored (i.e. when surrounded by a dynamic probe) and also average execution time without the dynamic probe (after its removal). The grey bars denote ratios where the difference to 1.0 is statistically significant under Welch’s one sided t test on $\alpha = 5\%$. Notice the logarithmic scale of the x axis.

Ideally, methods would have the same performance all the time and insertion of a dynamic probe wrapping the method would not influence its (baseline) performance at all.

Figure 3.15 illustrates our findings by plotting the ratio of baseline performance during dynamic monitoring to baseline performance after dynamic monitoring. Intuitively, we would expect that the histogram would have very high bars around 1.0 (noise can shift the ratios in both directions slightly), possibly with tail representing methods that were slowed down by the surrounded dynamic probe (e.g. the method would have longer bytecode, thus less inlining could be done, making the method slower).

Instead, we can see that the histogram stretches quite far away from 1.0 in both directions, suggesting that the very presence of a probe also impacts performance of code inside that probe. Notably, there is a significant amount of methods that actually performed better while being dynamically monitored.

Summary

Our experiment contributed information on the practical limits of dynamic performance monitoring on the Java platform. We find some results quite unexpected, and elsewhere discuss threats to validity of our experiment in great detail [39]. We believe effects similar to those in our (artificial) experiment can be observed in production environments, indicating that any attempt to interpret the results of dynamic monitoring must be adjusted accordingly. As with other contributions comprising this thesis, we have published the experimental framework and (due to data volume constraints) a subset of the collected data [40].
Part II

Collection of Papers
This thesis is based on approaches, methods and evaluations published in a scientific journal and on research conferences. In the following chapters we include the full versions of selected publications most relevant to the topic of the thesis.

The paper called *Unit Testing Performance with Stochastic Performance Logic* describes our contribution to performance unit testing and its automation. Our goals \((G_1)\) and \((G_2)\) are covered by this paper.

The paper starts with the software-engineering aspects – how to structure performance unit tests and what their binding to a programming language should look like. The paper continues with methods for robust statistical evaluation of the collected performance metrics to detect regressions. The evaluation includes a comparison of practical behavior of statistical tests in the domain of software performance and a case study demonstrating how performance unit testing could help developers detect performance regressions early in the development lifecycle.

The second paper, *Analysis of Overhead in Dynamic Java Performance Monitoring*, analyses and quantifies the overhead of dynamic performance monitoring on the Java platform, completing goal \((G_4)\).

An analysis of millions of measurements allowed the authors to formulate rules of thumb for precise performance monitoring. Among the questions the authors tried to answer were the following: What is the overhead of dynamic performance monitoring? What is the delay between probe activation and the moment it starts recording reliable data? Does the performance observed before a dynamic probe was added differ from the performance observed after the probe was removed?

The third paper, *Utilizing Performance Unit Tests To Increase Performance Awareness*, describes how performance unit tests can be converted into an interactive performance documentation, fulfilling goal \((G_3)\).
The evaluation shows that simple hints on performance behavior in the documentation can improve performance of individual application components. An experiment with students of advanced Java lectures shows that developers often have unrealistic expectations of the speed of their code and that even a straightforward task can lead to solutions that differ in speed by several orders of magnitude (without their authors realizing what is causing such difference).

A complete list of all author’s publications can be found at the end of the thesis.
Unit testing performance with Stochastic Performance Logic

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

Journal on Automated Software Engineering.


The original version is available electronically from the publisher’s site at http://dx.doi.org/10.1007/s10515-015-0188-0.
Unit testing performance with Stochastic Performance Logic

Lubomír Bulej 2 · Tomáš Bureš 1 · Vojtěch Horký 1 · Jaroslav Kotrč 1 · Lukáš Marek 1 · Tomáš Trojánek 1 · Petr Tůma 1

Received: 11 July 2014 / Accepted: 21 December 2015

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 implemented in a performance testing framework and evaluated in multiple experiments, demonstrating the ability to identify performance differences in realistic unit test scenarios.

Keywords Performance evaluation · Unit testing · Java

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 (Bourque and Fairley 2014)—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 extend the benefits to performance testing, that is, to use the unit tests to assess the temporal behavior of the implementation.

1.1 Motivation

To provide a realistic motivation, we borrow a practical example from the development history of JDOM, a software package “for accessing, manipulating, and outputting XML data from Java code” (JDOM Library 2013). We choose JDOM as a representative software project—with about 15,000 LOC, it is of reasonable size for experiments that require frequent compilation and manual code inspection; with over 1500 commits spread across 15 years of development, it provides ample opportunity to observe performance regressions; it also has an open source license.

One of the components making up JDOM is Verifier, a class responsible for performing various sanity checks on XML text. Among other tests, Verifier can check whether the characters used in a name are permitted by the XML specification. Following is the relevant code fragment from an early version of Verifier:

```java
public static String checkXMLName (final String name) {
  ...
  for (int i = 1 ; i < len ; i++) {
    if (!isXMLNameCharacter (name.charAt (i))) {
      return "Name cannot contain \"" + name.charAt (i) + "\"";
    }
  }
  ...
}

public static boolean isXMLNameCharacter (final char c) {
  return (isXMLLetter (c) || isXMLDigit (c) ||
          c == '.' || c == '-' || c == '_' || c == ':' ||
          isXMLCombiningChar (c) || isXMLExtender (c));
}

public static boolean isXMLLetter (final char c) {
  if (c < 0x0041) return false;
  if (c <= 0x005a) return true;
  if (c < 0x0061) return false;
  if (c <= 0x007a) return true;
  if (c < 0x00c0) return false;
  if (c <= 0x00d6) return true;
  // Some 200 similar tests follow ...
}
```
Obviously, Verifier is used often during parsing, the accumulated cost of individual checks can therefore impact the overall performance. Aware of this fact, JDOM developers have actually identified performance issues with the early version of Verifier—in particular, the way sequential range checks are applied means some names take longer to check than others, and even names made of common characters take relatively long to check (hunterhacker/jdom: Verifier performance 2013). To address these issues, a newer version of Verifier, which replaces branches with table lookup, was implemented:

```java
public static String checkXMLName (final String name) {
    ...
    for (int i = name.length () - 1 ; i >= 1 ; i--) {
        if (0 == CHARFLAGS [name.charAt (i)] & MASKXMLNAME)
            return "Name cannot contain \"" +
                     name.charAt(i) + "\";  
               }
    }
    ...
}
```

If this were a functional issue, chances are a unit test would have been written even before Verifier was implemented. Alternatively, a regression unit test (Beck 2002) would be written after the issue was remedied to prevent recurrence. Doing the same with a performance issue presents multiple challenges.

### 1.2 Challenges

Broadly, the goal of performance testing is making sure that the system under test executes fast enough. The very first challenge encountered when constructing a performance unit test is specifying what exactly is fast enough—the test condition.

The test condition is related to 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. 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. In the context of our motivating example, we need test conditions related to the identified performance issues. Rather than establishing an absolute time limit on the checkXMLName method, we need to express the requirement that regardless of what time a check takes, it does not change much with different character ranges, and possibly the even more vague requirement that this time is reasonable for the given workload and platform.
The second major challenge in constructing a performance unit test rests with the test implementation, which is sensitive to design errors. A performance unit test is essentially a microbenchmark, and many seemingly innocuous mistakes in microbenchmark design can produce misleading results.\footnote{C. Click. The Art of Java Benchmarking. http://www.azulsystems.com/presentations/art-of-java-benchmarking.}

Returning to our motivating example, we can illustrate this challenge on the implementation of the benchmark JDOM developers used to examine the 

\texttt{Verifier} performance. Available as the \texttt{PerfVerifier} class, the benchmark exhibits implementation patterns that are considered generally dangerous (Horký et al. 2015). For example, the return values of the measured methods are discarded, giving the compiler the option of optimizing away the very code that should be measured.

The third major challenge is associated with test execution, which is obviously sensitive to interference. A build environment can compile multiple tested units and execute multiple functional unit tests in parallel, but doing the same with performance unit tests would likely produce invalid results due to disruptions from parallel execution. Many other common mechanisms can interfere with measurements, prominent examples include the garbage collection and the just-in-time compilation.

To cope with interference and still provide sufficiently representative information, performance measurements require collecting multiple observations. This not only increases the test execution time, but also requires that the test conditions are evaluated probabilistically rather than deterministically. Besides measuring the tested methods multiple times, which the original \texttt{PerfVerifier} benchmark from above already does, we need to separate the transient performance artefacts from the steady state performance and evaluate the test conditions accordingly.

1.3 Contributions

We work on a performance testing framework that addresses these challenges (SPL Tool 2013). Central to the framework is Stochastic Performance Logic (SPL), a mathematical formalism for expressing and evaluating performance requirements (Bulej et al. 2012). The framework permits attaching performance requirement specifications to individual methods to define common performance tests (Horký et al. 2013).

SPL formulas express performance requirements by comparing performance of multiple methods against each other. This makes it easy for the developer to express common performance related assertions—in our motivating example, we can express requirements such as “future modifications of the \texttt{checkXMLName} method must not introduce performance regression”, “the \texttt{checkXMLName} method should perform equally well on names with different character ranges”, or “the performance of the \texttt{checkXMLName} method should have roughly the same order of magnitude as the performance of the standard string copying operation on the same workload and platform”.

Our performance testing framework provides support for implementing performance tests which are automatically executed and evaluated. The framework
cooperates with common versioning frameworks to support formulas that compare
current and historical performance—extending our motivating example, we evalu-
ate the use of the framework by implementing performance tests for assumptions
collected across the JDOM development history. Besides supporting our fitness-for-
purpose assertions, the experiments also show that some performance assumptions
made in the JDOM development history are now wrong.

SPL formulas are 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. To illustrate the trade off between factors such as test sensitivity,
measurement cost and false alarm rate, we evaluate the interpretations on collected
JDOM experiments.

We believe SPL formulas can be used for more purposes than just perfor-
mance testing. These uses include documenting performance provided by a method
implementation, or documenting performance expected from the environment of an
implementation. These, however, are not elaborated in detail in this paper.

This paper constitutes a comprehensive presentation of our performance testing
framework. Section 2 presents the basic definitions of the SPL formalism. Section 3
introduces practical SPL interpretations tailored for different measurement procedures.
The programming side of the performance test construction is described in Sect. 4.
Section 5 provides the experimental evaluation, including platform description that
also applies to all the illustrative measurements used throughout the paper. Connections
to related work are summarized in Sect. 6. Finally, Sect. 7 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 Bulej et al. (2012) and remain mostly the same here. The SPL
interpretations that consider multiple measurement runs are new, except for some initial
derivations that rely on Kalibera et al. (2005). The test construction is significantly
extended compared to Bulej et al. (2012). The experimental evaluation is partially
new and partially from Horký et al. (2013). The application of the SPL formalism for
performance awareness in the context of the ASCENS Project (Wirsing et al. 2015)
has been summarized in Bulej et al. (2015).

2 Stochastic performance logic

We address the need to express and evaluate performance requirements by introducing
a formal language for writing conditions on performance, and multiple interpreta-
ations that give meaning to these conditions in various contexts. The language and the
interpretations form the stochastic performance logic (SPL). Compared to the more
common treatment of performance as an intuitive concept with no formal background,
the formal framework enables a more rigorous reasoning about the properties of the
interpretations.

As in our motivating example, we consider the execution time of a particular method
on particular inputs. The time can depend on the inputs, we therefore introduce method
workload to model the inputs, and workload class to aggregate inputs whose impact on execution time is similar. Each workload class is described by workload parameters.

**Definition 1** Workload class is a function \( \mathcal{L} : W^n \rightarrow (\Omega \rightarrow I) \), where for a given \( \mathcal{L} \), \( W \) is a set of workload parameter values, \( n \) is the number of parameters, \( \Omega \) is a sample space, and \( I \) is a set of objects serving as method inputs in a chosen programming language. For some later definitions, we also require that there is a total ordering over \( W \).

**Definition 2** Method workload is a random variable \( L_{w_1, \ldots, w_n} \) such that \( L_{w_1, \ldots, w_n} = \mathcal{L}(w_1, \ldots, w_n) \) for a given workload class \( \mathcal{L} \) and workload parameters \( w_1, \ldots, w_n \).

Unlike conventional random variables, which map the individual observations to real numbers, method workload is a random variable that maps observations to object instances, which serve as random inputs for the method under test.

In our motivating example, we measure the performance of the checkXMLName method, which expects a single string as input. Assuming we are interested in the impact of the string length on the execution time, our workload parameter is simply the length, and our workload class is \( \mathcal{L}_{\text{checkXMLName}} : \mathbb{N} \rightarrow (\Omega_{\text{checkXMLName}} \rightarrow I_{\text{checkXMLName}}) \). The method workload returned by the workload class is a random variable whose observations are instances of random strings of given length. If we were also interested in the impact of forbidden characters on the execution time, we could add another workload parameter, denoting the position of the forbidden character, and our workload class would be \( \mathcal{L}_{\text{checkXMLName}} : \mathbb{N}^2 \rightarrow (\Omega_{\text{checkXMLName}} \rightarrow I_{\text{checkXMLName}}) \).

Without loss of generality, further definitions assume there is exactly one \( \mathcal{L}_M \) for a particular method \( M \) and that \( M \) has just one input argument.

With the formalization of workload in place, we proceed to define the method performance.

**Definition 3** Let \( M(i) \) be a method in a chosen programming language and \( i \in I \) its input argument. Then method performance \( P_M : W^n \rightarrow (\Omega \rightarrow \mathbb{R}^+) \) is a function that for given workload parameters \( w_1, \ldots, w_n \) returns a random variable whose observations correspond to the execution duration of method \( M \) with input argument \( i \) obtained from the observations of \( L_{M}^{w_1, \ldots, w_n} = \mathcal{L}_M(w_1, \ldots, w_n) \), where \( \mathcal{L}_M \) is the workload class for method \( M \).

In our motivating example, when our workload parameter is the string length, then \( P_{\text{checkXMLName}} : \mathbb{N} \rightarrow (\Omega \rightarrow \mathbb{R}^+) \) returns a random variable that models the execution duration of checkXMLName on strings of particular length.

Modeling execution time as an absolutely continuous random variable is quite common, as it enables the use of statistical inference methods for performance evaluation. Although the measurements are quantized and therefore discrete, time itself is considered continuous and the corresponding probability distribution function is constructed to interpolate appropriately. For some later definitions, we also require that the random variable has a finite expectation. This is not only evidently practical, but also true
for common probability distribution function constructions such as piecewise linear interpolation.

We can now define the language of the SPL, which allows us to make comparative statements about method performance under a particular method workload.

**Definition 4** SPL is a many-sorted first-order logic defined as follows:

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

Intuitively, $F_P$ contains the names of methods whose performance is subject to comparison, $F_T$ contains the names of functions that scale method performances before comparison, relations $\equiv$, $\leq$ compare workload parameters, and relations $\equiv_p$, $\leq_p$ compare scaled method performances.

**Definition 5** For $x, y, z \in W$ and $P_M, P_N \in F_P$, SPL has the following axioms:

\[
\begin{align*}
&x \leq x \tag{1} \\
&(x \leq y \land y \leq x) \iff x = y \tag{2} \\
&(x \leq y \land y \leq z) \to x \leq z \tag{3}
\end{align*}
\]

For each pair $tl, tr \in F_T$ such that

\[
\forall o \in \mathbb{R}^+: tl(o) \leq tr(o), \text{ there is an axiom} \tag{4}
\]

\[
\begin{align*}
P_M(x_1, \ldots, x_m) \leq_p tl(tr) & \quad P_M(x_1, \ldots, x_m) \leq_p tl(tr) \tag{5a} \\
(P_M(x_1, \ldots, x_m) \leq_p tm(tr) & \quad P_N(y_1, \ldots, y_n) \leq_p pm(tr) \langle x_1, \ldots, x_m) \rangle \tag{5b} \\
\iff & \quad P_M(x_1, \ldots, x_m) =_p pm(tr) \quad P_N(y_1, \ldots, y_n) \tag{5c}
\end{align*}
\]

Axioms (1)–(3) are common arithmetic axioms applied to the totally ordered workload parameters from $W$. In an analogy to (1)–(2), axiom (4) can be regarded as generalised reflexivity. Axiom (5) shows the correspondence between $\equiv_p$ and $\leq_p$. An analogy of transitivity (3) is not introduced for $\equiv_p$ and $\leq_p$, because it does not hold for the sample-based interpretations defined in Sect. 3.

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 where the performance relations $\equiv_p$ and $\leq_p$ are concerned. Specifically, the lack of transitivity for performance relations ensures that SPL formulas can only express statements that are consistent with the hypothesis testing approaches used in the SPL interpretations.

Using SPL formulas, we can express assumptions about performance in the spirit of our motivating example. As a notation shortcut, we specify the performance obser-
vation transformation functions in place—that is, \( p(x,10 \cdot x) \) is a shortcut for \( p(tl,tr) \) where \( tl(x) = x \) and \( tr(x) = 10 \cdot x \):

**Example 1** “On names of 5, 10, 50, 100, 500 and 1000 characters, the execution time of the checkXMLName method must not exceed the execution time of the standard string copying operation copyString more than ten fold.”

\[
\forall n \in \{5, 10, 50, 100, 500, 1000\} : 
P_{\text{checkXMLName}}(n) \leq p(x,10 \cdot x) \quad P_{\text{copyString}}(n)
\]

**Example 2** “On names of 5, 10, 50, 100, 500 and 1000 characters, the current name check method implementation checkXMLNameCurr must not introduce performance changes of more than 5 the historical implementation checkXMLNameHist.”

\[
\forall n \in \{5, 10, 50, 100, 500, 1000\} : 
P_{\text{checkXMLNameCurr}}(n) \leq p(x,1.05 \cdot x) \quad P_{\text{checkXMLNameHist}}(n) \\
\land P_{\text{checkXMLNameHist}}(n) \leq p(0.95 \cdot x, x) \quad P_{\text{checkXMLNameCurr}}(n)
\]

The SPL formulas provide sufficient flexibility for some more ambitious uses. We can, for example, attempt to express assumptions about algorithmic complexity:

\[
\forall n \in 1, \ldots, 1000 : P_{\text{getFromTree}}(n) = p(x, \log(x)) \quad P_{\text{getFromList}}(n)
\]

Unfortunately, while this formula may appear to state the intuitive property that the time to get an element from a tree is logarithmic compared to the time to get an element from a list of the same size, it does not work well in practice. For one thing, it lacks the multiplicative constants required to make the equality hold. For another, algorithmic complexity rarely translates into execution time in a simple fashion. While we believe this particular application of SPL is worth investigating further, it will require different interpretations and different measurement methods than those presented here.

The idea of applying SPL formulas to iteration counts rather than execution time can be extended further. We can easily imagine SPL formulas that express assumptions about arbitrary quantifiable properties such as power or memory consumption—again, however, these are uses that would likely require different interpretations and different measurement methods, and are not a contribution of this paper.

Finally, we point out that the use of random variables invites interpretations that consider measurements to be independent observations of said variables. In situations where this does not hold—for example when each single measurement requires an expensive setup to eliminate the effects of the previous ones—our formal framework is not suitable. Chances are such situations do not lend themselves to unit testing at all.

With the SPL language in place, we now proceed to defining the SPL interpretations. The semantics given to the SPL formulas by the interpretations follows the intent expressed in Examples 1 and 2, but the individual interpretations differ in the context they target.
3 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 the SPL interpretation, and particularly the interpretation of equality and inequality relations, on the expected value of method performance. We introduce the expected-value-based interpretation first—although it assumes the expected value is known, which is seldom the case in practice, it provides an intuitive starting point for explaining the other SPL interpretations we develop. These are the quick mean-value interpretation, meant for situations where quick measurement and evaluation are needed, and the parametric and non-parametric interpretations that explicitly address changing measurement conditions to achieve higher sensitivity when more measurements become available.

3.1 Expected-value-based interpretation

Each function symbol $P_M, P_N, \ldots \in F_P$ is interpreted as a method performance, i.e., an $n$-ary function that for workload parameters $w_1, \ldots, w_n$ returns a random variable $\Omega \rightarrow \mathbb{R}^+$, whose observation corresponds to Definition 3.

Each function symbol $tm, tn, \ldots \in F_T$ is interpreted as a performance observation transformation function, i.e., a function $\mathbb{R}^+ \rightarrow \mathbb{R}^+$ used in the context of equality and inequality relations between method performances.

The relational operators $\leq$ and $=p$ for arity $W \times W$ are interpreted in a standard way based on total ordering of $W$.

The interpretation of the relational operators $=p$ and $\leq p$ is defined as follows:

**Definition 6** Let $tm, tn : \mathbb{R}^+ \rightarrow \mathbb{R}^+$ 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 $\leq p(tm, tn), = p(tm, tn) : (\Omega \rightarrow \mathbb{R}^+) \times (\Omega \rightarrow \mathbb{R}^+)$ are interpreted as follows:

\[
P_M(x_1, \ldots, x_m) \leq p(tm, tn) P_N(y_1, \ldots, y_n) \quad \text{iff}
\]
\[
E(tm(P_M(x_1, \ldots, x_m))) \leq E(tn(P_N(y_1, \ldots, y_n)));
\]
\[
P_M(x_1, \ldots, x_m) = p(tm, tn) P_N(y_1, \ldots, y_n) \quad \text{iff}
\]
\[
E(tm(P_M(x_1, \ldots, x_m))) = E(tn(P_N(y_1, \ldots, y_n)));
\]

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 $X$.²

² 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.
For proofs that the expected-value-based interpretation is consistent with the SPL axioms, refer to ‘Expected-Value-Based Interpretation’ section in Appendix.

While the above interpretation illustrates the idea behind SPL, it assumes that the expected value $E(tm(X))$ can be computed. Unfortunately, this assumption hardly ever holds, because the probability distribution function of $X$ is typically unknown and so is the expected value. Instead, a practical interpretation has to rely on measurements.

### 3.2 Quick mean-value interpretation

To apply SPL when only measurements—that is, observations of the relevant random variables rather than the probability distribution functions—are available, we turn to sample based methods that work with estimates of distribution parameters derived from the measurements. The basic idea is to interpret the $=_p$ and $\leq_p$ relations as statistical tests. Given a set of observations of method performances, the test determines whether it is reasonably likely that the mean values of the observed method performances are in a particular relation, $=_p$ or $\leq_p$.

To formulate a sample-based interpretation, we first need to define what observations will be used to interpret the $=_p$ and $\leq_p$ relations. We assume an interpretation will use a finite set of observations of method performances under a particular method workload, which we call an experiment:

**Definition 7** Experiment $\mathcal{E}$ is a collection of $\mathcal{O}_{P_M(w_1,\ldots,w_m)}$, where

$$\mathcal{O}_{P_M(w_1,\ldots,w_m)} = \{ P^1_M(w_1,\ldots,w_m), \ldots, P^V_M(w_1,\ldots,w_m) \}$$

is a set of $V$ observations of method performance $P_M$ subjected to workload $L^{w_1,\ldots,w_m}_M$, and where $P^i_M(w_1,\ldots,w_m)$ denotes the $i$-th observation of method performance $P_M$.

With the concept of an experiment in place, we first introduce the quick mean-value interpretation, suitable for situations where quick measurement and evaluation take precedence over evaluation robustness. The interpretation relies on Welch’s $t$ test (Welch 1947) as a correspondingly simple formal procedure. The definitions are the same as before, except for Definition 6, used to assign semantics to the method performance relations.

**Definition 8** Let $tm, tn : \mathbb{R}^+ \rightarrow \mathbb{R}^+$ 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 $\alpha \in (0, 0.5)$ be a fixed significance level.

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

$$H_0 : E(tm(P^i_M(x_1,\ldots,x_m))) \leq E(tm(P^j_N(y_1,\ldots,y_n)))$$

cannot be rejected by one-sided Welch’s $t$ test at significance level $\alpha$ based on the observations gathered in the experiment $\mathcal{E}$;
Fig. 1 Execution times of the four example functions. 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

\[ P_M(x_1, \ldots, x_m) = p(tm, tn) \quad P_N(y_1, \ldots, y_n) \text{ iff the null hypothesis} \]

\[ H_0 : E(tm(P_M(x_1, \ldots, x_m))) = E(tn(P_N(y_1, \ldots, y_n))) \]

cannot be rejected by two-sided Welch’s t test at significance level 2α based on the observations gathered in the experiment \( E \),

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

For proofs that the quick mean-value interpretation is consistent with the SPL axioms, refer to ‘Quick Mean-Value Interpretation’ section in Appendix.

### 3.2.1 Considering normality violations

The use of Welch’s t test deserves closer attention because the assumptions contained in the test design do not necessarily hold for this test application. In particular, Welch’s t test assumes that the sample means are normally distributed, which holds when the samples themselves are normally distributed; normal sample distribution is therefore considered a requirement. In practice, this requirement is often not observed strictly—due to the central limit theorem, the sample mean distribution approaches normal as the number of samples increases regardless of the sample distribution.

Because the exact impact of normality violation depends on the nature of the samples, empirical experiments are used to support or challenge the practice of violating the normality assumptions (Chen et al. 2015). We do the same here, illustrating the behavior of Welch’s t test on four methods whose execution time distributions decidedly break the normality assumptions. The execution time histograms of the four methods are given in Fig. 1—the distributions are unimodal with a tail, bimodal with small and large coefficient of variation, and quadrimodal.

For each of the four methods, we introduce a change in method performance and look at how likely the test is to detect the change or raise a false alarm, depending on the magnitude of the change and the number of samples used. In statistical terms, we
postulate the null hypothesis (performance did not change) and examine the probability that the test rejects the null hypothesis in favor of a correct (performance decrease) or an incorrect (performance increase) alternative. Given a set of measurements $M$ of method $m$ and a change of scale $s > 1$ on $n$ measurements, we compute as follows:

1. Using random sampling without replacement, we split $M$ in disjoint halves $M_X$ and $M_Y$, $M = M_X \cup M_Y$, $M_X \cap M_Y = \emptyset$.
2. Using random sampling with replacement, we create a set of measurements for each half, producing $X$ and $Y$ of size $n$, $x \in X \Rightarrow x \in M_X$, $y \in Y \Rightarrow y \in M_Y$.
3. We scale the measurements in $Y$ by $s$, producing $Z = \{y \cdot s : y \in Y\}$.
4. We determine whether the 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 $\alpha = 0.01$.

We run the computation enough times to estimate the probabilities of the tests (correctly) favoring $\bar{Z} > \bar{X}$ and (incorrectly) favoring $\bar{X} > \bar{Z}$, which together characterize the test sensitivity. We then evaluate the dependency of the test sensitivity on the number of measurements $n$.

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 methods and other values of $s$. These results indicate that a test would require an unreasonably high number of measurements to detect changes of 0.1%, while changes of 10% are easily detected even on a small number of measurements.

For cases where normality violation matters, it is of course possible to use a non-parametric test, such as the Mann–Whitney–Wilcoxon test, in place of the Welch’s $t$ test. Still, when we consider the combination of robustness, sensitivity and computational cost, this does not appear to bring major benefits.

Welch’s $t$ test also requires that the observations $O_{P_{M}(p_1,\ldots,p_m)}$ are sampled independently. This matches the formalization of method performance in Definition 3, however, practical environments are likely to introduce statistical dependence between measurements. In particular, measurements collected between restarts of the experiment environment tend to be much more dependent than measurements collected
across restarts. As a consequence, method performance can appear to change between restarts even when neither the method nor the environment was actually modified (Kalibera et al. 2005).

To address statistical dependence between measurements, we assume that method performance observations \( P^i_M(w_1, \ldots, w_m) \) from Definition 7 come from runs, using \( P^i_M(x_1, \ldots, x_m) \) to denote the \( j \)-th observation in the \( i \)-th run. Partial or complete restarts of the experiment environment occur between runs. We investigate two major sources of dependent observations, the initial transient conditions of each measurement run and the changes of conditions between individual measurement runs.

### 3.3 Handling initial transient conditions

As an inherent property of many environments, 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. This effect is illustrated in Fig. 3.

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.

Many other mechanisms can introduce warmup. We use the HotSpot virtual machine as an example—in addition to just-in-time compilation, it can defer locking optimization for some time after start (Dice 2006; Dice et al. 2013), gradually adjust the heap size to reflect the application behavior (Oracle 2014), and more.

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.

![Fig. 3 Example of how initial transient conditions influence method execution time. The measured 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](image-url)
Unfortunately, the configuration changes that help reduce the initial transient conditions can also impact the steady state performance. We illustrate this on HotSpot 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. Obviously, the impact is almost arbitrary, suggesting that attempts to avoid the initial transient conditions have practical limits.

Warmup measurements can sometimes also be identified by analyzing the collected observations. Intuitively, long sequences of observations with zero slope (such as those on the right side of Fig. 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 Fig. 3), which likely come from warmup. Unfortunately, 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 (Oracle 2006) to discard observations taken before most method compilations complete. Afterwards, the logic interpretation can assume the observations do not suffer from transient initial conditions.

3.4 Handling condition changes between runs

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 conditions, which can affect the measurements. The memory layout example is one where a significant impact was observed in multiple experiments (Kalibera et al. 2005; Mytkowicz et al. 2009). 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 $S^2$ computed from all the observations together is not a reliable estimate of the population variance $\sigma^2$ and the quick mean-value interpretation becomes more prone to false positives, rejecting performance equality even between measurements that differ only due to changing conditions between runs. In our previous work (Horký et al. 2013), we solve the problem by introducing a sensitivity limit, ignoring changes below 5%. Here, we improve on Horký et al. (2013) with two logic interpretations that explicitly consider condition changes between runs.

3.5 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 (Kalibera et al. 2005). 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:
\[
\overline{M} = \frac{1}{o} \sum_{i=1}^{r} \sum_{j=1}^{o} P_{M}^{i,j}(x_1, \ldots, x_m)
\]

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

\[
\overline{M} \sim N(\mu, \rho^2 r + \frac{\sigma^2}{ro})
\]

where \( \sigma^2 \) denotes the average of run variances and \( \rho^2 \) denotes the variance of run means (Kalibera et al. 2005).

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 \( o \) (Kalibera et al. 2005).

**Definition 9** 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 \( \alpha \in (0, 0.5) \) be a fixed significance level.

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

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

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

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

\[
\frac{S_M^2}{r_M} = \frac{\sum_{i=1}^{r} \sum_{j=1}^{o} \left( t_m(P_{M}^{i,j}(x_1, \ldots, x_m)) - \frac{1}{o} \sum_{k=1}^{o} t_m(P_{M}^{i,k}(x_1, \ldots, x_m)) \right)^2}{r_M(o_M - 1)}
\]

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

and similarly for \( S_N^2 \) and \( R_N^2 \).
Fig. 5  Example histogram of run means from multiple measurement runs of the same method and work- 
load. Each run collects \( o = 20,000 \) observations after a warmup of 40,000 observations. The method is 
\texttt{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

\[
-p_M(x_1, \ldots, x_m) = p_{(m,tn)} P_N(y_1, \ldots, y_n) \iff
\left| \overline{M} - \overline{N} \right| \leq z(1-\alpha) \sqrt{\frac{o_M R_M^2 + S_M^2}{r M O_M} + \frac{o_N R_N^2 + S_N^2}{r N O_N}}
\]

For proofs that the parametric mean-value interpretation is consistent with the SPL 
axioms, refer to ‘Parametric Mean-Value Interpretation’ section in Appendix.

The interpretation relies on the central limit theorem to approximate the distribution 
of run means \( \overline{M_i} = o^{-1} \sum_{j=1}^{o} P_M^{i,j}(x_1, \ldots, x_m) \) with normal distribution. Figure 5 
illustrates that a high number of observations \( o \) in a run may not be enough to have the 
run means distributed normally. This means that the approximation may also require 
a high number of runs \( r \) to be reliable. Next, we therefore construct an interpretation 
that uses the distribution of \( \overline{M_i} \) directly.

3.6 Non-parametric mean-value interpretation

Informally speaking, the sample-based interpretations work by estimating what 
magnitude of variations is common within the measurements on each side of a 
comparison. Then, only those differences in method performance that exceed this 
magnitude are considered significant. The parametric mean-value interpretation has 
done this by approximating the performance variations with a normal distribution. 
The non-parametric mean-value interpretation will construct an empirical probability 
distribution function of the performance variations instead.

As a practical advantage, the interpretation makes it possible to construct the empirical 
probability distribution function of the performance variations not only from the 
measurements under comparison, but also from measurements of methods with similar 
performance variations—for example historical versions of the same methods.

Formally, 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:

- \( B_{\bar{M}, r_M, o_M} \) is the probability distribution function of

\[
(r_M o_M)^{-1} \sum_{i=1}^{r_M} \sum_{j=1}^{o_M} tm \left( \hat{P}_{i}^{j} (x_1, \ldots, x_m) \right)
\]

where \( \hat{P}_{i}^{j} (x_1, \ldots, x_m) \) denotes a random variable with distribution \( B^c_M \) for \( c \) drawn randomly once for each \( i \). In other words, \( B_{\bar{M}, r_M, o_M} \) denotes a distribution of a mean computed from \( r_M \) runs of \( o_M \) observations each.

- \( B_{\bar{M}, r_M, o_M - N, r_M, o_M} \) is the probability distribution function of the difference

\[ \hat{M} - \hat{N} = (\bar{M} - \bar{N}) \]

where \( \hat{M} \) is a random variable with distribution \( B_{\bar{M}, r_M, o_M} \), \( \hat{N} \) is a random variable with distribution \( B_{\bar{N}, r_N, o_N} \), and the expression \( (\bar{M} - \bar{N}) \) shifts the distributions \( B_{\bar{M}, r_M, o_M} \) and \( B_{\bar{N}, r_N, o_N} \) to have equal means.

Having the notation in place, the performance comparison is defined as

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

\[ B_{\bar{M}, r_M, o_M - \bar{N}, r_N, o_N} (\bar{M} - \bar{N}) \leq 1 - \alpha \]

where \( \bar{M} \) denotes the sample mean of \( tm(P_M(x_1, \ldots, x_m)) \), \( \bar{N} \) is defined similarly.

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

\[ \alpha \leq B_{\bar{M}, r_M, o_M - \bar{N}, r_N, o_N} (\bar{M} - \bar{N}) \leq 1 - \alpha \]

An important problem is that the distribution functions \( B_{\bar{M}, r_M, o_M} \) and \( B_{\bar{N}, r_N, o_N} \), and consequently \( B_{\bar{M}, r_M, o_M - \bar{N}, r_N, o_N} \) are unknown. To get over this problem, we approximate the distributions by bootstrap and Monte-Carlo simulations (Sheskin 2011). This can be done either by using the observations \( P_M(x_1, \ldots, x_m) \) 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 \( \hat{\theta}_{i,o} = o^{-1} \sum_{j=1}^{o} \hat{P}_{i}^{j} (x_1, \ldots, x_m) \) of some method \( X \) is estimated by sampling its bootstrap version \( \theta^*_{i,o} = o^{-1} \sum_{j=1}^{o} P^*_{X}^{i,j} (x_1, \ldots, x_m) \), where samples \( P^*_{X}^{i,j} (x_1, \ldots, x_m) \) are randomly drawn with replacement from samples \( P_{X}^{i,j} (x_1, \ldots, x_m) \).

Extending this line of reasoning to the mean of run means \( \bar{X}_{r,o} = r^{-1} \sum_{i=1}^{r} \hat{\theta}_{i,o} \), we estimate the distribution of \( \bar{X}_{r,o} \), i.e., \( B_{\bar{X}_{r,o}} \), 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. $\overline{X}_{r,o}$ in our case), where $1(A)$ denotes the indicator function of a statement $A$.

The whole apparatus of the bootstrap and the Monte-Carlo simulation can then be used to create the bootstrapped distributions $B_{\overline{X}_{r,o}^*}^*$, $B_{\overline{Y}_{r,y}^*}^*$ and their difference $B_{\overline{X}_{r,o}^*}^* - B_{\overline{Y}_{r,y}^*}^*$. To obtain the desired test distribution $B_{\overline{M}_{r,m}^* - \overline{N}_{r,n}^*}$, we use the approximation $B_{\overline{X}_{r,x}^* - \overline{Y}_{r,y}^*}$, 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 10** Let $tm, tn : \mathbb{R}^+ \rightarrow \mathbb{R}^+$ 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, $\alpha \in (0, 0.5)$ 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 $E$, 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
  $$B_{\overline{X}_{r,x}^* - \overline{Y}_{r,y}^*} (\overline{M} - \overline{N}) \leq 1 - \alpha$$

- $P_M(x_1, \ldots, x_m) =_{p(tm,tn)} P_N(y_1, \ldots, y_n)$ iff
  $$\alpha \leq B_{\overline{X}_{r,x}^* - \overline{Y}_{r,y}^*} (\overline{M} - \overline{N}) \leq 1 - \alpha$$

For proofs that within certain practical restrictions, the non-parametric mean-value interpretation is consistent with the SPL axioms, refer to ‘Parametric Mean-Value Interpretation’ section in Appendix.

### 4 Software development integration

With unit testing generally accepted as a 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 therefore strive to integrate SPL-based performance testing into the development process in a manner similar to that of commonly used unit testing frameworks. We also aim for similar use cases:

**Regression testing** A developer modifies an implementation of a particular method to fix a performance issue, and wants to make sure that the performance issue is resolved or stays resolved. Alternatively, a developer refactors or reimplements a particular method and wants to make sure the new version improves performance. **Capturing assumptions** A developer implements a method while assuming certain specific (relative or absolute) performance from his own, system, or third-party component and wants to make this assumption explicit in the code that relies on it, expecting a notification if the assumption fails or ceases to hold over time. **Providing documentation** A developer implements a method and wants to describe its performance by relating it to performance of another method with well understood performance behavior.

With these goals in mind, we describe and address the fundamental differences between functional and performance unit testing that influence how a performance unit test can be expressed and integrated into a programming language.

### 4.1 The anatomy of a performance unit test

Unit testing frameworks share the common structure of a unit test, which consists of the setup, execution, validation, and cleanup phases (ANSI/IEEE 1986; Beck 1997). Each test is represented by a single method, which serves as an entry point to a test-specific implementation of these phases. Most testing frameworks allow the setup and cleanup phases to be implemented outside the test method, because they concern the test context which is commonly shared among several tests, however, the test method retains the responsibility for performing the execution and validation phases. The execution phase performs operations on the test subject, and the validation phases ensures that the execution phase did what it was supposed to do. The entire test is written from the perspective of a client who is the sole arbiter of the correctness of the tested behavior. The testing framework is merely responsible for orchestrating the invocation of (suitably marked) test methods in a test-specific context—completely oblivious to what the test methods do to determine the test results.

The structure of a functional unit test, represented as a template method, is shown in Listing 1. The template method sequentially invokes test-specific methods corresponding to individual test phases. The ordering of method invocations is determined by anticipated data dependencies between the phases. The template method does not prescribe how to pass the data from one phase to the next, but the test class instance provides a common context which can be used for this purpose. Because each test phase is only executed once, the data exchanged between the phases is also used only once. The internal control flow of each phase is typically independent of the others.

When deriving a performance unit test structure from the functional unit test structure, we first need to obtain performance (execution time) measurements for the execution phase. To this end, we can surround the `execute` method invocation with
public abstract class FunctionalUnitTest {

    // Fixed test template method.
    public final boolean test() {
        setUp();
        execute();
        boolean result = validate();
        tearDown();
        return result;
    }

    // Implemented by test developer.
    protected abstract void setUp();
    protected abstract void execute();
    protected abstract boolean validate();
    protected abstract void tearDown();
}

Listing 1: Template method capturing the structure of a functional unit test

code to measure the elapsed time, and invoke the method repeatedly in a loop, collecting measurements for each invocation. The validation phase then analyzes the measured data (instead of inspecting the program state) to determine the test result.

One issue with this arrangement is that the execute needs to support repeated execution. This means that it either needs to be idempotent, or there must be some other mechanism to ensure equivalent conditions prior to all invocations of the execute method. For example, when measuring the time it takes to insert one new item into a hash map with one million items, we always need to start with a hash map instance with one million items. This means that we will either need to remove one item after each addition operation, or to create a new instance with one million items from scratch for each invocation. Similarly, if we measure an operation on a freshly opened file, we need to close and open the file between measurements.

A simple solution is to loop over the entire unit test, including the setup and cleanup phases. This only works when the setup and cleanup phases are cheap enough to be repeated in each iteration. Expensive setup and cleanup phases become an issue for timely test evaluation—robust statistical analysis requires thousands of performance measurements and efficiency in collecting the performance data is therefore important. A more complex solution introduces per-test and per-iteration variants of the setUp and tearDown methods from Listing 1.

As another issue, the number of loop iterations needed to collect representative test data is not known beforehand—measurement is performed in presence of disruptive mechanisms such as garbage collection and just-in-time compilation, whose effect is not easily predicted. The number of iterations therefore needs to be controlled dynamically by the testing framework.

The tentative conversion of a functional unit test to a performance unit test following these steps is shown in Listing 2. The developer is responsible for ensuring that the conditions prior to all invocations of the execute method are equivalent, by implementing the appropriate operationSetUp and operationTearDown methods. On the
public abstract class TentativePerformanceUnitTest {

    // Fixed test template method.
    public final boolean test() {
        testSetUp();

        while (needMoreData()) {
            operationSetUp();
            beginMeasurement();
            execute();
            endMeasurement();
            operationTearDown();
        }

        boolean result = evaluate();
        testTearDown();
        return result;
    }

    // Provided by testing framework.
    protected final boolean needMoreData() { ... };
    protected final void beginMeasurement() { ... };
    protected final void endMeasurement() { ... };
    protected final boolean evaluate() { ... };

    // Implemented by test developer.
    protected abstract void testSetUp();
    protected abstract void operationSetUp();
    protected abstract void execute();
    protected abstract void operationTearDown();
    protected abstract void testTearDown();
}

Listing 2: Template method capturing the tentative structure of a performance unit test

other hand, the developer is no longer responsible for implementing the validation phase, now handled by the provided evaluate method.

Because the test evaluation is handled by the framework, we need to tell it what performance comparison to apply to determine the test result. While comparison with absolute execution time limits is straightforward, comparison with performance of other methods is more complicated—the performance data for the other methods needs to be collected as well. Although we can imagine collecting the performance data by introducing multiple execution phases in one test (one for each method referenced in the performance comparison formula), such an arrangement is not practical—it enforces particular execution order and complicates the test structure. We therefore further depart from the functional unit test analogy, and arrive at the concept of a performance unit test composed of two principal components—the decision component for describing the validation phase and the workload components for collecting the performance data.

Decision component The decision component corresponds to an SPL formula capturing a performance assumption. It provides the performance testing frame-
work with complete information required to execute and evaluate a performance test—the methods to be tested and the workload to be used. The decision component is typically attached to relevant source code entities, but in some use cases also needs to support standalone specification.

### Workload component

The workload component encapsulates a parametric implementation of the method workload, and determines what inputs to use to exercise a measured method of certain signature. This component needs to be implemented by the test developer, typically as a part of a project-specific library of workloads.

Separate workload components provide the testing framework with flexibility needed for efficiently collecting representative performance data. As noted earlier, this includes collecting measurements from multiple test runs to properly sample the observable variance in performance.

So far, our exposition to performance unit test structure was mostly platform independent. We now discuss the specifics of mapping the decision and workload components into Java.

### 4.2 Decision component in Java

The decision component of a performance unit test is an SPL formula that captures a performance assumption. To simplify maintenance, it is desirable to associate the SPL formula with the code it refers to. Towards this goal, we use annotations—Java supports annotations as a means to associate arbitrary meta data with various program elements, we use an `@SPL` annotation to attach the SPL formula to the relevant method.

Due to limited expressive power of Java annotations (and limited support for language embedding in general), we can only attach SPL formulas to methods as plain strings, but this 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 the use of the `@SPL` annotation is shown in Listing 3. The `Properties::getProperty` method is annotated with an SPL formula capturing the assumption that accessing a named property in a collection of properties has the same performance as accessing a hash table. The `THIS` shortcut refers to the annotated method.

While it is possible to attach an SPL formula to an abstract (interface) method, we note that the formula can only be evaluated with concrete method implementations.

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

**Listing 3:** Documenting expected performance with an annotation
m1 := org.jdom2.input.SAXBuilder.build@6a49ef6
m2 := org.jdom2.input.SAXBuilder.build@4e27535
w := saxBuilderTest

# f denotes the workload parameter
# here the workload parameter is an XML file
# passed as second argument to saxBuilderTest
for f in { "tiny1.xml", "tiny2.xml", "big.xml" }:
    m1[w](f) >= m2[w](f)

Listing 4: Documenting performance improvement between two method versions. The description references the workload component, implemented by the `saxBuilderTest` method, shown in Listing 7

exercised by concrete workload components. The performance assumptions expressed by such an SPL formula are not automatically propagated through polymorphism.

Performance assumptions intended to prevent performance regressions can also use annotations, especially when they refer to the current version of a method. However, SPL formulas in the context of regression testing will often need to refer to method versions that may no longer exist in the active code base. To this end, SPL formulas can be also stored in a separate file with a descriptive name.

An example demonstrating performance unit test with an SPL formula in an external file is shown in Listing 4. The formula captures the assumption that the (newer) version 4e27535 of the `SAXBuilder::build` method is faster than the (older) version 6a49ef6. 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.

Finally, Listing 5 follows the spirit of Examples 1 and 2, specifying that the current version of the `Verifier::checkElementName` method should not regress more than

```java
history := org.jdom2.Verifier.checkElementName@JDOM-2.0.0
current := org.jdom2.Verifier.checkElementName@HEAD
copy := copyString
w := elementNameTest

# l denotes the workload parameter
# here the workload parameter is the length
# of the element name that is being checked
for l in { 5, 10, 50, 100, 500, 1000 }:
    10 * current[w](l) <= copy[w](l) and
    current[w](l) <= 1.05 * history[w](l)

Listing 5: Documenting performance requirement on current method versions. The implementation of the string copying operation and the workload component are omitted for brevity, syntactic sugar is used for the performance transformation functions.
```
public final class SPLController {
    public final boolean needsMore() { ... }
    public final void start() { ... }
    public final void end() { ... }
}

public class TrivialPerformanceTests {
    public void trivialTest(SPLController spl) {
        // optional code for testSetUp
        while (spl.needsMore()) {
            // optional code for operationSetUp
            spl.start();
            // measured code
            spl.end();
            // optional code for operationTearDown
        }
        // optional code for testTearDown
    }
}

Listing 6: Compact structure of the workload component

5% beyond version 2.0.0 and should not exceed the execution time of a string copying operation more than ten fold.

4.3 Workload component in Java

The workload component includes the setup, execution and cleanup phases, structured as outlined in Listing 2. To support simple tests, which often have short or even empty setup and cleanup phases, we depart from the template approach—rather than implementing the test template method in the testing framework and having the developer code a test-specific implementation of the setUp, execute and tearDown methods, we let the developer code a compact workload component and provide framework methods to delineate the measured code and control the iteration. With short setup and cleanup code, this arrangement is preferable to defining test-specific setup and cleanup methods, whose boilerplate would exceed useful code. The structure is shown in Listing 6, an entire test for the SAXBuilder::build method from our motivating example is in Listing 7.

Listing 7 shows that the workload component implementation is indeed compact. The saxBuilderTest method prepares input data for the operation under test and executes the test loop. In the loop body, the input data is reset before executing the measured code (here a call to the SAXBuilder::build method), surrounded by the calls to SPLController::start and SPLController::end that take care of collecting measurements (these calls can be omitted if the loop body contains only measured code, in that case SPLController::needsMore collects measurements). The test loop termination condition is controlled by the testing framework through calls to the
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 7: Workload component for testing the SAXBuilder::build method

SPLController::needsMore method. The framework can thus take care of collecting sufficient amount of performance data, and also detect compilation events and discard warmup measurements.

The need for an explicit loop in the workload component deserves attention, because the compiler can, for example, move invariant code outside the loop, thus leaving the measured code smaller and possibly distorting the measurement results. One solution to this problem is implemented in the Java Microbenchmark Harness (Oracle 2014), which encloses the measured code in a standalone function and instructs the virtual machine to avoid inlining the function (and therefore avoid optimizing across function boundaries). Although this solution can be enforced automatically by the testing framework, we have not adopted it yet, because it is not clear whether some optimizations across the measurement loop do not in fact make the measurements more realistic.

The workload component implementation from Listing 7 represents a compact and generic approach that fits most use cases, however, it does not explicitly support test code reuse. Some special situations do invite reuse—for example when comparing different versions of the same method to verify performance improvement or prevent performance regressions, or when comparing different implementations of the same interface methods, such as alternative implementations of collection interfaces.

For test code reuse, the testing framework provides a special mapping, where the (test-independent) generation of the workload is kept separate from the (test-specific) allocation and initialization of the measured object and the invocation of the measured operation. We illustrate the mapping on an example of a performance unit test that measures the time to locate a random integer in a collection. The workload component of the test is shown in Listing 8. The individual factory* methods allocate and initialize the measured collection object using the prepare method. The workload generation,
// Fills a collection of given size with integers.
Object prepare(Collection<Integer> col, int size) {
    for (int i = 0; i < size; i++) {
        col.add(i);
    }
    return col;
}

// Constructs a LinkedList instance to be measured.
Object factoryLinkedList(int size) {
    return prepare(new LinkedList<Integer>(), size);
}

// Constructs an ArrayList instance to be measured.
Object factoryArrayList(int size) {
    return prepare(new ArrayList<Integer>(), size);
}

// Prepares an argument list that can be used
// in subsequent call to Collection::contains.
void randomSearch(MethodArguments args, int size) {
    args.set(0, Random.nextInt(size));
}

Listing 8: Workload component of a performance unit test with separate instance creation and workload generation methods

arrayList := ArrayList.contains[randomSearch, factoryArrayList]
linkedList := LinkedList.contains[randomSearch, factoryLinkedList]
for i in {10, 100, 500, 1000}
    linkedList(i) >= arrayList(i)

Listing 9: Decision component binding the instance creation and workload generation methods. The component is an SPL formula similar to those in Listings 4 and 5

implemented by the randomSearch method, prepares a random integer as the sole argument for invoking the measured method, Collection::contains. The individual methods are bound together in an SPL formula representing the decision component of the test, shown in Listing 9—for each method referenced by the SPL formula, both the object factory and the workload generation methods are specified.

Compared to the workload component shown in Listing 7, 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 multiple unit test components. On the other hand, a performance unit test employing this separation can be easily extended to include additional types—here for example instances of the Vector class—without disrupting the existing test code base.

Unlike a functional unit testing framework, which executes all tests in a single virtual machine, the performance unit testing framework executes each performance test in
a new virtual machine. This prevents pollution of the virtual machine by the activity of the previously executed tests, which may influence the performance measurement. Code reuse in the workload component therefore occurs at source level, but each test executes the reused code anew at runtime.

### 4.4 Performance testing framework

To integrate SPL-based performance testing into software development, we have implemented an automated testing framework for Java. Similar to JUnit, we provide a tool which scans a given project to discover performance tests, exercises the methods referenced in SPL formulas using the specified workloads, collects performance data needed to evaluate the SPL formulas, and produces a test report. The report is an HTML document which provides both a quick overview and a detailed information for each test, including visualization of the measurements.

Our implementation of the performance testing framework includes additional features not presented here, such as support for the Git and Subversion version control systems, and integration with Eclipse and Hudson. Some of the features described here are necessarily a work in progress, evolving as we gather more experience with applying SPL-based performance testing in real-world scenarios. The framework is available as open source at SPL Tool (2013).

We have also developed a proof-of-concept SPL-based performance testing tool for C# on the .Net platform (Trojánek 2012). Compared to Java, the advanced features of the C# language enable better integration of SPL with code.

### 5 Overall evaluation

Central to our work is the SPL formalism, which introduces the formal language used to express performance requirements, and multiple interpretations used to evaluate measurements collected in different contexts. Although some properties of the SPL formalism have been derived formally, the evaluation ultimately requires real measurements collected on real platforms—such measurements can violate the design assumptions in many ways, each with a different impact on the eventual evaluation accuracy.

The evaluation has four parts. Section 5.1 examines how well the individual interpretations tolerate incidental differences in performance. Section 5.2 evaluates the sensitivity to actual performance differences. Section 5.3 analyzes unit test portability, and Sect. 5.4 illustrates work with real developer assumptions.

For the evaluation, we return to our motivation example, which uses the JDOM software project (JDOM Library 2013). To identify realistic measurement locations—that is, software components where the developers are evidently concerned about performance—we look for keywords such as “performance”, “refactor”, “improve” or “faster” in the commit log (hunterhacker/jdom 2013) and implement performance unit tests that measure performance at the associated locations. We focus on the SAX builder and the DOM converter as two essential high-level components of JDOM, and on the Verifier class as one low-level component already introduced in our motivation.
example. In all, we collect 102 method performance measurements at various locations across 46 commits, which we use to assess the interpretations from Sect. 3.

In the experiments presented throughout the paper, we use the following platforms (multiple platforms are used to evaluate portability and reduce chance of inadvertent platform bias):

- 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.

5.1 Incidental performance differences

A performance unit test assesses differences in observed performance. Such differences can include incidental fluctuations outside experiment control, which the test should tolerate rather than report. As explained in Sect. 3.4, a major source of incidental performance differences are the changing conditions between runs. We therefore evaluate how the individual interpretations tolerate such differences.

For each of the measured methods, we collect measurements in multiple runs and use the individual SPL interpretations to decide whether the measured performance differs between runs. Any such difference is by definition incidental. In detail, for a set of measurements $M$ of method $m$, interpretation $i$, significance $\alpha$ and run count $r$:

1. Using random sampling with replacement, we select $M_a \subseteq M$ and $M_b \subseteq M$ of $r$ runs each. Sets $M_a$ and $M_b$ represent the observations of the same method performance in different experiments, denoted $P(a)$ and $P(b)$.
2. We use the interpretation $i$ to decide whether $P(a) = p(x, x) P(b)$ at significance $\alpha$.

We repeat the computation multiple times to estimate the probability of the individual decisions. The computation is performed for 102 methods, sampling from 58 runs for each method, each run collects at most 20 min or 30,000 observations, of which half is discarded as warmup.

We first analyze the results for the quick mean-value interpretation from Sect. 3.2, plotted on Fig. 6 for $\alpha = 0.05$ and Fig. 7 for $\alpha = 0.001$. The leftmost column of both figures illustrates that performance differences due to condition changes between runs are indeed statistically significant. When comparing performance based on measurements from one run, $r = 1$, the quick mean-value interpretation considers performance
Fig. 6 Probability of reporting an incidental performance difference for individual methods with quick mean-value interpretation at $\alpha = 5\%$. The measurements come from the JDOM library, executing on Platform Alpha of the same method in different runs as different for most methods and runs. Even when comparing measurements from more runs, $r \in \{10, 20, 50\}$, the situation is not much better. Clearly, the quick mean-value interpretation does not tolerate condition changes between runs well.

To improve practical utility of the quick mean-value interpretation, we modify it to tolerate performance differences of given size, expressed as tolerance percentage in Figs. 6 and 7. The results show the limits of the quick mean-value interpretation. When comparing performance based on measurements from one run, $r = 1$, a 1% tolerance still results in significant probability of reporting an incidental difference. A 5% tolerance pushes the probability of reporting an incidental difference below 20% for most methods, more so when the interpretation compares measurements from more runs.

The results in Figs. 6 and 7 are also important in that they reveal the sensitivity limits of any interpretation that relies on performance measurements from one run. In our experiments, condition changes between runs frequently introduce statistically significant performance differences of as much as 5% in magnitude. To recognize these differences as incidental, an interpretation either needs to consider measurements from more runs, or to tolerate any differences of similar magnitude.

Figures 8 and 9 present similar evaluation for the two interpretations that explicitly consider multiple runs, the parametric mean-value interpretation from Sect. 3.5 and the non-parametric mean-value interpretation from Sect. 3.6. Both interpretations compare data from one or five runs, $r \in \{1, 5\}$, and rely on a history from 10 or 50 runs to estimate the $\rho^2$ variance or the $B^*$ distribution, respectively.
Fig. 7 Probability of reporting an incidental performance difference for individual methods with quick mean-value interpretation at $\alpha = 0.1\%$. The measurements come from the JDOM library, executing on Platform Alpha.

The figures show that both interpretations keep the probability of reporting an incidental difference low, at the cost of needing a history of runs to estimate the magnitude of typical incidental difference.

To summarize, the evaluation shows that the quick mean-value interpretation can be used to detect performance differences on the order of percents with one or few runs, but must be set to tolerate smaller performance differences lest incidental differences due to condition changes between runs are also reported. Once a history of runs from similar measurements is available, the parametric and non-parametric mean-value interpretations can detect performance differences with no artificial tolerance threshold.

5.2 Performance difference sensitivity

Tolerance to incidental performance differences is directly related to performance test sensitivity. Next, we therefore evaluate how quickly the individual interpretations recognize a gradually increasing performance difference that is not incidental.

A rigorous evaluation of sensitivity to realistic performance differences is complicated by lack of reliable ground truth data to evaluate against. Ideally, we would identify examples of performance differences from real software development and evaluate whether the interpretations recognize the examples reliably. However, if we identify the examples using common statistical approaches, we are caught in a circular evaluation where we evaluate the interpretations, which are based on common
**Fig. 8** Probability of reporting an incidental performance difference for individual methods with parametric and non-parametric mean-value interpretations at $\alpha = 5\%$. The measurements come from the JDOM library, executing on Platform Alpha.

**Fig. 9** Probability of reporting an incidental performance difference for individual methods with parametric and non-parametric mean-value interpretations at $\alpha = 0.1\%$. The measurements come from the JDOM library, executing on Platform Alpha.
statistical approaches, only on examples identified by the same or similar approaches. And if we identify the examples manually, we lose the more challenging examples where the magnitude of the differences is close to variance, because those examples tend to escape manual identification.

In our evaluation, we utilize the fact that the performance of many JDOM components depends on the size of the data being processed, and introduce performance differences by manipulating this data accordingly. On the upside, we avoid artificial manipulation with measurements such as was used in Sect. 3.2. On the downside, we must estimate the connection between the data size and the method performance.

We construct the evaluation scenario around a method that builds a DOM data structure from an XML input. 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 $P(s) \geq_{p(x,x)} P(b)$ and $P(s) \leq_{p(x,x)} P(b)$ at significance $\alpha$.

We repeat the steps enough times to estimate the probability of the individual decisions. The results are available in Figs. 10 and 11, each run collects $o = 20,000$ observations after a warmup of 40,000 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). For each input size, 10 runs were available for random sampling. A history of 50 runs was used to derive the $\rho^2$ variance and the $B^*$ distribution.

To interpret the results in Figs. 10 and 11, 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. As in the previous section, such performance difference is incidental. Again, we see that the quick mean-value interpretation is incorrectly rejecting the null hypothesis very often regardless of the number of runs used. In contrast, the parametric and non-parametric mean-value interpretations rarely report an incidental difference, especially at higher significance.

Next, we look on the measurements with non-zero file size difference. To provide a rough estimate of how the file size difference is reflected in performance, it takes on average 9.04 ms to execute the method for the largest input and 7.81 ms for the smallest input. Assuming roughly linear dependency, 1% change in the file size corresponds to 1.3% change in the execution time.

An ideal interpretation would reject $P(s) \geq_{p(x,x)} P(b)$, recognizing that there actually is a performance difference; it would also never reject $P(s) \leq_{p(x,x)} P(b)$,
recognizing the performance difference reliably. We see that the quick mean-value interpretation is very willing to identify performance differences, however, it is not really reliable until the difference in file size reaches about 4% or the number of runs is sufficient.

When used on a small number of runs, the parametric and non-parametric mean-value interpretations begin to identify performance differences of about 4%, that is when the file size difference reaches about 3%. When given only one run to consider, both interpretations recognize a performance difference of about 7% about half of the
Fig. 11 Sensitivity to input size change for combinations of interpretation and run count at $\alpha = 0.1\%$. The measured method is `SAXBuilder::build`, used to build a DOM tree from a byte array stream, from the JDOM library, executing on Platform Alpha.

Time at lower significance, or a performance difference of about 12% about half of the time at higher significance.

With higher number of runs, the parametric and non-parametric mean-value interpretations start recognizing performance differences as small as 0.2%. Both interpretations almost never report an incidental difference.

To summarize, a reliable evaluation of performance differences requires either that the differences are large or that the measurements use more runs. In practical settings,
new runs would be measured and assessed as machine time permits. The quick mean-value interpretation can serve for initial recognition of large performance differences when no measurement history exists. The parametric and non-parametric mean-value interpretations can refine the assessment when more measurements become available, obtaining high sensitivity.

We also observe that although the computation of all the interpretations is linear in the number of samples and observations, the non-parametric mean-value interpretation takes by far the longest time to compute in practice, due to the computationally intensive Monte-Carlo simulation.

5.3 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. We evaluate portability by looking at how the measurements related in each unit test differ between three very different experiment platforms, here labeled Bravo, Charlie, Delta.

We define a metric that describes the relative difference between measurements related in 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 12 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.

![Histogram of portability metric values](image)

Fig. 12  Histogram of portability metric values
5.4 Developer assumptions

So far, our evaluation has focused on the technical properties of the individual interpretations when applied on real measurements. Next, we look at the fitness for purpose in a more general sense, investigating whether the SPL formalism can indeed help express and validate developer assumptions about performance.

Our investigation is based on a retrospective case study with the JDOM software project. As noted earlier, we have traversed the JDOM commit log for keywords indicating concerns about performance. Based on the commit log records, we have implemented performance unit tests that express the recorded assumptions. We have thus obtained 103 performance comparisons over 102 measurements in 58 tests across 46 commits, which serve as examples of tests that the developers might have considered implementing themselves.

All assumptions were implemented using simple SPL formulas that related performance of multiple versions of the same method or performance of similar methods. The tests were typically used to confirm that a particular commit increased performance of a component subjected to optimization, or that a particular commit did not decrease performance of unrelated components. We are therefore satisfied that the SPL formulas provide sufficient flexibility to express common developer assumptions. We select three examples for illustration, with measurements from Platform Bravo given in Table 1.

5.4.1 Case I: Negative improvement

The first example shows a situation where the developers believed a change will improve performance significantly, but the opposite is now true. The change was

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Commit</th>
<th>Median (ms)</th>
<th>10% Q (ms)</th>
<th>90% Q (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAXBuilder::build</td>
<td>6a49ef6</td>
<td>9.2</td>
<td>9.1</td>
<td>9.3</td>
</tr>
<tr>
<td>SAXBuilder::build</td>
<td>4e27535</td>
<td>11.2</td>
<td>11.1</td>
<td>11.3</td>
</tr>
<tr>
<td>DOMBuilder::build</td>
<td>6a49ef6</td>
<td>171.4</td>
<td>170.9</td>
<td>217.0</td>
</tr>
<tr>
<td>DOMBuilder::build</td>
<td>4e27535</td>
<td>206.8</td>
<td>205.0</td>
<td>256.0</td>
</tr>
<tr>
<td>Verifier::checkAttributeName</td>
<td>500f9e5</td>
<td>22.5</td>
<td>22.5</td>
<td>22.6</td>
</tr>
<tr>
<td>Verifier::checkAttributeName</td>
<td>4ad684a</td>
<td>20.0</td>
<td>19.9</td>
<td>20.1</td>
</tr>
<tr>
<td>Verifier::checkCharacterData</td>
<td>500f9e5</td>
<td>72.3</td>
<td>56.0</td>
<td>74.0</td>
</tr>
<tr>
<td>Verifier::checkCharacterData</td>
<td>4ad684a</td>
<td>65.8</td>
<td>46.9</td>
<td>65.9</td>
</tr>
<tr>
<td>Verifier::checkElementName</td>
<td>500f9e5</td>
<td>30.6</td>
<td>28.0</td>
<td>30.7</td>
</tr>
<tr>
<td>Verifier::checkElementName</td>
<td>4ad684a</td>
<td>25.8</td>
<td>23.7</td>
<td>25.9</td>
</tr>
<tr>
<td>Verifier::checkAttributeName</td>
<td>e069d4c</td>
<td>21.4</td>
<td>21.3</td>
<td>21.5</td>
</tr>
<tr>
<td>Verifier::checkAttributeName</td>
<td>1a05718</td>
<td>25.2</td>
<td>25.2</td>
<td>25.4</td>
</tr>
<tr>
<td>Verifier::checkElementName</td>
<td>e069d4c</td>
<td>31.6</td>
<td>28.6</td>
<td>31.7</td>
</tr>
<tr>
<td>Verifier::checkElementName</td>
<td>1a05718</td>
<td>42.6</td>
<td>41.3</td>
<td>42.8</td>
</tr>
</tbody>
</table>
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” (hunterhacker/jdom 2013).

To test the assumption, we have implemented a unit test that compared the performance of the SAXBuilder::build and DOMBuilder::build methods, used to build a DOM tree from various input representations, before and after the change, on a selection of XML samples. Table 1 shows the performance observed by the unit test. Instead of the expected performance improvement, the version after the change executes about 20% slower than before.

5.4.2 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 (hunterhacker/jdom 2013). These are the improvements described in our motivation example.

This time, our unit test compared the performance of the checkElementName, checkAttributeName and checkCharacterData methods of the Verifier class before and after the change, on a selection of valid names and text data. The results in Table 1, which compare the performance with preceding commit 500f9e5, indicate the method execution times have decreased by around 10–15%.

5.4.3 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” (hunterhacker/jdom 2013). 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. The execution times refer to multiple consecutive invocations, because one invocation would be too short to measure accurately.

In total, our 58 tests have identified 6 cases where the developer assumptions do not hold. For interested readers, we have made the complete source of the unit tests and the complete measurement results available on the web at SPL Tool (2013). These results are indeed positive, however, they also need more discussion due to the retrospective context.
Evidently, the fact that we have observed certain performance now does not mean the same performance was observed when the relevant changes were committed. The development platform used at commit time was most likely different from the experiment platform used here, and the same goes for the expected workload and other relevant factors. We do observe, however, that if the commits were augmented with performance unit tests, the expected workload and the performance assumptions would have been explicitly recorded, making it easier to decide whether the currently violated assumptions deserve developer attention.

The retrospective context also made the unit test implementation more complicated, giving rise to backwards compatibility problems and other practical issues. Together, these make the case study rather expensive in terms of effort needed to address retrospection specific problems—we therefore believe additional fitness-for-purpose evaluation should come from live application.

6 Related work

The overview of related research starts with approaches for capturing of performance assumptions. Next, we briefly overview the performance unit testing tools in Java, and finally we connect our approach to work on continuous testing and performance regression detection.

PSpec (Perl and Weihl 1993) 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 Vetter and Worley (2002). 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 (Reynolds et al. 2006) 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, Liu et al. (2008) describe 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, Tjang et al. (2009) employ 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 (Clark 2013) and ContiPerf (Bergmann 2013) are notable extensions of the JUnit (2013) functional unit testing framework. Both tools use absolute time limits to specify performance constraints evaluated by the tests, which makes detecting smaller performance changes difficult—small changes require using tight timing bounds in test code. 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.

Performance unit testing is necessarily connected to exploratory performance analysis, where specialized benchmark tools such as the Java Microbenchmarking Harness (Oracle 2014) render invaluable service in supporting benchmark implementation. In general, avoiding benchmark implementation issues that distort measurements remains a difficult task that hinders performance unit testing.

Also related to our work are projects for continuous testing. Among those, the Skoll Project (Porter et al. 2007) 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 (Oliveira et al. 2013) 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. (2010) stress the need for automated approaches 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. (2013) 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 performance testing itself, Heger et al. (2013) tackle the problem of identifying the change that caused a performance regression. 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.
7 Conclusion

Pursuing the goal of applying unit tests for performance testing activities, we have presented 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, the 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. The evaluation of the SPL formulas relies on formally defined logic interpretations that adopt statistical hypothesis testing to cope with practical measurements. Three such interpretations were developed and evaluated—one for simple scenarios where large performance differences need to be detected quickly, 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.

We have evaluated the ability of the SPL formulas to express real developer assumptions by retrospectively implementing and evaluating performance tests based on the development history of the JDOM project. We have focused especially on the properties of the interpretations, demonstrating the limits on sensitivity to performance differences with real measurements collected on real platforms—a necessary complement to analyzing the interpretations formally. We show that the interpretations are capable of recognizing performance differences below 1% even when incidental performance fluctuations in the range of 1–5% distort the measurements.

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 SPL Tool (2013).

Acknowledgments We gratefully acknowledge the contribution of our colleagues František Haas, Jaroslav Kezník, Alena Koubková, Martin Lacina and Andrej Podzimek, who were authors of the conference papers that this article combines and extends. Portions of this work were previously published under the copyright of the corresponding publishers and partially supported by the EU Project 257414 ASCENS and the GAČR project P202/10/3042 FERDINAND. This work was partially supported by the COST CZ (LD) Project LD15051 and by the Charles University institutional funding (SVV).

Appendix

Expected-value-based interpretation

The following lemma and theorem show that the interpretation of \( =_p \) and \( \leq_p \), as defined by Definition 6, is consistent with axioms (4) and (5). The consistency with other axioms trivially results from the assumption of total ordering on \( W \).

Lemma 1 Let \( X, Y : \Omega \to \mathbb{R}^+ \) be random variables, and \( t_l, t_r, t_x, t_y : \mathbb{R}^+ \to \mathbb{R}^+ \) be performance observation transformation functions. Then the following holds:
\( \forall o \in \mathbb{R}^+ : tl(o) \leq tr(o) \rightarrow E(tl(X)) \leq E(tr(X)) \)

\( (E(tx(X)) \leq E(ty(Y)) \land E(ty(Y)) \leq E(tx(X))) \)

\( \leftrightarrow E(tx(X)) = E(ty(Y)) \)

**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 \( X \) and \( Y \) to be continuous random variables. The proof would be the same for discrete random variables, except with a sum in place of the integral.

**Theorem 1** The interpretation of performance relations \( \leq_p \) and \( =_p \), as given by Definition 6, is consistent with axioms (4) and (5).

**Proof** The proof of the theorem naturally follows from Lemma 1 by substituting \( P_M(x_1, \ldots, x_m) \) for \( X \) and \( P_N(y_1, \ldots, y_n) \) for \( Y \).

**Quick mean-value interpretation**

We need to show that the quick mean-value interpretation of SPL is consistent with axioms (4) and (5). Briefly, the interpretation relies on the Welch’s \( t \) test, which rejects with significance level \( \alpha \) the null hypothesis \( \overline{X} = \overline{Y} \) against the alternative hypothesis \( \overline{X} \neq \overline{Y} \) if

\[
\left| \frac{\overline{X} - \overline{Y}}{\sqrt{\frac{S^2_X}{V_X} + \frac{S^2_Y}{V_Y}}} \right| > t_{v,1-\alpha/2}
\]

and rejects with significance level \( \alpha \) the null hypothesis \( \overline{X} \leq \overline{Y} \) against the alternative hypothesis \( \overline{X} > \overline{Y} \) if

\[
\frac{\overline{X} - \overline{Y}}{\sqrt{\frac{S^2_X}{V_X} + \frac{S^2_Y}{V_Y}}} > t_{v,1-\alpha}
\]

where \( V_i \) is the sample size, \( S^2_i \) is the sample variance, \( t_{v,\alpha} \) is the \( \alpha \)-quantile of the Student’s distribution with \( v \) levels of freedom, with \( v \) computed as follows:
\[ v = \frac{\left( \frac{S_X^2}{V_X} + \frac{S_Y^2}{V_Y} \right)^2}{\frac{S_X^4}{V_X^2(V_X-1)} + \frac{S_Y^4}{V_Y^2(V_Y-1)}} \]

**Theorem 2** The interpretation of relations \( =_p \) and \( \leq_p \), as given by Definition 8, is consistent with axiom (4) for a given fixed experiment \( E \).

**Proof** For sake of brevity, we 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

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

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

Since the denominator is a positive number, the whole fraction is non-positive. However, the right hand side \( t_{v,1-\alpha} \) 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 3** The interpretation of relations \( \leq_p \), and \( =_p \), as given by Definition 8, is consistent with axiom (5) for a given fixed experiment \( E \).

**Proof** For sake of brevity, we 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 similarly defined for \( N \).

By interpreting axiom (5) according to Definition 6, we get the following statements:

\[ P_M(x_1, \ldots, x_m) \leq_p (tm, tn) P_N(y_1, \ldots, y_n) \]
\[ \iff -t_{v_{M,N,1-\alpha}} \leq \frac{\overline{M} - \overline{N}}{\sqrt{\frac{S^2_M}{V_M} + \frac{S^2_N}{V_N}}} \]
\[ P_N(y_1, \ldots, y_n) \leq_p (tn, tm) P_M(x_1, \ldots, x_m) \]
\[ \iff t_{v_{N,M,1-\alpha}} \leq \frac{\overline{N} - \overline{M}}{\sqrt{\frac{S^2_N}{V_N} + \frac{S^2_M}{V_M}}} \]
\[ P_M(x_1, \ldots, x_m) =_p (tm, tn) P_N(y_1, \ldots, y_n) \]
\[ \iff -t_{v_{M,N,1-\alpha}} \leq \frac{\overline{M} - \overline{N}}{\sqrt{\frac{S^2_M}{V_M} + \frac{S^2_N}{V_N}}} \]

90
Thus, we need to show that
\[
\frac{\bar{M} - \bar{N}}{\sqrt{\frac{s_M^2}{V_M} + \frac{s_N^2}{V_N}}} \leq t_{v_{M,N},1-\alpha} \land \frac{\bar{N} - \bar{M}}{\sqrt{\frac{s_N^2}{V_N} + \frac{s_M^2}{V_M}}} \leq t_{v_{N,M},1-\alpha}
\]
\[\iff -t_{v_{M,N},1-\alpha} \leq \frac{\bar{M} - \bar{N}}{\sqrt{\frac{s_M^2}{V_M} + \frac{s_N^2}{V_N}}} \leq t_{v_{N,M},1-\alpha}\]

This holds, because \(v_{M,N} = v_{N,M}\) and thus
\[
\frac{\bar{N} - \bar{M}}{\sqrt{\frac{s_N^2}{V_N} + \frac{s_M^2}{V_M}}} \leq t_{v_{N,M},1-\alpha} \iff -t_{v_{X,Y},1-\alpha} \leq \frac{\bar{M} - \bar{N}}{\sqrt{\frac{s_M^2}{V_M} + \frac{s_N^2}{V_N}}} \leq t_{v_{N,M},1-\alpha}
\]

Note that, as indicated in Sect. 2, transitivity (i.e., \((P_X(\ldots) \leq_{p_{(x,y)}} P_Y(\ldots) \land P_Y(\ldots) \leq_{p_{(y,z)}} P_Z(\ldots)) \rightarrow P_X(\ldots) \leq_{p_{(x,z)}} P_Z(\ldots))\) 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 \(\alpha = 0.05\): \(O_{P_X} = \{2, 4\}, O_{P_Y} = \{-1, 1\}, O_{P_Z} = \{-4, -2\}\).

### Parametric mean-value interpretation

We need to show that the parametric mean-value interpretation of SPL is consistent with axioms (4) and (5).

**Theorem 4** The interpretation of relations \(\leq_p, \text{ and } =_p\), as given by Definition 9, is consistent with axioms (4) and (5) for a given fixed experiment \(E\).

**Proof** The proof is very similar to the proofs of Theorems 2 and 3. To prove consistency with axiom (4), it is necessary to show that
\[
M_{il} - M_{tr} \leq z_{(1-\alpha)} \sqrt{\frac{o_M R_{il}^2 + S_{il}^2}{r_M o_M} + \frac{o_M R_{tr}^2 + S_{tr}^2}{r_M o_M}}
\]

where \(M_{il}\) denotes the sample mean of run means of \(tl(P_{i,j}^i(x_1, \ldots, x_m))\), \(S_{il}^2\) and \(R_{il}^2\) denote for the same the average of variances of samples per run and the variance of run means, respectively. \(M_{tr}, S_{tr}^2, \text{ and } R_{tr}^2\) are similarly defined for \(tr(P_{i,j}^i(x_1, \ldots, x_m))\). This inequality is obviously true because \(M_{il} \leq M_{tr}\) 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 9 with the second part of the definition.
Non-parametric mean-value interpretation

To show that the non-parametric mean-value interpretation of SPL is consistent with axioms (4) and (5), we assume pure bootstrap without the Monte-Carlo simulations. We further restrict $t_l, t_r$ to be linear functions and assume $\alpha \leq 0.35$ as reasonable practical restrictions.

**Theorem 5** Assuming that $t_l$ and $t_r$ are linear functions, the interpretation of relations $\leq_p$, and $=_p$, as given by Definition 10, with $M$ and $N$ used for $X$ and $Y$, is consistent with axiom (4) for a given fixed experiment $E$ and $\alpha \leq 0.35$.

**Proof** To prove the consistency of $\leq_p$ with axiom (4), we have to show that the following condition holds

$$B^*_X^{t_l,r,o \rightarrow X_{t_l},r,o}(X_{t_l} - X_{t_r}) \leq 1 - \alpha$$

The left-hand side of the expression above, i.e., $B^*_X^{t_l,r,o \rightarrow X_{t_l},r,o}(X_{t_l} - X_{t_r})$, corresponds to

$$\frac{1}{|S|^2} \sum_{s_{tl} \in S} \sum_{s_{tr} \in S} \mathbf{1} \left( \frac{1}{r^o} \sum_{i=1}^{r^o} \sum_{j=1}^{o} (tl(P^s_{X}^{i,j}) - tr(P^s_{X}^{i,j})) \right)$$

$$-(X_{t_l} - X_{t_r}) \leq X_{t_l} - X_{t_r}$$

where $\mathbf{1}$ is an indicator function; and $s_{tl}, s_{tr} : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \times \mathbb{N}$ are selector functions that determine observations to be drawn with replacement from observations $P^{i,j}_X(x_1, \ldots, x_m)$. We denote $S$ as the set of all selectors for the observations.

Since $t_l(x) = a_{tl}x + b_{tl}$, $t_r(x) = a_{tr}x + b_{tr}$ are linear functions, we can rewrite the condition (6) as

$$\frac{1}{|Z|^2} \sum_{i=1}^{|Z|} \sum_{j=1}^{|Z|} \mathbf{1} \left( a_{tl}z_i - a_{tr}z_j + b_{tl} - b_{tr} \leq 2(a_{tl} - a_{tr})X + 2(b_{tl} - b_{tr}) \right) \leq 1 - \alpha$$

where $Z$ is a multiset defined as

$$Z = \left\{ \frac{1}{r^o} \sum_{i=1}^{r^o} \sum_{j=1}^{o} P^s_{X}^{i,j} \, \bigg| \forall s \in S \right\}$$

and $z_i \in Z$ is a sorted sequence (i.e., $z_1 \leq \cdots \leq z_{|Z|}$) consisting of all items in $Z$.

To prove (6), let us assume that it does not hold. Then the $1 - \alpha$ quantile (denoted $q_{1-\alpha}$) of $B^*_X^{t_l,r,o \rightarrow X_{t_l},r,o}$ would have to be less than $2(a_{tl} - a_{tr})X + 2(b_{tl} - b_{tr})$. We show that this situation cannot happen for $\alpha = 0.35$ and consequently for any
\( \alpha \leq 0.35 \) (because \( q_{1-0.35} \leq q_{1-\alpha} \) for \( \alpha \leq 0.35 \)). For the sake of contradiction, we posit

\[
q_{1-0.35} < 2(a_{tl} - a_{tr})X + 2(b_{tl} - b_{tr})
\] (7)

Rewriting the sample mean \( \bar{X} \) as an average of values \( z_i \), we get

\[
q_{1-0.35} < 2(a_{tl} - a_{tr}) \frac{\sum_{i=1}^{\lfloor |Z|+1/2 \rfloor} z_i}{|Z|} + 2(b_{tl} - b_{tr})
\] (8)

Next, we approximate the quantile \( q_{1-0.35} \) with \( (a_{tl} - a_{tr}) \frac{z_{\lfloor (|Z|+1)/2 \rfloor}}{\lfloor |Z|+1 \rfloor} + b_{tr} - b_{tl} \leq q_{1-0.35} \)

This holds because the partial ordering of the differences \( a_{tl}z_i - a_{tr}z_j \) forms a lattice as shown in Fig. 13 (the arrows point from smaller values to larger ones). Thanks to this partial order, at least 35 of the differences \( a_{tl}z_i - a_{tr}z_j \) are not smaller than \( (a_{tl} - a_{tr}) \frac{z_{\lfloor (|Z|+1)/2 \rfloor}}{\lfloor |Z|+1 \rfloor} \) (the values delineated in the figure by the dashed and dotted red line, which is formed by a rectangular area of 1/4 of all values above the center and a triangular area of 1/8 of all values left of the center). Consequently \( (a_{tl} - a_{tr}) \frac{z_{\lfloor (|Z|+1)/2 \rfloor}}{\lfloor |Z|+1 \rfloor} + b_{tr} - b_{tl} \) has to be less or equal to \( q_{1-0.35} \). We thus rewrite (8) as
\[(a_{tl} - a_{tr})z_{\lceil \frac{|Z|+1}{2} \rceil} + b_{tl} - b_{tr} \leq q_{1-0.35} < 2(a_{tl} - a_{tr})\frac{\sum_{i=1}^{|Z|} z_i}{|Z|} + 2(b_{tl} - b_{tr})\]  
(9)

Since \(tr(o) \geq tl(o)\), we have \(a_{tl} - a_{tr} \leq 0\) and \(b_{tl} - b_{tr} \leq 0\). After removing \(b_{tl} - b_{tr}\), multiplying the inequality by \(|Z|/(a_{tl} - a_{tr})\) and approximating the sum on the right hand side of (9) we get the expected contradiction, which completes the proof for \(\leq p\).

\[|Z|z_{\lceil \frac{|Z|+1}{2} \rceil} > 2 \sum_{i=1}^{|Z|} z_i \geq 2 \left\lfloor \frac{|Z| + 1}{2} \right\rfloor z_{\lceil \frac{|Z|+1}{2} \rceil} \geq (|Z| + 1)z_{\lceil \frac{|Z|+1}{2} \rceil}\]

Note that \((a_{tl} - a_{tr})\) must be negative and \(z_i\) is positive because it averages performance observations, which are positive themselves. For the case \(a_{tl} = a_{tr}\), the contradiction in (7) is already clear from (9).

The proof for \(= p\) is similar. It has to additionally show that

\[\alpha \leq B_{X_{tl},o-X_{tr},o}^+ (X_{tl} - X_{tr})\]

which can be done following the same reasoning as already shown in the proof above.

\[\square\]

**Theorem 6** The interpretation of relations \((\leq p, and = p)\), as given by Definition 10, is consistent with axiom (5) for a given fixed experiment \(E\).

**Proof** Similar to the proof of consistency with axiom (5) in Theorem 4, the consistency here also directly comes from comparing the first part of Definition 10 with the second part of the definition.

**References**

Bergmann, V.: ContiPerf 2 (2013) [http://databene.org/contiperf.html](http://databene.org/contiperf.html)


JUnit Tool, Apr (2013). http://junit.org


Trojánek, T.: Capturing performance assumptions using stochastic performance logic. Master’s Thesis, Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic (2012)


Welch, B.L.: The generalization of “Student’s” problem when several different population variances are involved. Biometrika 34(1/2), 28–35 (1947)

Analysis of Overhead in Dynamic Java Performance Monitoring

Vojtěch Horký,
Jaroslav Kotrč,
Peter Libič,
Petr Tůma

Contributed paper at the 7th ACM/SPEC International Conference on Performance Engineering (ICPE 2016).


The original version is available electronically from the publisher’s site at http://dx.doi.org/10.1145/2851553.2851569
Analysis of Overhead in Dynamic Java Performance Monitoring

Vojtěch Horký, Jaroslav Kotrč, Peter Libič, Petr Tůma
Department of Distributed and Dependable Systems
Faculty of Mathematics and Physics, Charles University
Malostranské náměstí 25, Prague 1, 118 00, Czech Republic
first.last@d3s.mff.cuni.cz

ABSTRACT
In production environments, runtime performance monitoring is often limited to logging of high level events. More detailed measurements, such as method level tracing, tend to be avoided because their overhead can disrupt execution. This limits the information available to developers when solving performance issues at code level.

One approach that reduces the measurement disruptions is dynamic performance monitoring, where the measurement instrumentation is inserted and removed as needed. Such selective monitoring naturally reduces the aggregate overhead, but also introduces transient overhead artefacts related to insertion and removal of instrumentation. We experimentally analyze this overhead in Java, focusing in particular on the measurement accuracy, the character of the transient overhead, and the longevity of the overhead artefacts.

Among other results, we show that dynamic monitoring requires time from seconds to minutes to deliver stable measurements, that the instrumentation can both slow down and speed up the execution, and that the overhead artefacts can persist beyond the monitoring period.

Keywords
performance measurement overhead; dynamic instrumentation; Java

1. INTRODUCTION
Software performance is not only a common term, but also something of a misnomer, because it suggests performance is a property of software. In reality, software performance is a product of executing the software on a particular platform and neither the software nor the platform alone determines performance. This is also one of the reasons why performance monitoring is used – by observing the actual performance, it takes into account the software, the platform and the workload, something that is difficult to do otherwise.

Technically, essential tasks of performance monitoring include data collection and data storage or data processing, or both. These tasks consume resources, giving rise to monitoring overhead. The overhead can easily range from units of percent – for example when monitoring selected methods in an enterprise benchmark application – to orders of magnitude – for example when collecting calling context profile in standard application benchmarks. This is obviously a practically significant factor.

Because the monitoring overhead depends on the amount of data collected, it can be reduced by collecting less data at fewer locations. Particularly interesting is dynamic monitoring, where individual components of the monitoring infrastructure are enabled and disabled, or even inserted and removed, to cater to changing monitoring demands. Dynamic monitoring support exists in many contexts, from operating systems to enterprise application monitoring frameworks.

An important influence on dynamic monitoring overhead is exerted by probes – data collection components that are inserted directly into the monitored application. Probes can be inserted either through static instrumentation, which happens before the monitored application is executed, or through dynamic instrumentation, which happens during execution. In the former case, the probe code is always in place and contains support for enabling or disabling data collection. In the latter case, the probe code is simply inserted or removed as needed. Dynamic instrumentation is technically more challenging, because it entails modifying an executing application, but also more attractive, because it carries the implied promise of achieving zero overhead when not collecting data.

In this paper, we focus on dynamic performance monitoring in the context of Java. Starting with version 1.6, Java provides a standard support for changing the code of an executing application through mechanisms called class redefinition and class retransformation. By operating on bytecode, these mechanisms are much more portable than dynamic instrumentation based on machine code manipulation, but also much less transparent where performance overhead is concerned. We address this issue by presenting an extensive overhead study focused particularly on dynamic performance monitoring in Java.

We conduct our overhead study in the broader context of our research on performance awareness. Our general goal is to provide developers with information on software performance that is timely and relevant – that is, presented at a
time and in a manner that makes it useful rather than distracting. Towards that goal, we have implemented a framework capable of both static and dynamic performance monitoring, which we use for example to answer performance related queries in the context of regression testing [3] or to provide performance information in software documentation during development [12]. Here, we therefore analyze the overhead of the framework.

The structure of the paper follows our main contributions. In Section 2, we describe our performance monitoring framework, with focus on dynamic instrumentation as the new feature. Section 3 contributes a detailed analysis of overhead sources specific to dynamic instrumentation. In Section 4, we present the experimental overhead evaluation itself. Related work discussion and concluding remarks close the paper.

2. MEASUREMENT FRAMEWORK

Figure 1 presents a high level architecture of the performance monitoring framework we use throughout this paper. The framework executes in two virtual machines – the data collection components reside in the same JVM as the measured application, the data storage and data processing components use a helper JVM. This helps minimize the framework footprint in the application JVM and provides the possibility of running the helper JVM on a separate host. It also matches the architecture of the underlying instrumentation framework we use, called DISL [21].

The framework uses the launcher component to perform the necessary initialization and set up the connection between the application JVM and the helper JVM. Once the application executes, the measurement coordination component decides when a measurement should start – depending on circumstances, this can be in response to an interactive developer request, favorable load conditions, or other triggers. The component uses the control connection to deliver the instrumentation request to the application JVM, where the transformation agent fires a class transformation request. The application JVM reacts by asking the DISL agent to transform the measured class, the DISL agent in turn uses the DISL framework in the helper JVM to perform the transformation – which in this case takes the form of inserting probe code. Once the probe code is inserted, it starts feeding measurements to the data transfer component, which uses the data connection to deliver the measurements to the helper JVM for processing. Similar process is used when removing probe code.

Listing 1 provides a compact pseudocode listing of the probe code. The code simply collects the time at the entry to and the exit from the measured method – the somewhat more complicated listing is due to the need to handle recursion. When the probe is called recursively, only the top level iteration is measured. The probe state is thread local, implemented using efficient thread local variables offered by DISL. This minimizes synchronization.

We omit other elements of the framework, which are not essential for the purpose of this paper. These include the ability to differentiate between invocations of the same method based on the actual argument values, and the applications for regression testing and documentation generation. For more details on the performance regression testing features, refer to [3], for performance documentation generation features, refer to [12]. The framework is available as open source at http://d3s.mff.cuni.cz/software/spl.

3. OVERHEAD SOURCE ANALYSIS

A characteristic feature of contemporary computing platforms is the potential for complex interactions across multiple levels of the hardware and software stack. Dynamic measurement instrumentation influences these interactions in many ways, with the collective impact on performance forming the observed measurement overhead. Here, we discuss the sources of measurement overhead relevant to Java-like platforms – that is, platforms with applications written in a high level language, garbage collected memory, dynamic class loading and just-in-time (JIT) compilation. The discussion steers mostly clear of technical detail, available in platform-specific sources such as [14].

3.1 Probe Presence

The instrumentation inserts probes directly into the application, to be executed just before and just after the measured application code. The probe code consumes processor resources just as the application code does, introducing execution overhead.

As illustrated on Listing 1, the probe code samples time. The part of the probe code situated between the sampling points and the application code of interest will be measured together, introducing systematic measurement error. With some simplifications, the error is likely to be additive and can be possibly compensated by calibration. The remaining probe code, which resides outside the sampling points, is not measured but still counts towards the application execution time.

The systematic measurement error can grow when the measured application code is called recursively. When this is the case, the error accumulates with the depth of the recursion and, except for the top level iteration, includes the entire probe code rather than just the part of the probe between the sampling points. A similar situation arises when multiple instrumented methods call each other.

When examined in detail, the execution overhead is further influenced by interactions inside the processor microarchitecture. The probe code may or may not cause or suffer
Thread local variable

recursion : map String to integer

Invocation local variables

entryTime : integer
exitTime : integer
name : String

advice at method entry

▷ Call is converted to constant by DiSL
▷ at class loading (weaving) time
name ← GetCurrentMethodName
INCREMENT(recursion[name])
▷ Time sampled close to real entry moment
entryTime ← GetCurrentTime

end advice

advice at method exit

▷ Time sampled close to real exit moment
exitTime ← GetCurrentTime
name ← GetCurrentMethodName
DECREMENT(recursion[name])
if at top level of recursion then
    SendMeasurement(name, entryTime, exitTime)
end if

end advice

Listing 1: Probe pseudocode for dynamic instrumentation.

relatively expensive events such as cache misses or branch prediction failures, whose occurrence depends on the interaction with the surrounding application code. In principle, applications that are particularly tightly tuned to the processor microarchitecture — such as numerical applications that rely on tiling to efficiently utilize caches [26] — may be disrupted significantly, however, such tight tuning is not common on platforms that do not expose memory layout to applications.

With both the application and the probe written in a high level language, control over the execution overhead is somewhat limited. Still, it is possible to minimize the overhead by structuring the probe code so that the sampling points are close to the measured application code and by avoiding potentially expensive constructs such as synchronization or polymorphic invocations. Ultimately, the overhead determines practical measurement granularity — if the overall disruption to application execution is to be reasonable, the measured application code should execute orders of magnitude longer than the probe code.

3.2 Code Manipulation

The code manipulation associated with inserting and removing probes also consumes resources. The instrumentation needs to parse the application class to be measured, insert the probe code, and have the virtual machine load the instrumented application class. In general, these are operations that are about as disruptive as other class loading activity.

As an important consequence, class manipulation during instrumentation may trigger JIT compilation. If some methods of the class were JIT compiled before instrumentation, then these compiled versions are discarded, and may be JIT compiled again after instrumentation. The impact may extend to methods of other classes whose compiled versions depend on the instrumented application class, leading to cascades of JIT compilations that reflect prior inlining decisions.

Depending on circumstances, the virtual machine may initiate JIT compilation immediately after loading the instrumented application class, at some later time, or even never. Until the JIT compilation completes, those methods whose compiled versions were discarded can execute less efficiently or even block, again in effect contributing to overhead. In general, it is not possible to tell whether some future JIT compilation will deliver a more efficient compiled version of a method, it is therefore not possible to minimize the impact on measurement simply by waiting for the compiled version. It is, however, possible to wait for JIT compilations that immediately follow instrumentation to finish — those JIT compilations should cover most hot code, where the impact on measurement is also most likely significant.

3.3 Code Optimization

The JIT compilation involves optimization decisions that may change with instrumentation. This is true even when the interaction between the probe code and the application code is kept to a minimum — most importantly, the very presence of the probe code influences the heuristics that drive method inlining. Although these heuristics may vary, they are likely to include a limit on the size of the inlined method. Inserting probe code increases code size and therefore reduces the chance of the measured methods being inlined.

Method inlining is an important optimization because it impacts the scope of most other optimizations — with JIT compilation working on methods as compilation and optimization units, inlining one method into another means the caller and the callee are optimized together. The impact of inlining on performance experiments was demonstrated in detail with JMH benchmarks [24] that can selectively disable method inlining to prevent interaction between the benchmark harness and the measured method [31]. In very general terms, we can assume that by reducing the chance of inlining, instrumentation reduces the opportunity for optimization. We can therefore expect instrumentation to introduce another systematic measurement error, due to observing possibly less optimized versions of the measured methods. Keeping probe code small, however, should make this error less likely.

The optimization decisions made during JIT compilation also depend on past application execution. Factors such as method invocation count, loop iteration count, or type variability are taken into account — it is therefore not guaranteed that the same method will be compiled in the same way at different moments in application execution. In particular, it is not guaranteed that a method will have the same compiled version after removing probes as it had before inserting probes.

3.4 Other Overhead Sources

Significant sources of measurement overhead are also associated with data storage. Whatever data a probe collects or aggregates needs to be stored in memory and then exported outside the measured application. The memory storage overhead begins with allocation — when using the application heap, additional allocations will either cause the heap...
to expand or the garbage collector to run more often [16, 17]. After allocation, storing data in memory consumes additional memory bandwidth, and export similarly incurs additional storage or network bandwidth. The magnitude of these effects grows with the data volume.

Because there is no principal difference between data storage overhead coming from dynamic measurement instrumentation and similar overhead from standard instrumentation or even application I/O, we do not analyze this overhead source further. A thorough analysis of the export overhead and the related measurement framework implementation issues can be found in [37].

4. EXPERIMENTAL EVALUATION

The analysis of potential overhead sources associated with dynamic measurement instrumentation directly translates into questions we want to answer using experimental evaluation:

Q1. Given that some part of the probe code is necessarily situated between the sampling points and the measured application code, what is the typical difference between the measured execution time and the actual execution time?

Q2. Does the difference between the measured execution time and the actual execution time remain stable?

Q3. Given that the measured code is potentially interacting with the probe code through code optimization decisions and other channels, is the execution time of the measured code different from the execution time with no measurement?

Q4. Given that the measured code is potentially compiled differently before and after measurement, does the execution time after measurement differ from the execution time before measurement?

Q5. What is the typical duration of JIT compilation associated with dynamic measurement instrumentation?

For completeness, we also want to answer the ever present question associated with instrumentation, even if there is no reason why the result should be significantly different from other instrumentation overhead studies:

Q6. What is the total overhead in terms of application performance that can be attributed to dynamic measurement instrumentation?

4.1 Overall Design

To answer the overhead related questions, we need to observe an application both with and without dynamic measurement instrumentation in place — in other words, we need independent observation capabilities that exist alongside the dynamic instrumentation. We employ static instrumentation deployed throughout the measured application to perform continuous measurement. From the perspective of the dynamic instrumentation, the static instrumentation is just a part of the measured application that provides baseline measurements, as outlined in Figure 2. Compared to Figure 1, the application is now augmented with the static probe code, which relies on the static measurement agent to record the baseline measurements in local storage. An experiment coordination component is introduced to direct when a measurement should start and stop, but the dynamic instrumentation remains otherwise unchanged.

With both static and dynamic instrumentation available, we pretend that we perform dynamic measurements on an application that also produces baseline measurements for comparison. We structure the experiment to model a situation where a developer chooses to observe the execution time of an arbitrary application method using dynamic instrumentation, and use the static instrumentation to measure the overhead associated with the dynamic instrumentation. To collect a representative sample, we repeatedly choose the observed methods at random. In individual steps, outlined in Figure 3, the experiment proceeds as follows:

S1. Before launch, we use static instrumentation to augment the application. The statically instrumented application continuously reports the execution times of all methods considered in the experiment and the processor utilization.

S2. We launch the application and wait for the warmup period to pass before commencing measurement.

S3. We measure and record the execution time of all methods considered in the experiment using the static instrumentation. This data describes the performance before all dynamic measurements.

S4. We choose one of the methods considered in the experiment at random to be the observed method. We use dynamic instrumentation to insert the probe code at the start and the end of the observed method and measure the time it takes the JIT compilation associated with the code manipulation to complete.

S5. At all times between inserting and removing the probe code, we measure and record the execution time of the observed method using the dynamic instrumentation. This data is the dynamic measurement sample.

S6. We measure and record the execution time of all methods considered in the experiment using the static instrumentation. This data describes the performance during dynamic measurement.
S7. We use dynamic instrumentation to remove the probe code at the start and the end of the observed method and measure the time it takes the JIT compilation associated with the code manipulation to complete.

S8. We measure and record the execution time of all methods considered in the experiment using the static instrumentation. This data describes the performance after dynamic measurement.

S9. We continue with step S4 until enough methods are observed.

S10. Finally, we again measure and record the execution time of all methods considered in the experiment using the static instrumentation. This data describes the performance after all dynamic measurements.

The individual steps provide data to answer the overhead related questions — by comparing the measurements from steps S5 and S6, we evaluate the dynamic measurement accuracy; relating the measurements from steps S6 and S8 reveals the dynamic measurement overhead; comparing steps S3 and S10 identifies any permanent performance changes due to inserting and removing probe code, and so on.

4.2 Technical Specifics

The complete experiment implementation and configuration is available as open source, as is the performance monitoring framework. Here, we provide selected technical details necessary for interpreting the experiment results.

4.2.1 Static Probes

The static instrumentation is implemented independently of the dynamic measurement framework. AspectJ™ [1] is used to insert the probe code in the form of a before advice and an after advice. Both pieces of advice consist of a single JNI call to the actual probe code implemented natively, with statically assigned integer method identifier as the only argument. Listing 2 provides a compact pseudocode listing.

Listing 2: Probe pseudocode for static instrumentation.

To avoid excessive synchronization between the probe code and the static measurement agent, which records the measurements in thread local storage, we use versioning as in sequential locks [2]. Each measurement updates the generation counter twice, the agent records only measurements whose generation counter was odd and the same both before and after access. We also take care to use the same clock source in both the static and the dynamic instrumentation (clock_gettime with CLOCK_MONOTONIC). This makes it possible to pair the static and the dynamic measurement of the same method invocation, which is used in some parts of the evaluation.

Keeping the Java part of the static probe code as simple as possible is essential to preserve a realistic interaction between the application code and the dynamic probe code that the experiment examines. We note that AspectJ™ does not simply inline the JNI call at the method entry and method
exit points, but uses a somewhat more complex invocation sequence that first locates the (singleton) aspect and then invokes the aspect method which contains the JNI call. The code involves monomorphic invocation and predictable conditional branching, which should optimize reasonably well. The use of JNI carries some overhead as well [10].

4.2.2 Measured Application

Because the potential overhead sources depend on interaction between the application code, the probe code, and the execution platform, we need to conduct the experiment in a reasonably realistic context. We have chosen the SPECjbb2015™ benchmark [33], a Java server business benchmark that approximates a business information system of a supermarket company.

In the experiment, we consider methods that reside in the main JAR file of the benchmark as methods that the developer of the application would be likely to observe. As a technical necessity, we omit methods of anonymous classes, which cannot be selected by static instrumentation pointcuts. This leaves us with 5628 statically instrumented methods in 957 classes. For the dynamic instrumentation, we select methods that are invoked frequently enough to provide some data in 60 s of measurement. To do this, we run the benchmark with static instrumentation for 40 min and select methods called at least 100 times in the last 10 min. This yields 1286 methods.

4.2.3 Workload Generation

The SPECjbb2015™ benchmark uses an elaborate workload generation mechanism that first identifies the request rate bounds and then generates requests with gradually increasing rate to identify the benchmark score. For our experiment, the changing workload is not practical because individual measurements would be collected at different request rates – we therefore execute the benchmark with a fixed request rate. We choose the rate to be close enough to maximum rate to maintain high utilization, because that is where the instrumentation overhead is easily visible, but low enough to make overload situations rare. On the experiment platform, this is 4000 req/s.

The workload generation mechanism of SPECjbb2015™ implements an open workload model, where individual requests arrive at the configured rate regardless of the request processing speed (except for overload situations, which are detected and reported). Hence, the instrumentation overhead does not necessarily translate to changes in request rate – instead, the processor utilization rises so that the configured request rate can be maintained. Similarly, request queuing and thread scheduling effects may mask changes in response time [22]. We therefore monitor changes in processor utilization as an indication of instrumentation overhead.

Technically, we monitor processor utilization using the processor accounting subsystem of the process control group associated with the application JVM running the benchmark. This provides accurate information at nanosecond granularity, which we express as percentage of full utilization – 0 % means no processor was executing the application JVM threads in the measurement period, 100 % means all processors were exclusively executing the application JVM threads in the measurement period.

4.3 Experiment Platform

We perform the measurements on an Intel Xeon machine with 32 logical processors (E5-2660, two packages, 8 cores per package, 2 hardware threads per core). The processors are running at 2.2 GHz, the frequency is fixed for all measurements because frequency scaling and turbo boost would otherwise distort the processor utilization measurements that we use as an indication of the instrumentation overhead. The operating system is Fedora 20, 64 bit kernel 3.19.8, OpenJDK 1.7.0-79, AspectJ 1.8.6, DiSL 1.0.

The machine has 48 GB RAM in 2 NUMA nodes. We use the default configuration for heap size and force a garbage collection cycle before each processor utilization measurement to avoid including garbage collection in data intended to characterize instrumentation overhead. As a consequence, JVM arrives at a stable heap size of less than 5 GB that reflects the allocation rate between the utilization measurements.

We use a 5 min warmup period before collecting measurements, taking care to also exercise probe code during warmup, and restart the experiment every 2 h to randomize the initial conditions [13]. Figure 4 shows the initial processor utilization, indicating that by the end of the 5 min period, the benchmark execution is stable, the same is indicated by the JIT log.

![Figure 4: Processor utilization during warmup.](image)

In the experiment steps that collect measurements using the static instrumentation – S3, S6, S8 and S10 – we collect the processor utilization for 30 s and the execution time of all methods for 60 s with cyclic buffers of 256 elements per thread. In the steps that wait for the JIT compilation to complete – S4 and S7 – we consider the JIT compilation complete when no new compiled method appears for 20 s, with a timeout of 60 s. We also insert a random delay of 30 s to 90 s in step S9 to prevent inadvertent synchronization between the experiment and the application.

4.4 Measurement Results

We examine the measurement results in the same order as the overhead questions. Question Q1 deals with the measurement accuracy, that is, the difference between the measured and the actual time. Figure 5 answers with a distribution of the average difference between the time reported using the static and the dynamic instrumentation in steps S5 and S6. In numbers, the minimum average difference was observed to be 76 ns, the median was 1.34 µs, the maximum was 166.09 ms.
We (1) pair static and dynamic measurements of the same invocation, (2) compute paired difference, (3) compute average difference per method, (4) plot distribution of the averages.

Note broken scale in the top plot, the bottom plot provides a zoom in view. Results closer to zero indicate more accurate dynamic measurement, but not necessarily zero disruption due to dynamic measurement, which is examined later.

Figure 6 offers an alternative view, plotting the average ratio between the time reported using the static and the dynamic instrumentation, relative to the method execution time. The figure suggests that the relative measurement accuracy sharply declines for methods shorter than about 10 µs to 20 µs.

Figures 5 and 6 also relate to question Q2. The interquartile range of average differences is 3.33 µs, more than two times the median difference. This suggests the overhead is far from stable and therefore not easy to compensate by subtracting the average difference. We have also used one-way ANOVA to decide whether the choice of the measured method is an important factor. When ignoring the few methods with average difference over 10 µs, ANOVA returns $p$ close to 1, suggesting that the variability does not depend on which method is measured.

Questions Q1 and Q2 concern different observations of the same invocations. In contrast, the remaining questions concern observations of different invocations, we can therefore only talk about effects on average behavior. As a consequence, outliers and fluctuations have more influence over the results. To compensate for outliers, we compute averages after discarding 2.5% of the smallest and 2.5% of the largest measurements for each method.

Figure 7 is related to question Q3, examining the effect of repeated JIT compilation on dynamic measurement. The figure shows how the average method execution time, as measured by the static instrumentation, changes during dynamic measurement. Although the median rate of 1.014 indicates an intuitively reasonable small slow-down, the variability is again large, with 25% of methods exhibiting a slow-down of more than 1.23, and, more surprisingly, 25% of methods exhibiting a speed-up of more than 0.84. To distinguish the effects of instrumentation from normal execution time variability, we employ statistical testing with t-test – the slow-down is statistically significant at $\alpha = 0.05$ for 13.2% of methods, and the speed-up for 18.0% of them.

To provide more detail, Figure 8 shows a typical behavior during dynamic measurement, from initiating the measurement in step S4 to concluding the measurement in step S8. Upon inserting the probe code, the method execution time jumps up because the compiled version of the instrumented class, and possibly other related methods, is discarded. Soon after that, the executed code is compiled again and the performance returns to normal levels. Similar behavior appears upon removing the probe code. The few other outliers that are visible throughout the measurements appear at random
and are probably not due to instrumentation. We have observed this behavior with most methods.

Figure 9 is related to question Q4, examining the effect of repeated JIT compilation on application outside measurement. Here, the figure shows how the average method execution time changes from near the start to near the end of the benchmark, with dynamic measurement performed in between. The median rate of 0.994 indicates a reasonably stable performance, however, at the end of the benchmark 25% of methods are slower by a factor of over 1.12, and 25% of methods are faster by a factor of over 0.87. The slow-down is statistically significant at $\alpha = 0.05$ for 8.9% of methods, and the speed-up for 15.0% of methods.

To determine whether the changes of method execution time in Figure 9 are due to dynamic measurement, Figure 10 shows how the average method execution time changes from near the start to near the end of the benchmark when no dynamic measurement is done. The median rate of 1.003, as well as the lower and upper quartiles of 0.76 and 1.16, are similar, however, the extreme values are further apart in Figure 9 than in Figure 10. We conclude that although the benchmark exhibits long term changes in the average method execution time all by itself, dynamic measurement increases the magnitude of the most extreme changes. When no dynamic measurement is done, the slow-down is statistically significant at $\alpha = 0.05$ for 12.3% of methods, and the speed-up for 16.2% of methods.

By observing JIT compilation in steps S4 and S7, we also obtain statistics on the temporary disruptions due to code manipulation. As shown on Figure 11, JIT compilation takes more than 6.4s to complete in 50% of the probe insertion operations, and more than 3.7s to complete in 50% of the probe removal operations. Between 1% and 2% of code manipulation operations kept JIT compilation active for more than 60s.

We conclude with Figure 12, which provides an answer to question Q6 about the total overhead associated with dynamic measurement instrumentation. The figure plots the distribution of processor utilization without dynamic instrumentation, observed in step S6, and the distribution of processor utilization with dynamic instrumentation, observed in step S8. Both cases are very similar, confirming earlier findings that small scale instrumentation does not incur significant overhead – in fact, the average utilization is 73.03% without dynamic instrumentation and 72.17% with dynamic instrumentation, with the difference statistically significant at $\alpha = 0.05$.

4.5 Threats To Validity

We close our results with discussing threats to validity. We focus on the threats to statistical validity, internal validity and external validity as the most relevant validity categories.

4.5.1 Statistical Validity

To guard against threats to statistical validity, we report detailed statistical properties alongside summary results. We also provide complete data at http://d3s.mff.cuni.cz/resources/icpe2016.

The statistical analysis is complicated by the fact that, for reasons inherent to the SPECjbb2015™ benchmark implementation, the individual observations of the method execution times are not necessarily independent. As a particular consequence, if too many methods exhibit sufficiently large phases in behavior, then the conclusions on the statistical significance of the results may be distorted due to observing method behavior in different phases.

4.5.2 Internal Validity

When examining internal validity, we are concerned with the possibility that the observed overhead is not due to dynamic instrumentation, and the possibility that the dynamic instrumentation introduces overhead that is not observed. Here, most dangerous are effects that can synchronize with dynamic measurement, because such effects can introduce a systematic error when measuring the overhead. We believe such systematic synchronization is unlikely, because we randomize both the choice of the measured method and the delay between measurements. Effects due to events inherent to dynamic measurement, such as dynamic code manipulation, are obviously part of the overhead by definition.

Measuring the total overhead as a change in processor utilization similarly ensures we observe all processor overhead. The benchmark is configured to perform a constant amount of work per unit of time, anything that changes the processor demand per unit of work is bound to change the processor utilization. This deserves some attention – while
the benchmark does maintain a stable request rate, brief periods of increased overhead are likely to be compensated by queuing inside the benchmark. Because we perform every dynamic measurement for more than a minute, we believe we are likely to exhaust any queues that might mask the measurement overhead entirely.

As noted, we force a garbage collection cycle before each processor utilization measurement, and therefore influence the garbage collection overhead. Because utilization measurements happen much less frequently than young garbage collection cycles, we are not likely to influence the young collection overhead directly. We do make the full collection cycles more frequent, with multiple consequences – the total time spent in full collections is likely to be longer and the young collections may become more efficient because the references between generations are more likely to be live [17]. We believe this influence to be minor because no dynamic measurement instrumentation is likely to keep significant amounts of live data on the application heap for long, and the young collection overhead – which we are less likely to influence – should therefore dominate.

4.5.3 External Validity

External validity is concerned with how much the observed overhead generalizes to other dynamic instrumentation frameworks, other applications and other platforms. Much of the dynamic instrumentation framework revolves around the ability to redefine and retransform classes, frameworks that use the same mechanism are therefore likely to induce the same overhead due to code manipulation and code optimization. We note that this is pretty much the only reasonably portable dynamic instrumentation method currently available for Java, differences therefore should not be big.

Other dynamic instrumentation frameworks can also differ in their data storage and data processing implementation. There are many ways how this implementation can be optimized [37], we believe our implementation is reasonably straightforward to keep the results comparable with other probes written in Java.

To generalize to other applications, we must ask how much our measured application resembles other applications in those features that are relevant to dynamic instrumentation. Assuming the SPECjbb2015™ benchmark is reasonably representative, we have to account for the differences introduced by static instrumentation:

- The instrumentation slows the benchmark down roughly by a factor of four. The effect is somewhat similar to using a slower platform, but the overhead is not distributed evenly – by adding similar overhead to each method, we slow down shorter methods more than longer ones in relative terms. With the measured application becoming faster, the dynamic instrumentation overhead will become relatively smaller.

- The instrumentation increases the size of all methods by a small constant amount, making compilation and inlining somewhat less likely. Examining the JIT log, we see a total of 520 kB in 24 k inlined methods and 7 k failed inline attempts for the original benchmark, and a total of 750 kB in 36 k inlined methods and 21 k failed inline attempts for the benchmark with static instrumentation.

- The instrumentation inserts JNI calls, whose impact on compiler behavior may depend on subtle memory model implementation details [15]. Hypothetically, JNI calls may require optimization barriers, leading to more conservative optimization of the measured application. We have not included a specific evaluation of this possibility into our experiment.

Given the platform specific character of our experiment, we do not make any specific claims outside our platform. We believe the platform is representative enough to account for a large percentage of existing systems, however, different platforms – especially different JVM implementations – may behave in an arbitrary manner, yielding entirely different dynamic instrumentation overhead.

5. RELATED WORK

Instrumentation overhead is an obvious concern for any measurement framework. Instrumentation can interact with the measured system, making the measured performance different from the performance exhibited otherwise. This problem is carefully explained by Malony in [18] – in this sense our work is an experimental study of performance intrusion and performance perturbation due to dynamic instrumentation in Java.
Figure 9: Ratio between measurements reported by static instrumentation on methods before and after dynamic instrumentation. We (1) compute average static measurement per method from step S3, (2) compute average static measurement per method from step S10, (3) compute ratio of the initial average to the final average, (4) plot distribution of the ratios.

Note broken scale in the top plot, the bottom plot provides a zoom in view. The gray bars denote statistically significant differences at $\alpha = 0.05$. Results smaller than one indicate methods that run faster at experiment startup than teardown and vice versa.

Malony and Shende have investigated the measurement overhead issues especially in the context of the Tau Performance System [30]. In [19], they describe a method for compensating the measurement overhead by subtracting the execution time added by the instrumentation from the individual measurements. The method assumes the computation is calibrated for particular application and platform. Our experiment is a case of such calibration that highlights the limits of accuracy in a system where the overhead of the same probe code can vary depending on the measured method, the call site, or even ephemeral compilation decisions. Our experiment extends the overhead investigation towards dynamic instrumentation, Tau focuses on more heterogeneous platforms and more distributed applications [20].

Technologically, our work is related to Java performance monitoring frameworks that collect data through instrumentation. A prominent representative is the Kieker Framework [36], which can use multiple aspect oriented instrumentation frameworks. Detailed experiments with AspectJ™ instrumentation are in [35], where a microbenchmark consisting of a single method with known execution time is used to measure the overhead of the individual instrumentation components, and two real life monitoring tasks are reported to have no observable overhead.

We extend the results reported in [35] in multiple directions. Some are related to the differences between static and dynamic instrumentation – in particular, we measure and examine dynamic instrumentation effects, which the static instrumentation constrains to the warmup period where the measurements are discarded. On the overall design level, we consider multiple threads, and we preserve realistic conditions for interaction between the probe code and the application code. In contrast, the microbenchmark in [35] enforces method timing by observing virtual thread time and waiting for a computed deadline [37]. This solution masks possible application timing changes due to instrumentation. Kieker overhead experiments in [35] and [37] also very much complement our results – we do not deal in detail with overhead sources that are not unique to dynamic measurement instrumentation, in particular data storage and data processing. These are examined in detail especially in [37].

Another performance monitoring framework where our results are likely to apply is SPASS-meter [32]. SPASS-meter supports dynamic instrumentation, which can be used together with configurable monitoring scopes to restrict the instrumentation to relevant locations and therefore reduce overhead. Experiments that measure the instrumentation
overhead are presented in [8], where SPECjvm2008™ is used as the benchmark application and processing overhead is defined as the change in the combined benchmark score. Again, we complement these experiments by providing a much more detailed look at the dynamic instrumentation effects, and we consider our results complemented by these experiments where the more general overhead issues are concerned.

Some monitoring framework experiments [27, 9] analyze overhead in terms of average changes to application throughput or response time, which is certainly reasonable with static instrumentation and enterprise application context. Our results are generally compatible as far as the overhead magnitude is concerned.

Instrumentation overhead is analogous to overhead introduced through aspect weaving, which is examined and attributed to particular code constructs in [6]. The need for overhead analysis in dynamic aspect weaving is advocated in [11], however, the authors performed only a limited set of experiments for dynamic aspect features supported at that time. A study examining the use of aspects for profiling of heap usage, object lifetime and execution time on the SPECjvm2008™ benchmark is available in [28], again with static instrumentation – in this context, we contribute experimental results relevant to dynamic aspect weaving.

6. CONCLUSION

Dynamic performance monitoring is a promising method of reducing monitoring overhead. Coupled with dynamic instrumentation, it carries the promise of achieving zero overhead when not monitoring, because the probes that collect the monitoring data can be inserted and removed at will. On the other hand, dynamic instrumentation can interact with the execution platform in complex ways that give rise to new sources of overhead. We investigate these sources in the context of a dynamic measurement framework for Java.

Using experiments on a modified version of the SPECjbb-2015™ benchmark, we first show that the loss of measurement accuracy due to instrumentation overhead is not constant. On our platform, this limits practical measurements to methods whose execution time exceeds tens of microseconds, and also impacts the overhead compensation methods described in [19].

Next, we show that dynamic instrumentation can change the execution time of the instrumented method even when the overhead due to probes is not considered. This change can both slow down and speed up the method, sometimes significantly. We also show that the duration of the changes can vary, with short periods corresponding to compilation bursts on one end of the spectrum and periods spanning entire application execution on the other.

Looking at the compilation bursts, we show that although the code manipulation operations due to dynamic instrumentation are fast, the associated JIT compilation can last from several seconds to more than a minute. Any dynamic measurement framework looking to avoid disruptions due to JIT compilation should expect these effects to delay data collection.

Finally, we confirm that the total overhead in terms of application performance remains negligible when small scale instrumentation is deployed.

Our work is provided together with complete data and tools, available at http://d3s.mff.cuni.cz/resources/icpe2016.

Acknowledgments

This work was partially supported by Charles University Institutional Funding (SVV) and by the Research Group of the Standard Performance Evaluation Corporation (SPEC).

7. REFERENCES

Utilizing Performance Unit Tests To Increase Performance Awareness

Vojtěch Horký,
Peter Libič,
Antonín Steinhauser,
Lukáš Marek,
Petr Tůma

Contributed paper at the 6th ACM/SPEC International Conference on Performance Engineering (ICPE 2015).


The original version is available electronically from the publisher’s site at http://dx.doi.org/10.1145/2668930.2688051
Utilizing Performance Unit Tests
To Increase Performance Awareness

Vojtěch Horký Peter Libič Lukáš Marek
Antonín Steinhauser Petr Tůma
Faculty of Mathematics and Physics, Charles University
Malostranské náměstí 25, Prague 1, 118 00, Czech Republic
{horky,libic,marek,steinhauser,tuma}@d3s.mff.cuni.cz

ABSTRACT
Many decisions taken during software development impact the resulting application performance. The key decisions whose potential impact is large are usually carefully weighed. In contrast, the same care is not used for many decisions whose individual impact is likely to be small – simply because the costs would outweigh the benefits. Developer opinion is the common deciding factor for these cases, and our goal is to provide the developer with information that would help form such opinion, thus preventing performance loss due to the accumulated effect of many poor decisions.

Our method turns performance unit tests into recipes for generating performance documentation. When the developer selects an interface and workload of interest, relevant performance documentation is generated interactively. This increases performance awareness – with performance information available alongside standard interface documentation, developers should find it easier to take informed decisions even in situations where expensive performance evaluation is not practical. We demonstrate the method on multiple examples, which show how equipping code with performance unit tests works.

Categories and Subject Descriptors
D.2.6 [Programming Environments]: Interactive environments; D.2.8 [Metrics]: Performance measures; D.4.8 [Performance]: Measurements

General Terms
Performance, Measurement, Documentation

Keywords
performance documentation; performance awareness; performance testing; Java; JavaDoc

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or re-publish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

1. INTRODUCTION
The software development process can be perceived as a stream of decisions that gradually shape the final implementation of the initial requirements. Each of the decisions presents multiple options, such as choosing between available libraries, selecting appropriate algorithms and internal data structures, or adopting a particular coding style. The concerns affecting the decision are also many, ranging from cost or efficiency to complexity and maintainability, and the developers are expected to keep these concerns in balance.

The decisions that drive the development process also have a very different potential impact. Some decisions – for example whether to use a filesystem or a database to store persistent application data – are likely to have a major impact. Other decisions – for example whether to use a short integer or a long integer for a local counter variable – are likely to have a minor impact.

The perceived impact determines how the individual decisions are treated. Faced with a major-impact decision, the developer would deliberate carefully and use techniques such as modeling or prototyping to justify the eventual choice. In contrast, large-scale deliberation is not appropriate for minor-impact decisions, where the developer is more likely to simply fall back on an educated guess.

We illustrate the examples of several such choices on an imagined XML processing application. Listing 1 shows two functionally equivalent methods that accept a DOM tree [12] with purchase records as input and provide totals spent per user as output. Listing 1.a shows one developer using XPath [42] to navigate the DOM tree and TreeMap to store the totals, whereas Listing 1.b shows another developer choosing a sequence of getters for navigation and TreeMap for storage.

The impact of choices from Listing 1 is likely perceived as minor rather than major. As such, the decisions would not be made after a large-scale deliberation – choosing XPath might simply appear straightforward to a developer who has used XPath in the past, and choosing TreeMap might be similarly straightforward for a developer who thinks the totals will eventually be printed in a sorted sequence.

1But special circumstances can lend importance even to otherwise innocuous choices – for example, the code can be used in a hot loop, or availability of certain packages can be limited.
We focus on situations where the developer relies on insight to avoid large-scale deliberation. Ideally, the developer would correctly identify decisions whose impact will be minor and use educated guesses to make reasonably appropriate choices. For obvious reasons, we want to avoid situations where the developer fails to recognize that a choice deserves deliberation. We also want to avoid situations where the developer makes individually innocuous choices whose detrimental impact accumulates. Recent work on sources of software bloat suggests that such choices are common and can have a major impact on performance [33, 43, 44].

One way to avoid the bad situations is by making sure the developer can be reasonably aware of the concerns affecting each decision. For some concerns, this awareness often comes naturally with experience — simply by virtue of reading and maintaining code, the developer will have ample opportunities for feedback on criteria related to code readability and maintainability. Additional information can be provided by tools such as CheckStyle [5] or FindBugs [21].

The situation is different where awareness concerns software performance. Recognizing poor performance requires knowing what performance should be expected, and that information can only come from prototyping and measurement — in fact, the very kind of large-scale activities the developer wants to avoid. Apart from actively experimenting, the developer is therefore likely to receive feedback on software performance only when it is obviously insufficient.²

Our goal is to provide the developer with easily accessible information on software performance that is relevant to the software under development and thus increase performance awareness. This should in turn decrease the chance that the developer would make a poor choice due to lack of insight into performance.

We meet our goal by utilizing performance unit tests, introduced in detail in [4, 20]. When a performance unit test accompanies a particular software artefact, we use the workload generation component of the unit test to execute performance measurements and present the measurement results alongside the documentation for that artefact. Modern development environments such as Eclipse [14] make locating artefact documentation as easy as pointing with mouse to the artefact of interest, our solution extends the same comfort to locating performance information. Same as the unit tests, the performance information can be collected remotely on the target deployment platform and the workload can be adjusted to focus on relevant information.

In compact points, our contribution starts with identifying the documentation potential of performance unit tests and providing the technical design used to generate the performance documentation. Furthermore, we explore the benefits of our solution on multiple experimental examples. Rather than solving a particular technical issue, we provide a mechanism that helps build performance awareness — our contribution therefore carries the implied promise of improved software development process, with smaller room for mistakes due to lack of developer insight.

We start our presentation by introducing the performance unit tests in Section 2 and the motivating scenarios in Section 3. The technical design needed to generate the performance documentation is discussed in Section 4, followed by experimental evaluation in Section 5. Related work discussion and conclusion close the paper.

2. PERFORMANCE UNIT TESTS

Our mechanism for generating performance documentation uses code provided by performance unit tests, we therefore present the basic elements of the performance unit test design as the necessary context. We consider performance unit tests as described in [4, 20], using tools developed for the Java platform.

The general structure of a performance unit test is depicted on Figure 1. It is similar to the structure of a functional unit test, which usually consists of the setup, execution, validation and cleanup phases [1, 22]. In the setup phase, the workload for the system under test is prepared and the system under test is put into initial test state. In the

²And experiments carry their own risks [2, 17, 9].
### 3. Motivating Scenarios

The goal of this paper is to investigate the possibility of turning unit tests into performance documentation and to resolve the many issues associated with the technical side of this goal. To illustrate the benefits of our solution, we use multiple experimental examples.

#### 3.1 Case 1: Navigating DOM Tree

Our first example returns to Listing 1, where an XPath query is used to retrieve the content of relevant document nodes. The code is written assuming a fixed document structure and an infrequent execution. A similar situation can exist for example in applications that read their configuration from an XML file.

Two major approaches for accessing values stored in an XML file are using SAX or using DOM. With SAX, the content is presented as arguments to content handlers during parsing. With DOM, the content is available after parsing in the form of an object tree. Here, we assume DOM was chosen over SAX for simplicity.\(^3\)

With the content in the DOM tree, the developer can select a particular element using XPath, illustrated on Listing 1.a, or using a sequence of getters, illustrated on Listing 1.b. The former alternative appears more flexible – for example, the query string can be easily replaced with a more readable symbolic constant or modified to describe a more complex selection. In contrast, the latter alternative appears more straightforward – the developer may suspect that getters are simple and therefore efficient. With knowledge about performance of the two alternatives, the developer can make an informed choice.

#### 3.2 Case 2: Choosing A Collection

Our example from Listing 1 also involves choosing a collection implementation. The two alternative implementations of the `get` method both return a `Map` object, however, Listing 1.a uses a `HashMap` and Listing 1.b a `TreeMap`. Syntactically, the two alternatives are very similar, we can therefore assume the developer would decide based on criteria such as overhead or performance.

Again, a similar situation can arise in most applications that use collections. As an extension of the example from Listing 1, we also consider an imagined online store where each commodity has a list of attributes. These attributes describe optional properties of each commodity, for example the screen dimensions for computer monitors or the storage capacity for disks. The commodity descriptions reside in a database, our scenario deals with choosing the collection implementation used for caching the commodity attributes in memory. The choice should reflect these observations:

- The attributes are strings, values are objects.
- The attributes are few, typically fewer than ten.
- Some attributes are queried more often than others.
- Some attributes are used as searching criteria.
- The attributes are rarely updated.

The choices available in these scenarios are many, starting with the classes of the `java.util` package in the Java Class Library. There, the obvious choice is one of the available implementations of `Map<String, Object>`. However, a simpler list of pairs can be used as well – with the most

\(^3\)For the same reason, we assume the developer would not attempt using JAXB.
quered attributes kept in front, this choice can turn out to be more efficient than a map. The spectrum of choices is further extended by external collection implementations in libraries such as PCJ [36], Guava [19] or Trove [40].

3.3 Case 3: Choosing A Library

For the third example, we consider the common task of choosing among multiple libraries with similar purpose. In our experiment, we examine GHAL [18], Xchart [41] and JFreeChart [27], three open source libraries that offer graph plotting functions. The task at hand is generating image files with line charts of up to 10000 data points, we assume the developer found all three libraries functionally sufficient and needs to choose one.

As with the other examples, we do not mean to suggest that the developer should use performance as the sole factor guiding the decision. We do believe, however, that knowledge of performance should be used alongside other factors – in this case for example the quality of the library documentation or the maturity of the library code base – in reaching the decision. With other things being equal, performance should not be sacrificed needlessly.

4. TURNING TESTS INTO DOCUMENTATION

Assuming we have a component reasonably covered by performance unit tests, we now look at the issues involved in generating performance documentation for such component. The primary output of a performance unit test is the pass or fail status.4 This extremely condensed output is useful in automated build environments; however, it is also backed by the individual measurements collected during the test execution, which provide detailed information on the observed performance of the component under test and therefore constitute component performance documentation.

Unfortunately, distributing the measurements collected by the performance tests as a part of the component documentation is not a simple endeavour. Although some projects regularly publish their performance test results, the reports are limited to summaries – major examples of such activities include the Open Benchmarking Site [35], which offers summaries for several thousand test results, or the ACE+TAO+ CIAO Distributed Scoreboard [38], which provides results of the selected measurements across the entire project history.

One of the reasons why performance measurements are not provided together with the documentation is the measurement duration. The measurements required to generate a complete performance documentation would take too long for even a moderately sized projects – even for the testing purposes, the performance unit tests need to be run on carefully selected test cases only [20].

Another factor is the volume of the measurement data collected. The study in [20], where the tests have covered about 20% of code, has produced several hundreds of kilobytes of compressed measurement data, easily an order of magnitude more than the size of the byte code tested.

Finally, the performance measurements are platform-dependent. Although [20] shows that relative performance can be a reasonably stable property across platforms, the difference in absolute numbers generally makes it difficult to relate the measurement results collected on one platform to the expected performance on other platforms.

4.1 Using Workload Generators

Section 2 has introduced the unit test structure, in which the workload generator prepares the input arguments for test execution based on the specified workload parameters. Listing 3 contains an example generator code that prepares arguments for invoking the LinkedList.contains() method in a sequence that produces a given number of hits and misses. The workload parameters – the size of the list and the number of hits and misses – are specified as the generator inputs. Briefly, the generator first prepares the underlying list, on which the contains() method will be invoked. Next, the arguments of the individual invocations are prepared, first for hits (the invocation looks for a random integer from a range that is known to be in the list) and then for misses (the invocation looks for an integer beyond the range known to be in the list).

Although the generator prepares the arguments for individual method invocations in a form reminiscent of the invoke() method arguments from the java.reflect package, our tool for performance unit tests does not rely on reflection to execute the workload. Instead, the tool generates code that extracts the arguments (unboxing and recasting as necessary) and then performs a standard method invocation. This is because reflection introduces a disruptive overhead.

We use the workload generators to generate the performance documentation on-the-fly on the application developer side. This helps overcome the outlined challenges – rather than having the component provider collecting and distributing measurements, the application developers run the selected measurements of interest locally. Our performance unit test framework also supports remote testing, with the measurements performed on a remote deployment platform rather than on the build system itself, this makes it possible to display results directly relevant to the deployment platform.

4.2 Associating Generators With Methods

To generate a performance documentation for a method, we need to locate the workload generator associated with that method. In the performance unit tests, the association relies on annotations, as illustrated in Listing 4. In certain situations, such as when testing proprietary code, it is not easily possible to attach the annotation directly to the measured method. When this is the case, we introduce a helper class that defines an empty method with the same signature and attach the generator to this method instead, as illustrated in Listing 5.
class LinkedListGen {
    public Iterable<Object[]> contains (int size, int hits, int nohits) {
        ArrayList<Object[]> result = new ArrayList<> (hits + nohits);
        LinkedList<Integer> list = new LinkedList<> ();
        for (int i = 0; i < size; i++) {
            list.add (new Integer (i));
        }
        Random rnd = new Random ();
        for (int i = 0; i < hits; i++) {
            Integer searchFor = rnd.nextInt (size);
            Object[] args = new Object[] { list, searchFor }
                 result.add (args);
        }
        for (int i = 0; i < nohits; i++) {
            Integer searchFor = new Integer (size + i);
            Object[] args = new Object[] { list, searchFor }
                 result.add (args);
        }
        return result;
    }
}

Listing 3: Generator for invoking the contains() method of a linked list.

@TestHelper (for = java.util.LinkedList.class)
class LinkedListHelper {
    @Generator ("LinkedListGen#contains")
    public boolean contains (Object obj) {
    }
}

Listing 5: Binding workload generator with the measured method through a helper class.

In the straightforward example from Listing 4, we can locate the generators that can be used with a particular method simply by enumerating the method annotations. In the example from Listing 5, the situation can be likened to propagating documentation across an interface-implementation relationship. The annotation information is kept in byte code, the generators can therefore be located even for methods in packages that are distributed in compiled form.

Complex performance unit tests require workloads that invoke multiple methods of a component. In these cases, we use a special-purpose test method that executes the individual component methods and associate the generator with this special-purpose method. Listing 6 shows such a special-purpose method, used by a unit test of a graph plotting library – the workload uses the library to create an image file of given dimensions that shows given data points using a line plot, this requires calls to multiple library methods.

The example from Listing 6 makes associating the generator with individual component methods more difficult, we therefore use extra ShowWith* annotations that specify classes and methods whose performance the test exercises.

The use of helper methods shown Listing 5 and Listing 6 requires searching for the classes that implement these methods. To reduce the search time, we assume the developer would specify a separate class path to be searched.

4.3 Limiting Measurement Time

To avoid the issues with test execution duration, we expect that the performance documentation would be generated on demand, much in the same way as the JavaDoc [24] documentation is displayed on demand when the developer selects a particular method. To react quickly enough in an interactive environment, we need to execute the measurements in a short time frame. There are situations where this is clearly not possible, for example when even a single invocation of the measured method takes a long time to complete. When the measurement method executes in reasonable time, we also need the workload generator to prescribe a short enough workload.

As a complication, the requirement of relatively short workloads conflicts with the need to make the performance unit tests reliable – robust results are known to require long measurements, possibly with multiple restarts and multiple compilations [9, 17, 29]. This tendency was also reflected in our initial performance unit test experiments [20], where the workload generator design tended to put test robustness first and test duration second. This resulted in execution times inapplicably long for an interactive measurement context.

We aim to solve this issue by gradually updating the presented results. The very first time a developer displays the documentation for a method, we only measure the method for a short period of time, limiting both the scale of workload parameters used and the number of measurement repetitions performed. After displaying the initial results, further measurements are collected on the background and the initial results are gradually refined – a finer scale of the workload parameters is used and the measurements are repeated more times. Because the measurement results are preserved, this only happens when a method documentation is first examined.

Figure 2 illustrates the effect of gradually updating the presented results on a workload that measures the duration of the LinkedList<Integer>.contains() method when looking for an element that does not exist in the collection. We see that a very short measurement – 1 second – reveals the general linear complexity trend, but does not run long enough for runtime optimizations to occur. A slightly longer measurement – 5 seconds – suffices to present stable results.
for six workload parameter values. A measurement of 300 seconds is safely enough to collect stable measurements for 100 workload parameter values.

A more complex issue concerns our very ability to create workload generators that can drive short measurements. Most measurements must execute in a loop to provide reasonable results (for many reasons – for example to execute a representative workload, to trigger runtime optimization, or to compensate for measurement noise or measurement overhead). When the method invocations in the measurement loop change the measured object state, the collected measurements may no longer reflect the intended workload. For example, it is difficult to write a workload generator that would measure the time to add an element to a collection of particular size without invoking any other collection operation – with each measurement repetition, the collection would grow and the measurement would no longer apply to the initial collection size. A specific solution is required for each particular situation – in the collection example, we can simply measure the time to add and remove an element in the same loop, because this workload variation does not grow the collection as the measurement progresses.

4.4 Presenting Measurement Results

The workload produced by a workload generator depends on both the implementation of the generator and the supplied workload parameters. When selecting the measurement and presenting the results, we therefore need to provide both the description of the generator and the description of the workload parameters.

We rely on the fact that each generator is simply a method of a class and therefore can be documented using JavaDoc. JavaDoc can be used to capture the description of the whole generator as well as the description of the individual workload parameters, which are simply arguments to the generator method. The advantage is that the developer writing the workload generator uses a standard documentation tool. The disadvantage is that the comments are not preserved when compiling into byte code, the documentation may thus not be available in packages distributed in compiled form.

5. EXPERIMENTAL EVALUATION

The ultimate aim of our approach is to improve performance awareness among software developers so that they
can write more efficient code. With this aim in mind, the evident method of experimental evaluation is to conduct a study that would test whether developers with access to performance documentation write more efficient code. We investigate this evaluation method next, however, it turns out the study is too expensive to be practical. We therefore turn to additional methods of examining our approach, looking at whether reasonably realistic use cases can be found, and whether real software can benefit.

We have executed our experiments on a 2.33 GHz machine with two quad core Intel Xeon E5345 processors and 8 GB of memory, running Fedora Linux with kernel 3.9.9, glibc 2.16-33 and OpenJDK 1.7.0-25, all in 64 bit mode. The libraries used in the experiments were JDOM 2.0.5 [26] with Jaxen 1.1.6 [25], GRAL 0.9 [18], XChart 2.3.0 [41] and JFreeChart 1.0.17 [27].

Our experimental implementation includes a complete performance unit test framework for Java [39]. The framework supports for workload generators attached through annotations, local and remote measurement, result collection and processing. We have not yet implemented the user interface integration envisioned in our approach, specifically the workload generator and workload parameter selection and the integrated result display features. The graphs shown here are produced manually from the measurement data.

5.1 Developer Awareness Study

To test whether developers with access to performance documentation write more efficient code, we design an experiment where multiple developers are given the same implementation task, and the performance of the resulting implementations is compared. In terms of hypothesis testing, we postulate the following null hypothesis: availability of performance documentation during software development has no impact on the eventual implementation performance. Our independent variable is the availability of performance documentation, our dependent variable is the execution time of the resulting implementation.

With limited resources to hire professional developers, our test subjects are volunteer computer science students. The students have completed a Java Programming class and participated in an Advanced Java Programming class, the average self assessment of the relevant programming language skills is 3.5 on a scale of 1 to 5. The students were not told the purpose of the experiment beyond the bare minimum needed to ask for consensus.

As the implementation task, we choose XML processing with the JDOM library, for which we have developed the performance unit tests in [20]. The students were asked to implement an application that accepts a DocBook [11] file on the standard input and produces a list of cross references grouped per section on the standard output. This is a reasonably simple task – our reference solution has less than 200 LOC; yet it provides opportunity for exercising multiple different uses of the JDOM library and the standard collections.

We have assigned the task to 39 students split into three equal-sized groups – one control group and two test groups. The control group was given the standard JDOM library documentation, the two test groups were given two versions of documentation augmented with performance information, one strictly correct and one deliberately misleading. In both versions, methods relevant to the task were identified together with possible alternatives. In the first test group, the true performance measurements of all methods were provided, with the intent to guide the students towards more efficient implementation. In the second test group, the performance measurements of the fastest and the slowest methods in some alternatives were switched, to guide the students towards less efficient implementation.

The experiment results suffered from high attrition rate. Of the 39 students, only 12 have submitted implementations that have passed minimum correctness tests. The attrition rates have not differed greatly between the three groups, suggesting low general motivation to complete the task rather than bias particular to individual groups.

More importantly, the execution times of the implementations have exhibited very high variance. On a test input of 80 MB, the fastest implementation has finished in 3.28s, but the slowest implementation has not finished in one day. The median execution time was 5.38s. The high variance prevents making statistically significant rejection of the null hypothesis at reasonable scales – even if we filter out the execution times that exceed one minute as anomalies, the variance remains such that a two-sided t-test at the 5% confidence level would only spot average differences above 6.13s. Our approach does not aspire at performance improvements of such a large relative magnitude.

Given that our approach targets minor-impact decisions, we believe it could be considered successful if it brought average performance improvement in the order of tens of percent. We can use the common sample size estimation methods to guess the required experiment size. In statistical terms, we consider the probability $P$ that the sample average performance $\bar{X}$ estimates the true mean performance $\mu$ with a
relative error exceeding \( \delta \), and we want \( P \) to remain at a reasonably low confidence level \( \alpha \): 
\[
P((X - \mu)/\mu) \geq \delta = \alpha.
\]
Under normality assumptions, reasonable for this particular lower bound computation, we can estimate the minimum sample size \( n = (z_{1-\alpha} + \sigma^2)/(\delta^2 + \mu^2) \) \{34\}. For our experiment results, \( \alpha \) set to 5% and \( \delta \) set to 10%, this suggests a minimum of 2397 students per group, or 128 students per group if we again filter out the execution times that exceed one minute as anomalies.

Our study did not provide sufficient data to rule on whether our approach indeed helps improve performance awareness among software developers, however, it did point out another important observation – a direct evaluation of the possible effect would require a study with a minimum of several hundred participating developers. It is possible that some aspects of the experiment can help reduce this number. For example, using more experienced developers or constraining the assignment may reduce the execution time variance, however, neither solution is without drawbacks. Before considering this more expensive evaluation, we therefore turn to additional methods of examining our approach.

5.2 Evaluating Motivating Scenarios
To see whether reasonably realistic use cases can be found, we evaluate our approach in the context of the motivating scenarios from Section 3. For each scenario, we show what the generated performance documentation would reveal and discuss how the information relates to the eventual developer decision.

The exact shape of the performance documentation depends on the available workload generators. It is rather unlikely that a developer would address a particular scenario directly – for example, when a scenario calls for comparing the performance of two collection implementations on a particular workload, it would be ideally addressed by a workload generator that can drive both collection implementations with that exact workload. Having a performance unit test with such a workload generator would seem too much of a coincidence, we are more likely to have workload generators that drive individual collection implementations in some other – possibly similar – workloads. We discuss this issue with each scenario too.\(^5\)

5.3 Case 1: Navigating DOM Tree
In this scenario, the developer considers whether to navigate a DOM tree using a sequence of getters or using XPath. The choice with sequence of getters relies on the \texttt{Element.getChild()} method. Internally, the method is fairly complex, using a lazy element name filter and a cache of filter results – we can therefore reasonably assume the component developer would equip the method with a performance unit test that makes sure both the lazy filtering and the result caching work. This suggests a workload generator that calls the \texttt{getElement()} method on an element with a variable number of children and a variable position of the matching child, coupled with a performance unit test that makes sure the \texttt{getElement()} timing does not depend on the child count when the matching child position stays constant. The

\(^5\)For the curious reader, we have also evaluated the alternatives from Listing 1. On 10000 purchase records of 1000 customers, Listing 1.a takes an average 304 ms to complete, Listing 1.b completes in an average of 7 ms.

![Figure 3: Measurements of \texttt{Element.getChild()} for varying child count and selected matching child position.](image)

![Figure 4: Measurements of XPath query for varying element count and selected matching element position. Compilation amortized over 100 queries.](image)

5.4 Case 2: Choosing A Collection
In the second scenario, the developer decides what collection to use to store a relatively small number of variable attributes. Evaluating the performance of a collection implementation against a particular workload is a common endeavor, we therefore assume the evaluation would provide a workload generator. Our implementation of such a workload generator accepts basic workload parameters – the initial size of the collection, the number of operations to perform, and the relative frequencies of individual operations in the workload. The operations are inserting and removing an element, iterating over the collection, and two versions of searching the collection (one that searches for an existing
library developers can use the examples distributed with the

our needs exactly, we assume each library would provide
pect the libraries to provide workload generators matching
different graph plotting libraries. Although we cannot ex-
is potentially feasible.

is potentially feasible.

exhibiting memory requirements for performance by using arrays

The results would suggest that for small collections, trad-

not be much slower than searching a sophisticated collec-
sizes suggested in the scenario, searching an array might

to also look at the performance of collections that do not
exist).

Ideally, the generator would also permit specifying the
type of the collection elements (the type parameter of the
collection type). So far, we have not tackled the issue of
specifying a type as a workload parameter, and instead as-
sume multiple workload generators would be present – one
for each type for a small set of common types. Figure 5
shows the measurements on a workload suggested in the
scenario, that is, a mix of one-third iterations, one-third
successful searches, one-third unsuccessful searches on the
Map<String,String> type. The figure can help the devel-
oper realize that in this scenario, all three collections per-
form reasonably well, with perhaps a small saving to be
made by using LinkedHashMap.

Among other likely concerns in the choice of a collection
is the memory overhead [33]. This might lead the developer
to also look at the performance of collections that do not
implement the Map interface – after all, for the collection
sizes suggested in the scenario, searching an array might
not be much slower than searching a sophisticated collec-
tion. The developer can look at the same workload on the
Collection<String> type, with results shown on Figure 6.
The results would suggest that for small collections, trad-
ning memory requirements for performance by using arrays
is potentially feasible.

5.5 Case 3: Choosing A Library

In the last scenario, the developer needs to select among
different graph plotting libraries. Although we cannot ex-
pect the libraries to provide workload generators matching
our needs exactly, we assume each library would provide
tests demonstrating typical usage – as a matter of fact, the
library developers can use the examples distributed with the

library documentation, because the amount of additional
work required to turn the examples into generators is low.

In our scenario, the developer would look at workload gen-

ators that plot line charts. The workload parameters of
the generators can differ from library to library, the devel-
oper will thus be presented with separate results rather than
the combined result plot we present here. The generator we
use creates a PNG image with a line chart, the workload pa-
rameters were the image dimensions and the number of data
points. Figure 7 offers a comparison of the three libraries
under consideration.

As another example of an interesting behavior, the de-
dependency on Figure 7 is not strictly monotonic. This be-
havior correlates with the line plot appearance – with too
many data points, the lines merge into larger blotches that
are easier to compress.

5.6 Evaluating Existing Projects

Although our motivating scenarios were inspired by real
code, they are not from real projects. Lacking the means
to involve a sufficient number of external developers, we in-
stead examine the existing projects ourselves, looking for
opportunities for performance improvement based on per-
formance documentation. Many of our performance unit
tests were developed for the JDOM library [20], we have
therefore looked for open source projects that use JDOM.

We have used the Ohloh\(^6\) open source project tracking
site to look for projects that import classes from the JDOM
library package, locating roughly 100 projects. We did not
consider projects that are simply too big to evaluate, such
as the Eclipse development environment. We have also ex-
cluded projects that use JDOM merely to read their con-
figuration files, because in such projects the performance
improvement is unlikely to matter. Finally, some projects
did not build on our experimental platform. The following
sections document cases of performance improvement.

5.7 Project 1: Buildhealth

Buildhealth\(^7\) is a utility that parses the reports of common
software development tools, such as JUnit or FindBugs, to
create a build health summary. Many of the parsed reports
are stored in XML and Buildhealth uses JDOM for their
analysis. The individual modules for parsing the reports
often use XPath. Our performance documentation reveals
high initial cost associated with XPath compilation, we have
therefore decided to replace simple XPath expressions – such

\(^6\)http://www.openhub.net

\(^7\)https://github.com/pescuma/buildhealth
We do not analyze other qualities of the modification, such as code readability or maintainability. Without doubt, complex XPath expressions would be very difficult to replace in a similar manner, however, in this case the expressions were sufficiently simple to justify the change.

We have evaluated the performance effect of our changes in three different settings. All concern the processing time of JUnit reports for the Apache Ant project, the size of the report files is approximately 4 MB. The average execution times are in Table 1, boxplots are displayed in Figure 8.

The first of the three settings serves to explain the performance improvement due to XPath compilation, the API calls used by the modified version of the utility are faster than even compiled XPath.

In the second of the three settings, we have executed Buildhealth as an Ant task. Using the ProfileLogger support, we have measured only the time needed to execute Buildhealth, without the overhead of the Ant invocation.

In the third setting, we have measured the total time to execute Buildhealth. This setting is the most realistic, but it does not allow us to filter the warm up effects, which are therefore included in this and further results. Overall, the changes have improved performance of the utility on our data by about 5 %, which can be considered a success especially given that the modifications were small and performed without deep knowledge of the source code.

5.8 Project 2: METS Downloader

The METS (Metadata Encoding and Transmission Standard) is a “standard for encoding descriptive, administrative, and structural metadata regarding objects within a digital library” [32]. To facilitate downloading METS documents recursively (with referenced files), an unofficial downloader exists.

The downloader uses XPath to extract the list of referenced images that need to be downloaded. Technically, the XPath expression selects a link attribute from elements nested in certain order. Motivated by the same information as in the previous project, we have replaced this XPath expression with a series of nested loops iterating over the individual child elements. The modification is located in the getImageURLs method of the MetsDocument class.

Table 2 shows both the total execution time of the downloader and the execution time of the getImageURLs method, measured on about 67 MB of data from the UCB library site. While the impact on the overall performance – which is influenced much more by the network latency and throughput – is negligible, the method alone executes in one fifth of the original time, as also illustrated on Figure 9.

5.9 Project 3: Dynamic Replica Placement

For our last project, we have chosen a prototype implementation that accompanied a paper about dynamic replica placement in CDN [8]. Again guided by our performance documentation, we have replaced a recursive document traversal based around the getChildren method with a single iteration over elements returned by the getDescendants method in one of the included tests.

The results are displayed in Table 3 and in Figure 10. The modifications were again relatively small, we were able to improve the total execution time by 2 % and the affected method alone by 6 %.

6. DISCUSSION AND RELATED WORK

In our evaluation, we have demonstrated the kind of information that performance documentation can provide to the software developer. We have also shown that real software projects do contain the kind of code constructs that lead to Improved

Table 1: Buildhealth results

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>No XPath</th>
<th>Cached XPath</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repeated</td>
<td>938.3 ms</td>
<td>908.4 ms</td>
<td>929.8 ms</td>
</tr>
<tr>
<td>Ant task</td>
<td>2.23 s</td>
<td>2.16 s</td>
<td>–</td>
</tr>
<tr>
<td>Standalone</td>
<td>2.52 s</td>
<td>2.40 s</td>
<td>–</td>
</tr>
</tbody>
</table>

Table 2: METS downloader results

<table>
<thead>
<tr>
<th></th>
<th>XPath</th>
<th>Nested getChildren</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total time</td>
<td>120.2 s</td>
<td>120.4 s</td>
</tr>
<tr>
<td>getImageURLs</td>
<td>131.5 ms</td>
<td>27.4 ms</td>
</tr>
</tbody>
</table>

Figure 8: Buildhealth

as selecting all children of given name – with more efficient but less versatile API calls. The changes were a few lines in size and done within minutes.

We do not analyze other qualities of the modification, such as code readability or maintainability. Without doubt, complex XPath expressions would be very difficult to replace in a similar manner, however, in this case the expressions were sufficiently simple to justify the change.

We have evaluated the performance effect of our changes in three different settings. All concern the processing time of JUnit reports for the Apache Ant project, the size of the report files is approximately 4 MB. The average execution times are in Table 1, boxplots are displayed in Figure 8.

The first of the three settings serves to explain the performance improvement due to XPath compilation, the API calls used by the modified version of the utility are faster than even compiled XPath.

In the second of the three settings, we have executed Buildhealth as an Ant task. Using the ProfileLogger support, we have measured only the time needed to execute Buildhealth, without the overhead of the Ant invocation.

In the third setting, we have measured the total time to execute Buildhealth. This setting is the most realistic, but it does not allow us to filter the warm up effects, which are therefore included in this and further results. Overall, the changes have improved performance of the utility on our data by about 5 %, which can be considered a success especially given that the modifications were small and performed without deep knowledge of the source code.

5.8 Project 2: METS Downloader

The METS (Metadata Encoding and Transmission Standard) is a “standard for encoding descriptive, administrative, and structural metadata regarding objects within a digital library” [32]. To facilitate downloading METS documents recursively (with referenced files), an unofficial downloader exists.

The downloader uses XPath to extract the list of referenced images that need to be downloaded. Technically, the XPath expression selects a link attribute from elements nested in certain order. Motivated by the same information as in the previous project, we have replaced this XPath expression with a series of nested loops iterating over the individual child elements. The modification is located in the getImageURLs method of the MetsDocument class.

Table 2 shows both the total execution time of the downloader and the execution time of the getImageURLs method, measured on about 67 MB of data from the UCB library site. While the impact on the overall performance – which is influenced much more by the network latency and throughput – is negligible, the method alone executes in one fifth of the original time, as also illustrated on Figure 9.

5.9 Project 3: Dynamic Replica Placement

For our last project, we have chosen a prototype implementation that accompanied a paper about dynamic replica placement in CDN [8]. Again guided by our performance documentation, we have replaced a recursive document traversal based around the getChildren method with a single iteration over elements returned by the getDescendants method in one of the included tests.

The results are displayed in Table 3 and in Figure 10. The modifications were again relatively small, we were able to improve the total execution time by 2 % and the affected method alone by 6 %.

6. DISCUSSION AND RELATED WORK

In our evaluation, we have demonstrated the kind of information that performance documentation can provide to the software developer. We have also shown that real software projects do contain the kind of code constructs that lead to
An important aspect of approaches that extend the interface description is the choice of metrics for quantifying performance. One possible approach is to devise a portable metric that summarizes the performance as a platform-independent value. For example, JavaPSL [16], used for detecting performance problems in parallel applications, normalizes the values to \([0, 1]\) to simplify comparison. The performance unit tests in [4] also rely on relative comparison to tackle the platform-dependent nature of the measurements. In contrast, the approach described here provides platform-specific information in absolute numbers.

As a practically useful extension, we also consider measuring more than just the execution time of a specific method. Of eminent interest are the memory-related metrics such as heap consumption or cache utilization. The basic idea is a straightforward extension of this paper, however, the technical process of defining and collecting the memory-related metrics presents specific challenges especially for the separation of the workload generator from the measured method.

In a broader context, our work also complements the research effort in the performance adaptation domain. We have touched on the issue of choosing a suitable collection implementation, which is addressed in depth by the Chameleon tool [37] – the tool observes access patterns on individual collections and, based on a set of static rules, issues recommendations on which collection implementation to use. The problem of choosing from multiple available implementations was explored for example in the context of selecting the best parallel algorithm [45]. Other frameworks address the need for adaptive configuration [10] and other situations. What these approaches have in common is that the developer has to be aware of the potential for dynamic adaptation to attempt the adaptation in the first place. Our work improves the awareness of the likely performance of individual software components and therefore helps the developer identify the adaptation opportunities to be explored in detail.

On the benchmarking side, our work is also related to the existing benchmarking tools, especially those in the micro benchmark category. Among such tools for Java are jmri [28], Japex [23] or Caliper [7]. These projects allow the developer to mark a method as a benchmark and collect the results. Our approach stands apart especially in using the unit test code and in integrating the performance evaluation into the interactive software development process.

7. CONCLUSION

Our work seeks to improve the perception of typical software performance that the software developers form in their work. We propose a system where performance unit tests acquire dual purpose – besides evaluating a component, the unit tests also serve to generate performance documentation for application developers that use the components. Our approach facilitates building software architectures where the performance of individual components can be easily examined and where the decisions that steer the development process can take this performance into account.

We have illustrated the potential use of performance documentation on multiple examples, each accompanied by measurements carried out using a real performance unit test tool. We believe the potential benefits of our approach parallel those of functional unit testing, and although such benefits are difficult to quantify experimentally [30], we find it

<table>
<thead>
<tr>
<th>getChildren</th>
<th>getDescendants</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total time</td>
<td>1.93 s</td>
</tr>
<tr>
<td>Processing alone</td>
<td>744.3 ms</td>
</tr>
<tr>
<td>Original</td>
<td>3 ms</td>
</tr>
<tr>
<td>Improved</td>
<td>89 s</td>
</tr>
</tbody>
</table>

Figure 10: CDN simulation, the process method.
reasonable to expect that a better-informed developer makes fewer wrong decisions.

Among the plethora of performance optimization opportunities, we see the contribution of our approach especially with the many low profile decisions. An experienced developer should not make major performance mistakes often, however, that same developer can make a conscious decision to ignore the performance impact of low profile decisions simply for the sake of fast development. Better performance awareness reduces the need for this particular sacrifice. We also provide more chances to recognize situations where advanced performance solutions, such as dynamic adaptation or manual optimization, are warranted by the potential performance benefit.

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

8. REFERENCES
Part III

Related Work and Conclusion
This chapter provides an overview of the related work. It complements the related work sections present in the included papers by focusing on the most relevant publications and providing a deeper comparison than was possible in the limited space of the included papers.

We have split this chapter into four sections that group the related work based on the dominant contribution. Section 8.1 describes approaches related to capturing and describing performance expectations and tests. In Section 8.2, we describe frameworks for performance monitoring and measurement that are most relevant to our work. Regressions and anomaly detection techniques are covered in Section 8.3. Works related to accuracy and overhead of runtime performance monitoring are described in Section 8.4.

### 8.1 Performance Expectations Description

A brief overview of approaches for formalization of performance expectations is given in Table 8.1. In the following text we describe the approaches in more detail, focusing on comparison with SPL.

Ghanbari et al. [32] argued the need of a generic language suitable for inspection of performance of multi-tier systems. The SelfTalk language allows to capture performance analysts’ assumptions about performance and to verify them at runtime.

The assumptions can be written as simple comparisons (i.e., one value is smaller than another one) but can also describe more complex relations (e.g., linear dependence). An example of the SelfTalk language is shown in Listing 8.1, capturing an assumption that “throughputs of all components are linearly correlated [32].”
Table 8.1: Comparison summary of languages for capturing performance assumptions.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Data collection</th>
<th>Data comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPL</td>
<td>Performance unit tests</td>
<td>Statistical tests</td>
</tr>
<tr>
<td>SelfTalk/Dena [32]</td>
<td>Existing probes, system monitoring</td>
<td>Score derived from goodness of curve fitting</td>
</tr>
<tr>
<td>PSpec [75]</td>
<td>Application logs</td>
<td>Simple aggregations(^{\text{a}}), scalar comparison</td>
</tr>
<tr>
<td>PA [93]</td>
<td>Manually inserted probes</td>
<td>Scalar expressions</td>
</tr>
<tr>
<td>Pip [77]</td>
<td>Probes log to files</td>
<td>Simple aggregations(^{\text{a}}), scalar comparison</td>
</tr>
<tr>
<td>D(^{3})S [58]</td>
<td>Automatic probes</td>
<td>User-provided evaluation</td>
</tr>
<tr>
<td>A language [89, 90]</td>
<td>System monitoring</td>
<td>Simple aggregations, outlier filtering</td>
</tr>
<tr>
<td>Uncertain(T) [9]</td>
<td>Developer-defined</td>
<td>Wald’s sequential probability ratio test</td>
</tr>
</tbody>
</table>

\(^{\text{a}}\) For example arithmetic mean, sum or number of samples.

Listing 8.1: Example of performance assumption in SelfTalk (taken from [32]). The assumption captures that “throughputs of all components are linearly correlated”.

```haskell
HYPOTHESIS HYP-LINEAR
RELATION LINEAR(x,y) {
    "x.unit='1/s' and y.unit='1/s'"
}
CONTEXT { } 
```

The assumptions are validated using the Dena runtime system. It uses curve fitting algorithms, the overall score (i.e. how much the relation expresses the actual state) is derived from a goodness-of-fit metric. Dena relies on existing probes and data sources for computing the score.

Compared to SPL, SelfTalk/Dena targets multi-tier systems, although the concepts of SelfTalk can be projected to unit testing as well. The language allows the analysts to capture more complex expectations – the language is more verbose than SPL, which targets simpler comparisons. From the perspective of validation, Dena relies on curve-fitting while SPL uses statistical hypothesis testing. For specific scenarios, the evaluation “engines” in Dena and SPL are interchangeable but generally they target different granularity of the components, requiring different data processing approaches.

Perl and Weihl [75] introduced PSpec, a language for expressing performance assertions. Their approach assumes that a performance log with in-
Listing 8.2: Example of performance assumption in PSpec (taken from [75]) expressing the requirements about average request rate.

```plaintext
interval ReadReq =
  s: StartRead,
  e: EndRead
metrics
  time = ts(e) - ts(s)
end ReadReq;

assert { mean r : ReadReq : r.time } >= 0.1 sec.
```

Listing 8.3: Example of performance assertion in PA measuring sparse matrix vector multiply loop (taken from [93]).

```plaintext
pa_start (&pa, '$ipc_peak*0.5<$ipc');
for (j = 1; j <= lastrow - firstrow + 1; j++) {
  sum = 0.0;
  for (k = rowstr[j]; k < rowstr[j + 1]; k++) {
    sum = sum + a[k] * p[colidx[k]];
  }
  w[j] = sum;
}
pa_end (pa);
```

Interesting events is available and PSpec queries information in such log. The user can use scalar comparisons of individual events or construct derived events directly in the PSpec language, as can be seen in Listing 8.2, which describes the average request rate using a simple aggregation.

The general goal of PSpec is very similar to ours as PSpec can be used for performance testing of arbitrary granularity. While PSpec offers derivation of new events (something SPL lacks altogether as we are generally interested in execution time only at the unit level), the evaluation provides less information (e.g. no p-value).

Vetter and Worley [93] introduced the PA language as a way to insert performance assertions directly into the source code. The developer specifies the assertion as a regular function call preceding the block of code that is supposed to be benchmarked. The actual assertion is evaluated at the end of the measured block where an end-of-assertion call is inserted as illustrated in Listing 8.3.

The initial call `pa_start` determines what sensors are needed and initializes them. `pa_end` then evaluates the assertions. The assertions are formed by scalar expressions only, though the framework offers several aggregation-like variables (e.g. `ipc_peak`). PA can use hardware counters or variables provided by the developer, allowing parametrized assertions.

PA provides truly runtime performance assertion checking while SPL targets build-time testing although PA can be used for that as well. SPL is almost a superset of the features provided by the PA language, which lacks the variety of available metrics. SPL gives more information about assumptions that do not hold because it employs statistical testing; on the other hand
Listing 8.4: Example of performance assertion in Pip. The validator describes a quorum-system 2-out-of-3, performance assertion is the last statement (taken from \[77\], simplified). The Client asks the Coordinator for the data. The Coordinator sends to three Peers the request to confirm. Once two replies are received, Coordinator can answer the Client. The Peer only needs to answer the message.

```plaintext
validator ReadQuorum {
  thread Client(*, 1) {
    send(Coordinator, "READ")
    recv(Coordinator, "DATA");
  }
  thread Coordinator(*, 1) {
    recv(Client, "READ")
    task("rpc::Read") {
      repeat 3 send(Peer, "QUORUM_READ")
      repeat 2 recv(Peer, "QUOROM_OK"
    }
    send(Client, "DATA")
  }
  thread Peer(*, 3) {
    recv(Coordinator, "QUORUM_READ")
    task("rpc::ReadReq") {
      send(Coordinator, "QUOROM_OK")
    }
  }
}
assert(average(REAL TIME, ReadQuorum) < 30ms)
```

runtime checking calls for rapid evaluation where scalar comparison is more suitable.

A more generic approach was outlined by Reynolds et al. \[77\]. The Pip system is used to describe the overall behavior of a distributed system (i.e. its functional properties) and also attach performance assumptions to the description. The basic block of Pip is a path instance which is a series of time stamped events – an event can be either a task (i.e. instrumented function call), a message (i.e. communication between threads, including inter-host communication or local locks) or a notice (an opaque log message). The user provides his expectations through recognizers that are matched against the path instances. A recognizer can be accompanied by an assertion. In this assertion performance assumptions can be expressed. An example capturing distributed quorum-based data retrieval is depicted in Listing 8.4.

Unlike SPL, Pip targets distributed systems and generally is more suitable for high-level assumptions about the system under test. While SPL aims for build-time testing, Pip can be used on production systems to monitor actual behaviour and performance. Related to that is the use of absolute performance metrics in the expectations together with simple aggregation functions and direct comparison (in contrast to statistical testing used with SPL).
Listing 8.5: Example of performance assertion in the A language capturing requirement of balanced load of individual application servers (taken from [89], simplified).

In this example, we work with single database server and a set of application servers. The task that is about to be executed by the operator is adding a new application server. Once the task is completed (represented by the `wait()` operation), the system checks that all application servers have connection to the database and that CPU load of all application servers is the same. The `EQUAL` operator uses outlier filtering as CPU utilization can jump spuriously.

```plaintext
task add_application_server {
  db DBServer(IP="dbserver.domain.tld") with config MySql
  as_all ApplicationServerGroup(IP="*") with config Tomcat

  wait()
  assert all_connected(db, as_all)
  assert balanced(EQUAL(as_all.cpu_utilization))
}
```

Similarly, Tjang et al. [89] use the A language [90] to capture properties of distributed services. Their approach relies on model validation where service engineers describe valid behaviours of the system. The operators can then use these validators to check that their actions (e.g. deployment on a new machine) do not disrupt the service. The A language offers a very high-level view on the system, similarly the assertions are describing high-level goals (not only performance related). An example checking that load balancing works is displayed in Listing 8.5.

Similarly to Pip, A targets large distributed systems while SPL focuses on smaller building blocks. Unlike in Pip, Tjang et al. use outlier filtering to achieve more statistically sound results. The A language does not directly support any relative comparisons like SPL but because it focuses on very high-level goals, it makes more sense to use absolute time limits. The described system can be used both in production as well as in testing; expected use is at the boundary of these two where the operator would validate updates before deploying them to production.

Liu et al. [58] took a different approach where the developer imperatively writes the checking code. Their D³S checker targets distributed systems and is capable of checking correctness as well as performance expectations.

The system consists of a so-called state-exposer that is dynamically injected into the running system and exposes the current state of the system as a set of tuples. D³S takes care of global ordering and runs individual verifiers that check the individual expectations. When an expectation is violated, the system is able to reconstruct the sequence of state changes leading to the problem.

Similarly to Pip and A, D³S targets distributed systems. Unlike SPL, D³S does not provide any performance evaluation out-of-the-box and it is the responsibility of the developer to actually evaluate the performance. Probably,
SPL interpretations can be injected as the evaluation engine of performance checks and thus could be used to test production environments.

Bornholt et al. [9] introduced the concept of an “uncertain” type that represents a numerical data type storing not only the value but also its precision (distribution). Arithmetic operations can then operate with the extra information and track the measure of uncertainty across operations. As an example, they demonstrate that typical GPS applications do not use the internal information about precision (as given by the GPS sensor itself), giving the users misleading information about their position. By incorporating the Uncertain<T> type, the code is virtually unchanged but the precision information is not lost.

Although the authors do not mention usage of their approach for performance, the “uncertain” type is well suited for operations on performance data and can be easily used for runtime adaptation based on performance. We have envisioned usage of SPL for runtime adaptation in [13] by incorporating SPL formulas into the source code. In this sense, Uncertain<T> is another way how to bind performance data into the code.

We were not able to find many works that deeply studied the effects of performance documentation of (low-level) API. Monperrus et al. [64] surveyed how various documentation directives (e.g. @throws in Javadoc) are used, how they help understand the API and in general how API documentation is organized. Related to our work is the observation that performance-related documentation is rarely present and is often vague. In this context, propagating the @SPL annotation to Javadoc should improve the documentation considerably.

Stylos et al. [87] created a Jadeite tool that enhances standard Javadoc by looking at actual API usage and extends the documentation with examples and increases visibility of the most often used classes and methods. We mention this tool as it might be an interesting precursor for creating performance unit tests in SPL – it may help identify classes and methods that should be targeted first.

### 8.2 Performance Benchmarking

A brief overview of benchmarking frameworks (mostly targeting the Java environment) is given in Table 8.1. In the following text we describe the frameworks in more detail, focusing on comparison with SPL.

We have conducted a survey of usage of these frameworks in open source projects [85]. In that study we focused on frameworks that are technologically close to SPL, such as JMH or ContiPerf. Surprisingly, these frameworks are rarely used: out of almost 100 thousand projects, less than 0.5% use them. Yet we feel we need to mention and analyze them as their existence and their features and flaws helped to shape the SPL toolchain.

JUnitPerf [18] and ContiPerf [4] are extensions of the JUnit testing framework [47], which targets functional unit testing of Java programs. Since both extensions are similar, they will be discussed together. Both extensions wrap
### Table 8.2: Comparison summary of benchmarking frameworks.

<table>
<thead>
<tr>
<th>Framework</th>
<th>Scope</th>
<th>Metrics</th>
<th>Asserts</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPL</td>
<td>Unit testing</td>
<td>Time</td>
<td>yes</td>
<td>CSV, charts</td>
</tr>
<tr>
<td>JMH [71]</td>
<td>Microbenchmarks</td>
<td>Time</td>
<td>no</td>
<td>Text, JSON</td>
</tr>
<tr>
<td>Caliper [50]</td>
<td>Microbenchmarks</td>
<td>Time</td>
<td>no</td>
<td>Text</td>
</tr>
<tr>
<td>JUnitPerf [18]</td>
<td>JUnit extension</td>
<td>Time</td>
<td>yes</td>
<td>Text</td>
</tr>
<tr>
<td>ContiPerf [4]</td>
<td>JUnit extension</td>
<td>Time</td>
<td>yes</td>
<td>CSV, charts</td>
</tr>
<tr>
<td>Japex [73]</td>
<td>Microbenchmarks</td>
<td>Time</td>
<td>no</td>
<td>XML, charts</td>
</tr>
</tbody>
</table>

- a Charts are embedded in an HTML report.
- b Throughput is computed as well.
- c Linux perf subsystem can be used too.
- d Human-readable text (machine-processing possible but not straightforward).

**Listing 8.6:** Example of a performance test in JUnitPerf. Boilerplate code is omitted for brevity (e.g. suite creation or set-up and tear-down methods).

```java
class MyPerformanceTestCase extends TestCase {
  public void testSomething() {
    // Actual code that would be measured
  }
}

class MyPerformanceTest {
  public static Test suite() {
    Test inner = new MyPerformanceTestCase();
    // Max 1000ms plus 100ms as a tolerance
    return new TimedTest(inner, 1000 + 100);
  }

  public static void main(String args[]) {
    TestRunner.run(suite());
  }
}
```

normal JUnit test cases as performance tests. Such tests are executed multiple times and/or in multiple threads and their duration is then compared with an absolute time limit that is part of the test specification.

For a JUnitPerf test the developer has to wrap the actual test inside a TimedTest (or similar) as can be seen in Listing 8.6. ContiPerf uses a different approach where the test is annotated with specification (1) how to run the test and (2) what is the performance requirement – see Listing 8.7.

Unlike SPL, JUnitPerf and ContiPerf rely on absolute time limits to specify the performance assertions. That simplifies the encoding of the expectation and specification of the test as each test becomes a completely standalone unit but it also makes the tests less portable (recall Section 3.1). The use of JUnit as the foundation simplifies the deployment of both frameworks, which is an aspect where the SPL framework is lacking.

133
Listing 8.7: Example of a performance test in ContiPerf.

```java
public class MyPerformanceTestCase {
    @Test @PerfTest(invocations = 100, threads = 5)
    @Required(max = 1000, average = 500)
    public void testSomething() {
        // Actual code that would be measured
    }
}
```

Listing 8.8: Example of a performance benchmark in Caliper (left) and JMH (right). Both code examples are using the dce variable as a way to prevent dead code elimination – otherwise the compiler might detect that the method has no side-effect and might decide to eliminate the body completely (thus measuring an empty method call only).

```java
public class MyBenchmark {
    @Benchmark
    public int myBenchmark(int reps) {
        int dce = 0;
        for (int i = 0; i < reps; i++) {
            // Benchmarked code
        }
        return dce;
    }
}
```

Caliper [50] and JMH [71] are Java benchmarking frameworks. Neither Caliper nor JMH provide any means for capturing the assumptions about performance as their focus is solely on measurement, not on evaluation.

Both frameworks are very similar in how the users prepare the benchmarks. Caliper and JMH use annotations to mark the method with the code that is to be measured; Caliper expects that the user writes the loop running the code multiple times while JMH generates such loop automatically. The difference is illustrated in Listing 8.8.

Although SPL tools use several techniques to ensure high precision and accuracy of the measurements, it lacks some subtle features that help Caliper and JMH provide more reliable results. Especially JMH takes great care to ensure that no inadvertent optimizations take place [80]. In this sense, SPL would benefit when integrated with JMH – where JMH offers high measurement precision but is lacking on the evaluation side, SPL provides reliable evaluation with less perfect benchmarking capabilities. To this end, an experimental Maven plugin is available [84].

Another Java benchmarking framework is Japex [73]. This framework is no longer maintained (latest commit is from 2011) and we mention it here only for completeness. To write a benchmark in Japex the developer has to prepare a special class for each benchmark and a separate XML configuration file. Otherwise Japex is similar to JMH or Caliper: it focuses on correct benchmarking with rather simplistic evaluation.
Performance unit testing is also technologically related to performance monitoring. We picked Skoll \cite{76}, DataMill \cite{70} and Kieker \cite{36} as representative examples of existing frameworks that might bring more benefits when combined with SPL.

Porter et al. \cite{76} introduced the Skoll project as a decentralized solution for continuous quality assurance (QA). Users enter QA scripts that describe criteria the application has to pass. Skoll then takes care of scheduling these scripts at individual nodes and collecting their results. The scheduling is feedback-driven as dependent tasks are scheduled based on the results of the previous tasks. Multiple strategies for isolating or refining the root causes are available – the user needs to provide only the actual QA conditions.

We believe there are no principal obstacles to using SPL as the QA engine within Skoll to map SPL to larger applications.

The DataMill project by de Oliveira et al. \cite{70} offers a specialized environment for running arbitrary tests. The purpose of the project is to give researchers and testers access to a heterogeneous environment – both in the underlying hardware as well as in the installed software such as the operating system – yet provide reproducible results. No actual analysis is integrated in DataMill, users are expected to download the results and process them separately.

Since DataMill is evaluation-agnostic, it would be possible to use it for collecting data from performance unit tests. This would allow the developers to practically assess the degree of platform independence of the tests as we did with the JDOM library (see Section \ref{sec:jdom} or \cite{38}).

The Kieker Framework \cite{36} is a prominent representative of a Java performance monitoring framework. Kieker gives the user tooling for complete performance analysis – it instruments the running application, and collects and evaluates the performance data. Both the instrumentation and the evaluation is pluggable and the user can provide his own implementation.

Again, we believe that integrating SPL as the evaluation engine into Kieker should meet only technical obstacles but no principal ones.

### 8.3 Anomaly Detection

A brief overview of approaches for detection of anomalies in performance data is given in Table \ref{tab:anomaly}. In the following text we describe the approaches in more detail, focusing on comparison with SPL.

Lee et al. \cite{54} propose to use Cumulative Sum Shewhart charts to detect anomalies in performance of a database engine. These charts are a well established procedure for quality assurance originating in the manufacturing processes. They trigger an alarm based on several criteria that typically include deviating too much from the mean (usually more than $3\sigma$) or not oscillating around the mean (e.g. all values are, albeit slightly, bigger than the mean). The authors show that such conditions work well for software performance and can detect changes smaller than $1.5\sigma$. An interesting part
Table 8.3: Comparison summary of anomaly detection approaches. Absolute data comparison means that the method allows to compare the data with an absolute limit, e.g., it is possible to specify that certain operation must finish within a fixed time limit (e.g., page must be loaded within 5 s). Relative data comparison allows to compare two data sets and determine their relation (less/equal/greater than); this kind of comparison is required for automated anomaly detection where historical data is available but absolute limit might not be.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Data comparison</th>
<th>Comparison method</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPL</td>
<td>relative, absolute</td>
<td>t-test, custom statistical tests</td>
</tr>
<tr>
<td>Lee et al. [54]</td>
<td>relative</td>
<td>CUSUM-Shewhart charts</td>
</tr>
<tr>
<td>Foo et al. [31]</td>
<td>relative</td>
<td>machine learning</td>
</tr>
<tr>
<td>Heger et al. [34]</td>
<td>relative</td>
<td>ANOVA, bisection</td>
</tr>
<tr>
<td>Fahringer, Júnior [29]</td>
<td>relative, absolute</td>
<td>value normalization</td>
</tr>
</tbody>
</table>

* Absolute comparisons are not available for all interpretations.*

of the paper is the initial calibration of the system. Traditionally, the charts are initialized with values coming from the specification (e.g., desired product dimensions) but that is often not possible with software. Therefore the authors also designed an automated method that self-calibrates appropriate window sizes of the moving average to “bootstrap” the whole process. The approach is complemented with root cause detection to automate the process as much as possible.

This approach looks that it could be used as another SPL interpretation. However our initial experiments shown that the application is not straightforward. First of all, measurements of individual components tend to oscillate a lot, triggering false alarms. This can be suppressed by aggregating the values (e.g., using arithmetic mean) but that also hides small regressions that our interpretations were able to detect. The self-calibration process did not help with the data we usually see. When SPL would be used for system-wide testing, this approach should be usable as an alternative interpretation.

Foo et al. [31] focus on detection of performance regressions in the whole system by using load tests. In their approach, many performance metrics are collected during the test (ranging from CPU utilization and I/O over to application specific ones such as request arrival rates). The metrics are then collated and discretized so that the system behavior is described in regular intervals by a vector of metrics. Every metric in each interval is converted to low/medium/high category that corresponds to how the metric deviated from the average at that point of time. On these vectors machine-learning techniques are used to compare the current run of the test with previous ones and to flag and rank regressions.

The big advantage of this approach over SPL is that it offers a ranking of the regressions, thus giving the developer a sense of what issues matter the most. Unlike in SPL, the categorization of metric values (low, medium, high) may hide differences in variation that alone may indicate a problem.
– something that our interpretations (recall Section \ref{sec:3.2.1}) take into account. Extending SPL to collect more metrics is technically possible but our experiments have shown that additional metrics are usually tightly correlated in performance unit tests.\footnote{Because of the nature of the performance unit tests, we were limited to hardware counters as the only available sensors. The unit tests are typically only CPU and/or memory bound so I/O or even network traffic is virtually nil; none of the tests could have provided application-specific metrics. JVM-related metrics (GC counts etc.) seem to fluctuate too much and only add noise to the data. Our experiments have shown that the hardware counters are usually extremely stable and thus would either always fit into the “medium” category or – with smaller width of this category – would fluctuate too much, again creating only noise.} We have not tried any machine-learning approaches to evaluate data from the performance unit tests we worked with. However, not enough detail is given by Foo et al. in \cite{Foo1988} to permit replicating the process and the categorization of values into low/medium/high seems to be manually tuned to work well in the case studies.

Recognizing a performance defect is usually only the first step towards fixing it. Heger et al. \cite{Heger1984} created a system (PRCA – Performance regression Root Cause Analysis) that bisects through the commit history and after identifying the commit responsible for the regression does a call-tree analysis to pinpoint the root-cause function. The authors use performance-aware functional unit tests as readily available benchmarks to gather performance data. The approach relies on using ANOVA to decide whether there is a regression in a given commit. An iterative approach is used for identification of the root cause. First, the whole call tree is recorded. After that, the tool goes from root to the leaves and in each iteration, methods at different levels are measured. This ensures minimal overhead. The tool then looks for differences between performance of the call trees of the two compared commits. The call tree with the function identified as a probable root cause is then reported to the user. The authors demonstrated the quality of their approach on an example of the Apache Commons Math library \cite{Apache2008} where they were able to quickly identify a performance issue that in reality went unnoticed for more than half of a year.

The described approach adds the very nice feature of root-cause identification; something we have not tried to add to SPL. The bisection relies on Git so extending SPL towards that goal would certainly be possible. The authors decided to reuse functional unit tests that are “big enough” to serve as benchmarks. In our experience, most JUnit-based tests are very small and often target corner cases – i.e. cases that are generally not good workload generators. Commons Math is rather an exception with huge test suite where it is possible to find reasonable tests that can double as benchmarks. In this sense we believe that relying on special performance tests is generally better. We also note that the use of ANOVA could be replaced by a traditional two-group statistical test and a more robust evaluation could be used (e.g. approaches mentioned in Section \ref{sec:3.2.1}). Nevertheless the PRCA approach is certainly something to keep in mind when extending the SPL toolchain.

An interesting approach for devising a portable metric was described by Fahringer and Júnior \cite{Fahringer2009}. Instead of trying to somehow capture platform
Table 8.4: Comparison summary of overhead analysis approaches.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Calibration</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ours</td>
<td>Run twice</td>
<td>SPECjbb2015 [82]</td>
</tr>
<tr>
<td>Malony, Shende</td>
<td>On start, adjusts on-line</td>
<td>NAS-PB [68]</td>
</tr>
<tr>
<td>van Hoorn et al.</td>
<td>Microbenchmark</td>
<td>Industrial [94]</td>
</tr>
<tr>
<td>Waller et al.</td>
<td>Microbenchmark</td>
<td>MooBench [49]</td>
</tr>
<tr>
<td>Eichelberger et al.</td>
<td>Benchmark</td>
<td>SPECjvm2008 [83]</td>
</tr>
</tbody>
</table>

\(^a\) Subtracts overhead in nested calls.
\(^b\) Photo service provider portal (CS-1) and EJB-based four-tier architecture with web-frontend (CS-2).

differences, all performance properties are expressed as booleans accompanied by severity and confidence, both of these must fit into the \([0, 1]\) range to simplify their comparison. The authors target parallel applications where mapping the severity/confidence to interval \([0, 1]\) intuitively expresses how well the software exploits parallel execution. Finding an anomaly is then reduced to mere comparison of two numbers.

The authors do not describe how their approach handles program restarts as that changes the issue of comparison of two numbers to the issue of comparison of a set of numbers. Based on the data shown in the paper it seems that the experimental programs exhibit such (large) differences in their performance that restarts would not be an issue anyway (i.e. all values in one set would be bigger than any value in the other one). Despite this shortcoming, the approach is very interesting as it provides a reasonable simplification of possibly complex data sets that is easy to visualize for the end-user.

This approach could be applied to performance unit testing as well. But as our experiments have shown, multiple restarts are typically needed to assess the performance reasonably well. Therefore, we would need to extend the approach to handle multiple values and it is not clear whether we would arrive at significantly different results than when working with the original data.

8.4 Accuracy and Overhead of Performance Monitoring

A brief overview of papers dealing with the accuracy and overhead of runtime performance monitoring is given in Table 8.4. In the following text we describe the approaches in more detail, focusing on comparison with our methodology and results.

Malony and Shende [59] have investigated the issues of performance measurement overhead in the context of the Tau Performance System [78]. Their paper contains a very detailed explanation what are the potential issues of runtime performance monitoring and why it is important to know the actual overhead of the measurements (another valuable information on
this topic is available in [60]). In their work they did one more step than we did in our work – they subtract the (assumed) overhead from the reported results to give user as precise data as possible. Since the overhead cannot be measured precisely without adding a new source of it, they calibrate on application start. The system calls several empty functions to measure the overhead of the measurement infrastructure before the main application is launched. The system is then able to compensate even overhead incurred in nested measurements. Malony and Shende did their experiments on the NAS parallel benchmarks [68].

The method proposed by Malony and Shende relies on the assumption that the overhead is basically constant and start-up calibration is enough. In this sense, they balanced quite nicely the issue of providing more accurate data (compared to measurement without overhead compensation) and keeping the overhead reasonably small (simple extension of existing infrastructure).

From this point of view, our experiments focused on the stability of the calibration. We have shown that the overhead depends on many factors and is not constant during application execution. This renders the results reported by Malony less accurate. Nevertheless, if the overhead estimates were updated continuously, the reported results should be quite accurate. Unfortunately, these updates would surely introduce another source of overhead, so they would have to be performed only occasionally. The reported values would then be adjusted by the latest self-calibration.

A slightly different approach to measure the overhead of runtime monitoring was described by van Hoorn et al. [35] in the context of the Kieker Framework [36]. The experiment had two parts – one using a microbenchmark to measure the actual overhead, second one using real-life monitoring tasks to assess the observed overhead of the whole application. The microbenchmark was used to measure the overhead of the individual instrumentation components. To determine the ground truth, the measured method was actively spinning for a defined time interval. To assess the effects of nested measurements in a call chain, a recursive method was used (with the desired depth as a parameter and active spinning in the leaf call).

The authors were interested in classifying multiple sources of overhead – the overhead of using Kieker to prepare insertion points for the probes, the sensor itself and the overhead of writing the measured value to a disk. For the microbenchmark, the overall overhead was less than 2.5 µs with very low variation. For the real-life tasks, Kieker was used to monitor industrial applications. No specific numbers are provided and because the monitored systems were production ones, getting baseline truth would be at least problematic. The authors limit themselves to stating that the “[industrial partners] could not observe any perceivable runtime overhead [35]”.

A very similar study on Kieker was done later by Waller et al. [94]. They also studied and formalized the individual components that add to the total overhead (sensors, storage etc.). Similarly to van Hoorn et al. [35] they use a microbenchmark to evaluate the overhead called MooBench that is available for download [49]. Also Ehlers et al. [26] measured the overhead of Kieker instrumentation confirming results of van Hoorn et al. [35] that non-active
probes add negligent overhead. A non-active probe is a probe where the
method is instrumented but the action part of the instrumentation is empty
(but can be later replaced by the sensor reading code).

We extend the results of van Hoorn et al. [35] and Waller et al. [94] in
multiple directions (because very little detail is given about the workload in
the industrial cases studies, we will focus on the microbenchmarking part of
the evaluation). First of all, we measured the overhead on a wide variety of
functions, thus better assessing the limits of accuracy. On the other hand,
vvan Hoorn et al. and Waller et al. measured the overhead on an established
monitoring platform while our experiment used the simplest solution that
provided results for our particular experiment. Another improvement we
made was that our experiment was using a highly parallel application while
the Kieker experiments were run on a single-threaded microbenchmark. In
our setup we were unable to precisely control the depth of the call-stack and
thus we were not able to evaluate precisely the issues of nested measure-
ments. But that is the price for using a standard benchmark instead of an
artificial one. Also, by using a large application we have probably seen a
wider variety of interactions between the code, the execution platform and
the JIT compiler. Our findings confirmed that these interactions can influ-
ence the performance significantly, especially for small (fast) methods. That
is something that is not discussed in [35] but probably because Kieker is used
for continuous monitoring and thus most of the time only large methods are
instrumented. But unlike in [35] we have not investigated what are the indi-
vidual components that contribute to the overhead (sensors vs data storage
vs synchronization etc.).

An interesting comparison of overhead of several monitoring frameworks
was done by Eichelberger and Schmid [27]. They compare the overhead of
OpenCore 2, Kieker [36] and their own tool, SPASS-meter. The authors used
the SPECjvm2008 [83] benchmark suite as the workload when measuring the
overhead. The overhead of the monitoring frameworks was measured by dif-
ference in the benchmark score with and without the instrumentation. It is
unclear what was actually monitored – the text mentions collecting the “ex-
ecution time of all separate functions of the 37 individual benchmarks [27]”,
which suggest that every function was instrumented, but the results contradict this
by declaring an overhead of less than 1%, which would be practically im-
possible when measuring all functions.

Compared to our work, Eichelberger and Schmid evaluated the overhead
on a wider variety of workloads as the whole SPECjvm2008 suite was used.
Unfortunately, the text lacks information that is critical for more in-depth
comparison. As mentioned above, it is unclear what was actually instru-
mented: we have instrumented several hundreds of methods which slowed-
down the application by factor of four. Furthermore, the authors state that
they restarted the JVM five times for each configuration (monitoring frame-
work, collected metrics etc.). Some information is probably missing because
in our experience, the reported overhead of several percents is in the range

\[ \text{http://opencore.jinspired.com/} \] does not work any more as of
March 2018.
of the accuracy of the results that can be obtained with such (rather low) number of restarts.

In our work, we focused solely on the execution time of the function measured by the wall clock. Eichelberger and Schmid [27] compared multiple metrics and our work is lacking in this sense. However, extending our framework to collect other metrics is certainly possible and we are not aware of any principal obstacles.

Many works include assessment of overhead of the used (monitoring) solution but stop short of evaluating where the limits are. A notable exception is the work by Parsons et al. [74] that analyses non-intrusive tracing of J2EE systems. Their solution is built on top of the COMPASS framework [66]. The core of the paper is not directly related to our work but it contains an interesting evaluation of the performance overhead. The performance overhead is evaluated by emulating different user load of the tested system, in two different settings: with and without the probes. Unfortunately, the paper does not contain much of raw data and provides only a generic observation that unless the system is saturated, the overhead is negligible. The authors identified that with 100 users the system is still not fully loaded and the overhead is not noticeable while with 150 users the system is close to being saturated and overhead rose to almost 20%. But the authors have not investigated further where the breaking point is when the overhead starts to have a noticeable impact on the overall performance. The assumption is that when the system is almost fully saturated, the overhead of the probes utilizes the remaining power and saturates the machine completely.

Also somewhat related is the work of Dufour et al. [23] where the authors studied the overhead of using different AspectJ [25] constructs. Their approach is mostly based on tagging bytecode instructions with their origin – i.e. whether the instruction was present in the original application code, or comes from the aspect itself, or is part of a helper code of AspectJ (i.e. the actual overhead). Based on various bytecode-related metrics the authors were able to classify what aspects (advices) are cheaper or more expensive to use. Unsurprisingly, the slowest advices are those that need information available only at runtime (control flow etc.) and even in some obvious static-information-only cases, the AspectJ compiler was unable to optimize the modified code. The cited work is relatively old so it is difficult to draw conclusions relevant to more current just-in-time compilers and the current version of AspectJ.

In our work [39] we heed the advice from [23] and strictly use only aspects/advices where all the information was available at weave-time. We have used the static weaving features of AspectJ – i.e. producing woven code offline – and manually checked that the modified bytecode does not contain any extra instructions.
Conclusion

This thesis presents our work on increasing the developers’ awareness of performance in the context of agile software development. We argue that while functionality checking permeates all phases of software development, the same emphasis is not present for software performance. The developers are often aware of performance in the big but not in the small. For testing in the big, classical system-wide tests can reveal both functional and performance issues. For testing in the small, developers can insert assertions and add unit tests to check code against functional bugs, but there is no widely accepted practice of doing this for software performance in as similarly lightweight and straightforward manner.

The contributions presented in this work focus on automating the performance testing of individual components and on documenting component performance. There is also presented the practical accuracy limits of performance measurements in production environments through dynamic instrumentation, and the challenges of detecting regressions in performance datasets automatically.

Our contribution to each of the goals stated in Chapter 2 is summarized in more detail in the following sections.

9.1 G1 (Performance Testable Code)

Our contribution in the area of performance-testable code is twofold. We have designed and developed a prototype of a testing framework for writing performance unit tests (Section 3.1.2) that allows to quickly detect performance issues on the level of individual components. We have also conducted a retrospective case study on an open-source project (Section 3.1.3),
which demonstrated that our framework can detect performance bugs earlier and possibly lessen the tedious work of finding the root cause. However, the effort needed to integrate our tool in a project is bigger than is usually the norm for unit-testing frameworks. In that sense our ongoing work for integration with JMH seems to be a promising approach.

As for future work, there is still space for improvement of the API for writing tests and especially workload generators. Perhaps using a different language with better support for DSL, such as Scala, would lead to a more readable code and simplify the process of writing performance tests.

### 9.2 G2 (Robust Regression Detection Mechanism)

Because the real benefit of having performance tests is in the ability to detect performance regressions early, we have also contributed new methods for regression detection in the measurement data. Our methods focus on the detection of difference between two data sets (Section 3.2) and are based on traditional statistical tests as well as on bootstrap. We have provided an extensive evaluation on real-world measurements.

Our ongoing work further extends these methods to achieve sufficient regression detection sensitivity even with highly variable benchmarks. One of the issues we aim to address is constructing a benchmark dataset that can be used to assess the efficiency and accuracy of regression detection methods. Obviously, the biggest challenge is in finding performance measurements where we can clearly decide the correct answer without introducing any form of circular reasoning.

### 9.3 G3 (Performance Documentation)

To help document performance of individual software components, we have designed a mechanism that binds the workload generators from performance tests with the application methods the performance of which is to be documented (Section 3.3). Assuming a generator of a typical workload is provided, we use it to measure the actual method performance on the platform of developer’s choice. We then use these measurements to extend the standard API documentation with performance information in a non-intrusive manner. Although our prototype works well, we were not able to validate the benefits of our approach in a user study yet (due to high variability of the study inputs). Our other experiments suggest we are generally moving in the right direction towards improving performance awareness.

We continue the work on our tooling to lower the cost of carrying out a larger case study, which would help us validate the approach more thoroughly. On the technical side, this includes improving the integration of the extended documentation with existing IDE platforms. Adding an opposite function to workload generators – modules that distill significant parameters from the runtime measurements – will also allow us to project performance
of production runs back into the documentation. In DevOps style, the documentation will then report performance of a deployed application rather than measurements from a testing environment.

9.4 G4 (Limits of Runtime Monitoring)

To assess whether it is possible to feed live production data into performance documentation, we have also focused on the practical limits in dynamic monitoring. The limiting factors proved to be not only the precision of the actual sensors, but also the complex interactions across the full hardware and software stack. Our prototype environment is tuned for minimal overhead, which permits investigating the limiting factors in detail (Section 3.4).

As for future work, we want to integrate our tools with established monitoring frameworks such as Dynatrace and use the collected data to provide the performance documentation. We also want to test whether we can evaluate the performance assumptions stated in performance tests with data coming from the production environment. These two improvements should further close the gap between developer assumptions about performance and the actual results from the production environment.
The following is a list of all publications published by the author with their abstracts.


Resource awareness is a key requirement for dynamic adaptation in resource-constrained systems. Achieving resource awareness with clean separation of concerns and reasonable overhead is still a challenge - especially where this awareness concerns runtime performance. Among the difficult issues are for example transparent performance monitoring or platform independent performance evaluation. To advance the current state of the art in resource awareness, we propose a performance awareness framework for the domain of component-based systems. The framework is based on the Stochastic Performance Logic (SPL), which enables explicit description and automatic evaluation of assumptions about performance using logic formulas. We demonstrate the potential of the framework on multiple use-cases and outline extensions that facilitate the runtime resource awareness.


Mobile cloud computing in the context of ad-hoc clouds brings new challenges when offloading computation from mobile devices. The management of application deployment needs to ensure that the offloading provides users with the expected benefits, but it suddenly needs to cope with a highly dynamic environment which lacks a central authority and
in which computational nodes appear and disappear. We propose an approach to the management of ad-hoc systems in such dynamic environment using component ensembles that connect mobile devices with more powerful computation nodes. Our approach aims to address the challenges of scalability and robustness of such systems without the need for central authority, relying instead on simple patterns that lead to reasonable adaptation decisions based on limited and imprecise information.


Including performance tests as a part of unit testing is technically more difficult than including functional tests – besides the usual challenges of performance measurement, specifying and testing the correctness conditions is also more complex. In earlier work, we have proposed a formalism for expressing these conditions, the Stochastic Performance Logic. In this paper, we evaluate our formalism in the context of performance unit testing of JDOM, an open source project for working with XML data. We focus on the ability to capture and test developer assumptions and on the practical behavior of the built-in hypothesis testing when the formal assumptions of the tests are not met.


Ensembles of autonomic components are a novel software engineering paradigm for development of open-ended distributed highly dynamic software systems (e.g. smart cyber-physical systems). Recent research centered around the concept of ensemble-based systems resulted in design and development models that aim to systematize and simplify the engineering process of autonomic components and their ensembles. These methods highlight the importance of covering both the functional concepts and the non-functional properties, specifically performance-related aspects of the future systems. In this paper we propose an integration of the emerging techniques for performance assessment and awareness into different stages of the development process. Our goal is to aid both designers and developers of autonomic component ensembles with methods providing performance awareness throughout the entire development life cycle (including runtime).

Garbage collection is an element of many contemporary software platforms whose performance is determined by complex interactions and is therefore difficult to quantify and model. We investigate the difference between the behavior of a real garbage collector implementation and a simplified model on a selection of workloads, focusing on the accuracy achievable with particular input information (sizes, references, lifetimes). Our work highlights the limits of performance modeling of garbage collection and points out issues of existing evaluation tools that may lead to incorrect experimental conclusions.

The ASCENS project deals with designing systems as ensembles of adaptive components. Among the outputs of the ASCENS project are multiple tools that address particular issues in designing the ensembles, ranging from support for early stage formal modeling to runtime environment for executing and monitoring ensemble implementations. The goal of this chapter is to provide a compact description of the individual tools, which is supplemented by additional downloadable material on the project website.

The ASCENS project works with systems of self-aware, self-adaptive and self-expressive ensembles. Performance awareness represents a concern that cuts across multiple aspects of such systems, from the techniques to acquire performance information by monitoring, to the methods of incorporating such information into the design making and decision making processes. This chapter provides an overview of five project contributions – performance monitoring based on the DiSL instrumentation framework, measurement evaluation using the SPL formalism, performance modeling with fluid semantics, adaptation with DEECo and design with IRM-SA – all in the context of the cloud case study.

Vojtěch Horký, Peter Libič, Antonín Steinhauser, and Petr Tůma. “DOs and DON’Ts of Conducting Performance Measurements in Java”. In: Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering. ICPE
The tutorial aims at practitioners - researchers or developers - who need to execute small scale performance experiments in Java. The goal is to provide the attendees with a compact overview of some of the issues that can hinder the experiment or mislead the evaluation, and discuss the methods and tools that can help avoid such issues. The tutorial will examine multiple elements of the software execution stack that impact performance, including common virtual machine mechanisms (just-in-time compilation and garbage collection together with associated runtime adaptation), some operating system features (timers) and hardware (memory) - although the focus will be on Java, some of the take away points should apply even in a more general performance experiment context.


Many decisions taken during software development impact the resulting application performance. The key decisions whose potential impact is large are usually carefully weighed. In contrast, the same care is not used for many decisions whose individual impact is likely to be small – simply because the costs would outweigh the benefits. Developer opinion is the common deciding factor for these cases, and our goal is to provide the developer with information that would help form such opinion, thus preventing performance loss due to the accumulated effect of many poor decisions. Our method turns performance unit tests into recipes for generating performance documentation. When the developer selects an interface and workload of interest, relevant performance documentation is generated interactively. This increases performance awareness – with performance information available alongside standard interface documentation, developers should find it easier to take informed decisions even in situations where expensive performance evaluation is not practical. We demonstrate the method on multiple examples, which show how equipping code with performance unit tests works.


In managed memory environments, code changes influence performance both through time spent executing the code and time spent collecting garbage generated by the code. This complicates decision making when considering performance impact of code changes—while the impact on execution time is usually easy to assess in isolation, the impact on garbage collection time depends on the memory allocation be-
behavior of the code surrounding the changes. In our paper, we describe a method to estimate the impact of code changes with additional allocations on garbage collection time, which can be applied, e.g., when assessing the overall performance impact of alternative changes. The method is demonstrated on experiments with the HotSpot virtual machine.


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 implemented in a performance testing framework and evaluated in multiple experiments, demonstrating the ability to identify performance differences in realistic unit test scenarios.


In production environments, runtime performance monitoring is often limited to logging of high level events. More detailed measurements, such as method level tracing, tend to be avoided because their overhead can disrupt execution. This limits the information available to developers when solving performance issues at code level. One approach that reduces the measurement disruptions is dynamic performance monitoring, where the measurement instrumentation is inserted and removed as needed. Such selective monitoring naturally reduces the aggregate overhead, but also introduces transient overhead artefacts related to insertion and removal of instrumentation. We experimentally analyze this overhead in Java, focusing in particular on the measurement accuracy, the character of the transient overhead, and the longevity of the overhead artefacts. Among other results, we show that dynamic monitoring requires time from seconds to minutes to deliver stable measurements, that the instrumentation can both slow down and speed up the execution, and that the overhead artefacts can persist beyond the monitoring period.


Although methods and tools for unit testing of performance exist for
over a decade, anecdotal evidence suggests unit testing of performance is not nearly as common as unit testing of functionality. We examine this situation in a study of GitHub projects written in Java, looking for occurrences of performance evaluation code in common performance testing frameworks. We quantify the use of such frameworks, identifying the most relevant performance testing approaches, and describe how we adjust the design of our SPL performance testing framework to follow these conclusions.


Basic topics from probability and statistics – such as probability distributions, parameter estimation, confidence intervals and statistical hypothesis testing – are often included in computing curricula and used as tools for experimental performance evaluation. Unfortunately, data collected through experiments may not meet the requirements of many statistical analysis methods, such as independent sampling or normal distribution. As a result, the analysis methods may be more tricky to apply and the analysis results may be more tricky to interpret than one might expect. Here, we look at some of the issues on methods and experiments that would be considered basic in performance evaluation education.


Modern software systems often employ dynamic adaptation to runtime conditions in some parts of their functionality – well known examples range from autotuning of computing kernels through adaptive battery saving strategies of mobile applications to dynamic load balancing and failover functionality in computing clouds. Typically, the implementation of these features is problem-specific – a particular autotuner, a particular load balancer, and so on – and enjoys little support from the implementation environment beyond standard programming constructs. In this work, we propose Adaptive Dispatch as a generic coding pattern for implementing dynamic adaptation. We believe that such pattern can make the implementation of dynamic adaptation features better in multiple aspects – an explicit adaptation construct makes the presence of adaptation easily visible to programmers, lends itself to manipulation with development tools, and facilitates coordination of adaptation behavior at runtime. We present an implementation of the Adaptive Dispatch pattern as an internal DSL in Scala.
Bibliography


