Utilizing Performance Unit Tests To Increase Performance Awareness

Vojtěch Horký Peter Libič Lukáš Marek
Antonín Steinhauser Petr Tůma

Department of Distributed and Dependable Systems
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

ICPE'15, Jan. 31–Feb. 4, 2015, Austin, Texas, USA.
Copyright © 2015 ACM 978-1-4503-3248-4/15/01 ...$15.00.
http://dx.doi.org/10.1145/2668930.2688051.

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 `HashMap` 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.[1] 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.

[1] But 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.

```
Map<String, Double> get(Document doc) {
    Map<String, Double> result
        = new HashMap<>();

    XPathExpression<Element> expr
        = XPathFactory.instance().compile(
        "/rec/purchase", Filters.element());

    for (Element e : expr.evaluate(doc)) {
        String customer
                = e.getChildText("customer");
        double price = Double.parseDouble(
                e.getChildText("price"));
        Double sum = result.get(customer);
        if (sum == null) sum = price;
        else sum += price;
        result.put(customer, sum);
    }

    return result;
}
```

(1.a) XPath and HashMap

```
Map<String, Double> get(Document doc) {
    Map<String, Double> result
        = new TreeMap<>();

    List<Element> purchases
        = doc.getRootElement()
        .getChildren("purchase");

    for (Element e : purchases) {
        String customer
                = e.getChildText("customer");
        double price = Double.parseDouble(
                e.getChildText("price"));
        Double sum = result.get(customer);
        if (sum == null) sum = price;
        else sum += price;
        result.put(customer, sum);
    }

    return result;
}
```

(1.b) Getters and TreeMap

Listing 1: Alternative implementations of imagined XML processing.

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

[2] And experiments carry their own risks [2, 17, 9].

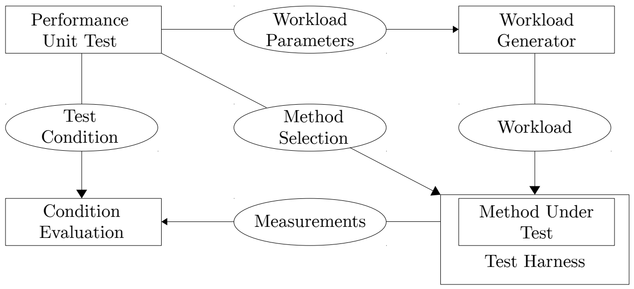

Figure 1: Performance unit test structure.

```
for $i \leftarrow 1$ .. sample count do
    test arguments ← GENERATOR(workload parameters)
    start the measurement
    for all args ← test arguments do
        MEASUREDMETHOD($args[0]$, $args[1]$, ...)
    end for
    stop the measurement
    store the result
end for
```

Listing 2: Repeated measurement with workload generator.

execution phase, the system under test is subjected to the previously prepared workload and the performance is measured. In the validation phase, the observed performance is evaluated against the test criteria. The cleanup phase takes care of preparing for the next test, if any.

Two notable distinctions between performance and functional unit tests are the separation of the workload generator and the test criteria evaluation.

In the setup phase, the unit test code prepares the sequences of input arguments, which will be used in the execution phase to invoke the method under test. In effect, the input arguments determine the unit test workload, we therefore refer to this code as the *workload generator*. Technically, the generator is a standalone method that returns the `Iterable<Object[]>` type. Each `Object[]` contains arguments for a single method invocation, used in a manner similar to the `invoke()` method from the `java.reflection` package.

Making the workload generator a standalone method has two reasons. The first reason is code reuse – we can use the same generator for different unit tests when those tests use the same workload. The second reason is related to the performance measurements. By preparing the workload before the execution phase, we minimize the disruptive influence. We can also easily repeat the test multiple times to obtain more robust results. This process is illustrated in Listing 2.

The measurement results are evaluated against the test criteria using statistical hypothesis testing. The criteria is described in a formalism called Stochastic Performance Logic (SPL) [4], which defines the performance of a method as a random variable with a probability distribution that reflects the *workload parameters*. Workload parameters are arbitrary parameters that characterize the workload, given as arguments to the workload generator – examples of workload parameters used in this paper include sizes of collections to be measured, frequencies of individual collection operations, or sizes of generated graphs.

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 attribute names 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.

queried 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 GRAL [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 selected performance tests 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.

```
class LinkedList {
    @Generator ("LinkedListGen#contains")
    public boolean contains (Object obj) {
        /* ... */
    }
}
```

Listing 4: Binding workload generator generator with the measured method.

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.reflection` 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.

[4] To be completely accurate, the test in our tool implementation can also return a third status that indicates the test does not yet have enough data to decide on the test condition.

```
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

```
@Generator (...)
@ShowWithClass (de.erichseifert.gral.plots.XYPlot.class)
@ShowWithMethod ("de.erichseifert.gral.io.plots.DrawableWriter#write")
void plotLinesToPng (DataTable data, OutputStream output, int width, int height) {
    DrawableWriterFactory factory = DrawableWriterFactory.getInstance ();
    DrawableWriter writer = factory.get ("image/png");
    XYPlot plot = new XYPlot (data);
    LineRenderer lines = new DefaultLineRenderer2D ();
    lines.setSetting (LineRenderer.COLOR, Color.BLUE);
    plot.setLineRenderer (data, lines);
    writer.write (plot, output, width, height);
}
```

Listing 6: Special-purpose test method for a plotting library. Exceptions omitted.

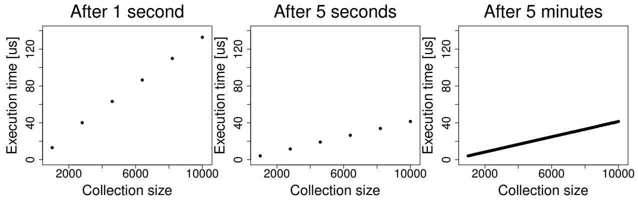

Figure 2: Improving the precision over time.

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.

To address the issue of packages distributed in compiled form, we add extra annotations that can be used to describe the workload parameters in a compact manner, as shown on Listing 7. The annotations serve dual purpose – besides being used when JavaDoc is not available, they also describe the axes of the measurement result plots together with valid parameter ranges.

4.5 Realistic Measurement Context

So far, we have considered measurements collected using workload generators originally designed for performance unit tests. When associated with a component, these generators produce workloads that the component developer considered useful to test – these can be common workloads for the component, workloads that exhibit interesting behavior, or even workloads that target a particular performance regression. However, these workloads can still substantially differ from the workload expected by the application developer. They are also designed to be executed in relative isolation, which makes them similar to classical micro benchmarks.

Correctly interpreting a micro benchmark result is tricky – even with a reasonable workload, the micro benchmark still executes under conditions that can be very different from those in the eventual application. This is not an issue for the performance unit tests, which are designed with the knowledge of the benchmark execution conditions. It is, however, a potential issue when trying to interpret the micro benchmark result with the application conditions in mind. Although our tool permits adding custom workload generators that can remedy this issue, the surest way to determine the behavior of a component in an application is still by measuring it in the application.

Our work on performance awareness in component systems [3] suggests a solution. We use the DiSL framework [31] to instrument the component inside the application and collect measurements in much the same way as with the performance unit tests – except now, the workload is generated by the application itself rather than the workload generator. To determine the workload parameters required for presenting the measurements, we employ sizers as an inverse complement to generators – where the generator produces workload given the relevant parameters, the sizer produces the parameters while observing the workload.

5. EXPERIMENTAL EVALUATION

The ultimate aim of our approach is to improve performance awareness among software developers so that they

```
/** Generator for testing Collection.contains() method.
 *
 * @param size Size of the underlying collection.
 * @param hits Number of searches that hit.
 * @param nohits Number of searches that miss.
 */
@Generator("Collection.contains() with mix of hits and misses.")
public Iterable<Object[]> contains(
        @Param("Collection size", min=10) int size,
        @Param("Searches that hit", min=0) int hits,
        @Param("Searches that miss", min=0) int nohits) {
    /* ... */
}
```

Listing 7: Generator documentation with annotations.

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 3.28 s, but the slowest implementation has not finished in one day. The median execution time was 5.38 s. 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 6.13 s. 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 μ with a relative error exceeding δ, and we want P to remain at a reasonably low confidence level α: $P((\bar{X} - \mu)/\mu) \geq \delta) = \alpha$. Under normality assumptions, reasonable for this particular lower bound computation, we can estimate the minimum sample size $n = (z_\alpha^2 * \sigma^2)/(\delta^2 * \mu^2)$ [34]. For our experiment results, α set to 5 % and δ 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 generator 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 `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 `getChild()` 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 `getChild()` timing does not depend on the child count when the matching child position stays constant. The measurement results collected using such a generator are on Figure 3.

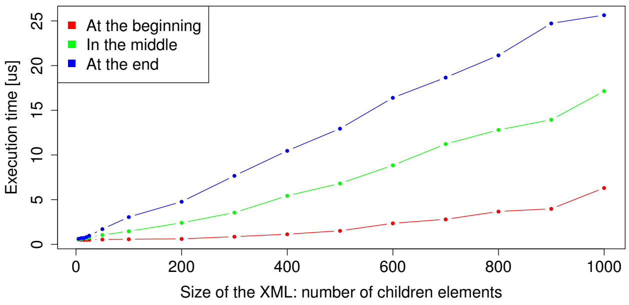

Figure 3: Measurements of `Element.getChild()` for varying child count and selected matching child position.

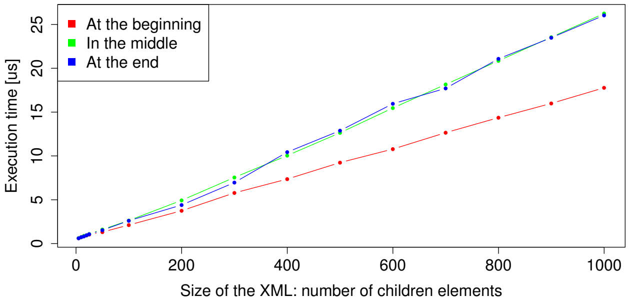

Figure 4: Measurements of XPath query for varying element count and selected matching element position. Compilation amortized over 100 queries.

For the alternative choice with XPath, we do not have to hypothesize what workload generator can be reasonably available. The developers that use Jaxen, the XPath engine used in our experiments, have already implemented a simple performance benchmark [6] that measures the time needed to execute a query that locates a unique node in various positions of a tree with a given size. We have implemented the corresponding workload generator, the measurement results are on Figure 4.

Equipped with the information from Figures 3 and 4, the developer can properly balance the difference in performance with other concerns. In particular, the information helps notify the developer of some performance realities that are not self evident, for example the linear dependency between the `getChild()` time and the position of the matching child among siblings.

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

[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 12 ms. The best combination uses a sequence of getters and `HashMap` in 7 ms.

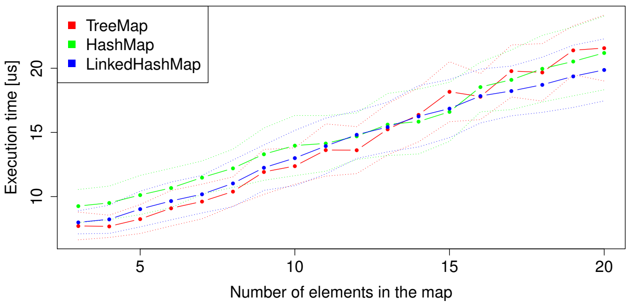

Figure 5: Measurements of operation mix on Map<String, String> collections. Average time, dotted lines at 3σ.

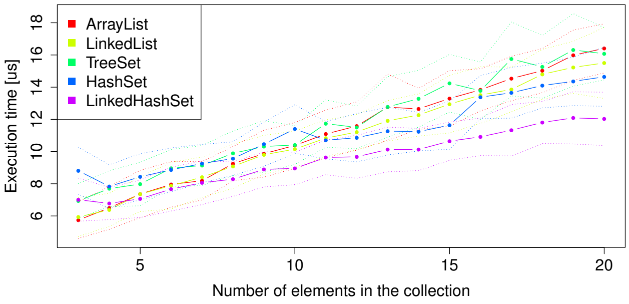

Figure 6: Measurements of operation mix on Collection<String> collections. Average time, dotted lines at 3σ.

element and one that searches for an element that does 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 assume 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 developer realize that in this scenario, all three collections perform 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 collection. 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, trading 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 expect 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.

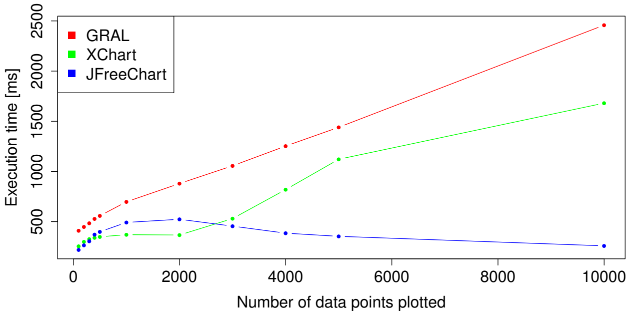

Figure 7: Measurements of line plot creation time. Image dimension 800 × 600 points.

In our scenario, the developer would look at workload generators that plot line charts. The workload parameters of the generators can differ from library to library, the developer 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 parameters 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 dependency on Figure 7 is not strictly monotonous. This behavior 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 instead examine the existing projects ourselves, looking for opportunities for performance improvement based on performance 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 excluded projects that use JDOM merely to read their configuration 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

Table 1: Buildhealth results

	Original	No XPath	Cached XPath
Repeated	938.3 ms	908.4 ms	929.8 ms
Ant task	2.23 s	2.16 s	–
Standalone	2.52 s	2.40 s	–

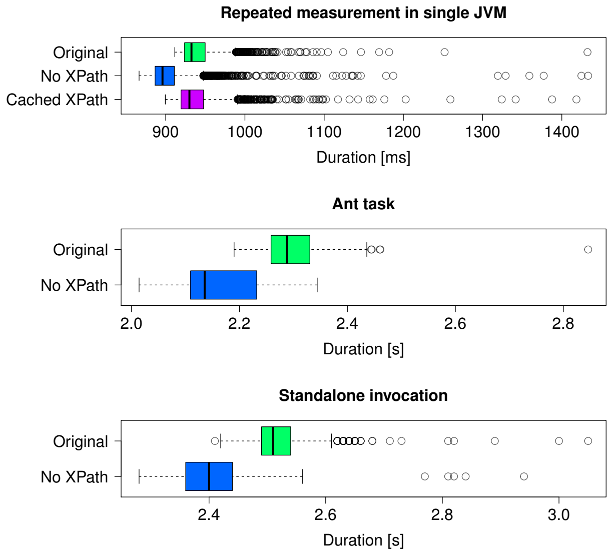

Figure 8: Buildhealth

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 effect of our changes. To filter out the usual warm up effects, we run the core of the Buildhealth utility in an artificial loop. Besides the original and the modified versions of the utility, we also show the performance when the compiled XPath expression is cached. The results indicate the part of 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`[8] 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.

Table 2: METS downloader results

	XPath	Nested `getChildren`
Total time	120.2 s	120.4 s
`getImageURLs`	131.5 ms	27.4 ms

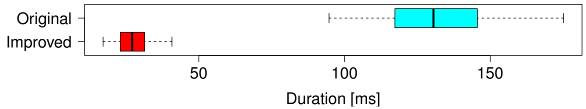

Figure 9: METS downloader, the `getImageURLs` method.

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.[9]

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[10] 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

[8] http://ant.apache.org/manual/listeners.html#ProfileLogger

[9] The sources are available from https://svn.thulb.uni-jena.de/repos/maven-projects/mets-downloader, we are not aware of an official project homepage.

[10] https://code.google.com/p/dynamicreplicaplacement

Table 3: CDN simulator test results

	`getChildren`	`getDescendants`
Total time	1.93 s	1.89 s
Processing alone	744.3 ms	700.3 ms

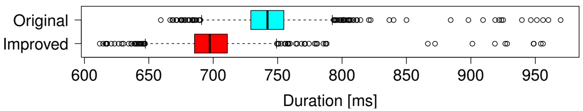

Figure 10: CDN simulation, the `process` method.

needless performance loss and that can be easily fixed. Additionally, we saw that even on an assignment that is short and thus limited in the number of decisions taken during development, the performance of individual implementations can vary significantly.

On the down side, our evaluation is not comprehensive – while we did show cases where our approach is useful, we did not quantify the share of those cases in some representative sample of software development activities. One can easily observe that such quantification will require long term effort and should be expected to deliver results in the form of developer acceptance or rejection, rather than hard numbers. Towards this goal, we continue to develop and improve our prototype tools.

We can also see several potential drawbacks of our approach. Obviously, when the performance unit tests are constructed improperly, the derived performance documentation can provide misleading information. Interpreting measurements is also a skill that requires some experience – in this context, we can only argue that the developers should be trusted to acquire that skill, and eventually develop best practices for writing the documentation.

In a broader view, we should also stress that performance documentation is not a substitute for efficient algorithms. While it can warn the developer of unexpected complexity in the used code, our approach complements, rather than replaces, the need to choose proper algorithms and architectures. Along the same lines, our approach complements techniques such as profiling or parameter tuning, which are useful at other software development stages.

There is also the question of costs vs benefits. Where we point out that our approach increases performance awareness, it also brings costs associated with developing the unit tests and executing the measurements. Possibly, the performance information can draw the developer attention away from other topics, which might have more impact on the overall software performance. Again, these are likely issues that are best evaluated through practice – we can draw a parallel with functional unit testing, which has been shown to improve software quality without adding extra costs [15].

On the related work side, much work has been dedicated to enriching interface specifications with non-functional properties, which often include performance metrics. One such example is the performance-enabled WSDL [13], which adds requirements and assumptions about performance of web services. Where the existing work focuses on automatic service selection at runtime, we aim at providing performance relevant information to the developer.

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 jmh [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 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

[1] K. Beck. *Simple Smalltalk Testing*. Cambridge University Press, 1997.

[2] A. Buble, L. Bulej, and P. Tuma. CORBA benchmarking: a course with hidden obstacles. In *Proc. IPDPS 2003 PMEOPDS*, 2003.

[3] L. Bulej, T. Bures, V. Horky, J. Keznikl, and P. Tuma. Performance awareness in component systems: Vision paper. In *Proc. COMPSAC 2012 CORCS*, 2012.

[4] L. Bulej, T. Bures, J. Keznikl, A. Koubkova, A. Podzimek, and P. Tuma. Capturing Performance Assumptions using Stochastic Performance Logic. In *Proc. ICPE 2012*. ACM, 2012.

[5] O. Burn et al. *Checkstyle*, 2014. http://checkstyle.sf.net.

[6] M. Böhm and J.-J. Dubray. *Dom4J performance versus Xerces / Xalan*, 2008. http://dom4j.sf.net/dom4j-1.6.1/benchmarks/xpath.

[7] *Caliper: Microbenchmarking framework for Java*, 2013. http://code.google.com/p/caliper.

[8] Y. Chen, R. H. Katz, and J. D. Kubiatowicz. Dynamic replica placement for scalable content delivery. In *Proc. IPTPS 2002*. Springer, 2002.

[9] C. Click. The Art of Java Benchmarking. http://www.azulsystems.com/presentations/art-of-java-benchmarking.

[10] C. Ţăpuş, I.-H. Chung, and J. K. Hollingsworth. Active Harmony: Towards automated performance tuning. In *Proc. SC 2002*. IEEE, 2002.

[11] *DocBook*, 2014. http://www.docbook.org.

[12] *Document Object Model*, 2005. http://w3.org/DOM.

[13] A. D'Ambrogio. A WSDL extension for performance-enabled description of web services. In *Proc. ISCIS 2005*. Springer, 2005.

[14] *Eclipse*, 2014. http://www.eclipse.org.

[15] M. Ellims, J. Bridges, and D. Ince. The economics of unit testing. *Empirical Software Engineering*, 11(1), 2006.

[16] T. Fahringer and C. S. Júnior. Modeling and detecting performance problems for distributed and parallel programs with JavaPSL. In *Proc. SC 2001*. ACM, 2001.

[17] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. In *Proc. OOPSLA 2007*. ACM, 2007.

[18] *GRAL*, 2014. http://trac.erichseifert.de/gral.

[19] *Guava: Google Core Libraries for Java 1.6+*, 2014. http://code.google.com/p/guava-libraries.

[20] V. Horký, F. Haas, J. Kotrč, M. Lacina, and P. Tůma. Performance Regression Unit Testing: A Case Study. In *Proc. EPEW 2013*. Springer, 2013.

[21] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12), Dec. 2004. http://findbugs.sf.net.

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

[23] *Japex Micro-benchmark Framework*, 2013. https://java.net/projects/japex.

[24] *Javadoc Tool*, 2014. http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html.

[25] *Jaxen*, 2013. http://jaxen.codehaus.org.

[26] *JDOM*, 2013. http://www.jdom.org.

[27] *JFreeChart*, 2013. http://www.jfree.org/jfreechart.

[28] *JMH: Java Microbenchmark Harness*, 2014. http://openjdk.java.net/projects/code-tools/jmh.

[29] T. Kalibera, L. Bulej, and P. Tuma. Benchmark precision and random initial state. In *Proc. SPECTS 2005*, 2005.

[30] L. Madeyski. *Test-Driven Development: An Empirical Evaluation of Agile Practice*. Springer, 2010.

[31] L. Marek et al. DiSL: a domain-specific language for bytecode instrumentation. In *Proc. AOSD 2012*, 2012.

[32] *Metadata Encoding and Transmission Standard*, 2014. http://www.loc.gov/standards/mets.

[33] N. Mitchell and G. Sevitsky. The causes of bloat, the limits of health. In *Proc. OOPSLA 2007*. ACM, 2007.

[34] *NIST/SEMATECH e-Handbook of Statistical Methods*, 2014. http://www.itl.nist.gov/div898/handbook.

[35] *OpenBenchmarking.org: An Open, Collaborative Testing Platform For Benchmarking & Performance Analysis*, 2014. http://openbenchmarking.org.

[36] *Primitive Collections for Java*, 2003. http://pcj.sf.net.

[37] O. Shacham, M. Vechev, and E. Yahav. Chameleon: Adaptive selection of collections. In *Proc. PLDI 2009*. ACM, 2009.

[38] *ACE+TAO+CIAO+DAnCE Distributed Scoreboard*, 2014. http://www.dre.vanderbilt.edu/scoreboard.

[39] *SPL Tools*, 2013. http://d3s.mff.cuni.cz/software/spl.

[40] *Trove*, 2012. http://trove.starlight-systems.com.

[41] *Xeiam XChart*, 2014. http://xeiam.com/xchart.jsp.

[42] *XML Path Language (XPath) 2.0*, 2010. http://w3.org/TR/xpath20.

[43] G. Xu, M. Arnold, N. Mitchell, A. Rountev, and G. Sevitsky. Go with the flow: Profiling copies to find runtime bloat. In *Proc. PLDI 2009*. ACM, 2009.

[44] G. Xu et al. Software bloat analysis: Finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In *Proc. FoSER 2010*. ACM, 2010.

[45] H. Yu, D. Zhang, and L. Rauchwerger. An adaptive algorithm selection framework. In *Proc. PACT 2004*. IEEE, 2004.