Towards Using Code Coverage Metrics for Performance Comparison on the Implementation Level

Mathias Menninghaus
mathias.menninghaus@uos.de

Elke Pulvermüller
elke.pulvermueller@uos.de

University of Osnabrück, Institute of Computer Science
Albrechtstraße 28, 49069 Osnabrück, Germany

ABSTRACT

The development process for new algorithms or data structures often begins with the analysis of benchmark results to identify the drawbacks of already existing implementations. Furthermore it ends with the comparison of old and new implementations by using one or more well established benchmark. But how relevant, reproducible, fair, verifiable and usable those benchmarks may be, they have certain drawbacks. On the one hand a new implementation may be biased to provide good results for a specific benchmark. On the other hand benchmarks are very general and often fail to identify the worst and best cases of a specific implementation. In this paper we present a new approach for the comparison of algorithms and data structures on the implementation level using code coverage. Our approach uses model checking and multi-objective evolutionary algorithms to create test cases with a high code coverage. It then executes each of the given implementations with each of the test cases in order to calculate a *cross coverage*. Using this it calculates a *combined coverage* and *weighted performance* where implementations, which are not fully covered by the test cases of the other implementations, are punished. These metrics can be used to compare the performance of several implementations on a much deeper level than traditional benchmarks and they incorporate worst, best and average cases in an equal manner. We demonstrate this approach by two example sets of algorithms and outline the next research steps required in this context along with the greatest risks and challenges.

Keywords

performance comparison, algorithm engineering, test case generation, performance tests

1. INTRODUCTION AND RELATED WORK

Developing new algorithms and data structures often starts with the analysis of the already existing approaches. A helpful tool for finding drawbacks are well established benchmarks [6] for the particular problem domain. They are also used to compare the performance of the new to the performance of the existing approaches. Either type of benchmark has two major drawbacks when used for the comparison of new to previous implementations. On the one hand, an implementation may be biased to provide good results for a specific benchmark. On the other hand, benchmarks are designed to evaluate the impact of specific data sets and configurations and not for a specific implementation. They therefore do not necessarily expose the worst and best cases of an implementation.

We propose a solution to these problems by comparing implementations via performance tests which are dynamically created and can not be biased as easily as common benchmarks. Additionally our performance tests cover all aspects of an implementation in equal measure and not only those parts which where foreseen by the creators of a benchmark. [8] propose a model driven automated benchmark creator which relies on a UML 2.0 testing profile and creates benchmarks from the architecture and basic design of the software. Contrary to that, we create test sets on basis of the actual implementation and not the specification. This way, we create test sets which do not miss any part of the code and therefore our tests do not miss certain best and worst case scenarios. [2] propose WISE, a tool to automatically generate inputs which cause worst case performance using symbolic execution. Since they use the actual implementation as a basis, WISE seems to be a good start. However, it searches for a certain behavior instead of representing the overall performance conduct with one test set, which is our goal. [7] criticize the lack of coverage in performance testing and question significance if the coverage is ignored. By generating test sets with the aim of a maximized coverage, we also face this problem.

Beside the generation of test cases which cover the complete implementation, we develop a metric to compare implementations of the same algorithm or implementations of different algorithms which solve the same problem. Such a metric is useful to decide if two implementations are comparable at all. In some cases, several implementations may be used to solve the same problem but focus on different aspects of this problem. In contrast to focusing on one of the aspects and therefore specializing the comparison, we want to compare every speciality of the implementations automatically. Therefore, we need a metric which determines how different the implementations are. To our knowledge there is no

ICPE'16, March 12 - 18, 2016, Delft, Netherlands

ISBN 978-1-4503-4080-9/16/03…$15.00

DOI: http://dx.doi.org/10.1145/2851553.2858663

other metric which examines the comparability of implementations available at present.
In order to provide complete and somehow fair performance tests we combine both aspects, the test sets with a maximized coverage and the comparability metric. Using this, we can evaluate which implementation performs generally best, independently from certain performance profiles.
We give an overview of our framework and the creation of test case sets along with the explanation of our comparability metric. Using a simple example we explain our comparison formula, which is called *combined coverage* and introduce the *weighted performance* which evaluates an implementation based on the aforementioned comparability and test cases with maximized coverage. As the framework is still work in progress we address the main risks and challenges in chapter 3 and conclude with an outlook in chapter 4.

2. PERFORMANCE TEST COMPARISON

In this chapter, we will describe the generation of test cases and the calculation of the combined coverage and weighted performance.
First, for each implementation a test case set with maximized coverage is generated. Second, each implementation is executed with each of the test case sets as parameters. The coverage and average performance of this executions is measured for each test case set. Third, using the coverage of each test case set given by an implementation, the *combined coverage* for measuring the comparability of one implementation to the others is generated. Fourth and last, using the coverage and performance of each test case set on an implementation, the *weighted performance* is generated.

As an introduction consider the four Java code-snippets as depicted in Figure 1. Each of them is an implementation of the method `public int max(int a, int b)` which returns the maximum value of the two given parameters `a` and `b`. We want to evaluate which implementation performs best in comparison to the others. It is crucial to note that two different implementations of the `max` method can be compared much better, if the test cases generated for comparison cover the whole code of both implementations. Otherwise the test cases miss uncovered sections and no valid conclusion about the overall performance can be derived. The only requirement for our procedure is that every implementation fulfills the same specification and reacts with an identical output on certain inputs. So either the same algorithm has been implemented or algorithms which solve the same problem. The framework does not test the functionality of the given implementations, but measures their comparability and performance.
The framework does not generate test cases which cover all given implementations, since that would cause a great computation overhead every time a new implementation has to be compared to the existing ones. Instead, it generates test cases for each of the given implementations. It can either use a model checking approach using Java Path Finder (JPF) [5, 3] or an evolutionary approach using the multi-objective evolutionary algorithm (MOEA) [1] framework. The first is able to find all simple paths in the control flow and therefore maximal covering test cases but fails for rather complex implementations as it faces a state-explosion and too many possible paths. The latter is not confronted with state explosions and may find an ideal implementation which has a maximal coverage and minimal number of test cases but this depends on the given fitness function and mutation operators. An evolutionary algorithm may get stuck in an local optima and not find a good implementation. As the given examples are rather simple we use the JPF-based approach and get the test cases as depicted in Figure 1 in the second column. Another point for adjustment in our framework is the coverage metric on which the comparison weights are based. Since there are many different code coverage metrics available, the framework only provides a general interface for the implementation of additional metrics together with the major ones like statement, branch and path coverage metrics. The decision for one of the metrics should be made in respect for the intended aspect and how detailed the comparability should be determined. For the given example we simply calculate the basic block coverage not only for each of the implementations but for each implementation and each set of test cases. Each entry c_{ij} in Table 1 contains the coverage of test case set j executed on implementation i.

Table 1: Basic block coverage for each of the test case sets executed on each of the algorithms

	test cases of			
implementation	**A**	**B**	**C**	**D**
A	1.0	2/3	1.0	1.0
B	2/3	1.0	1.0	2/3
C	0.8	0.8	1.0	0.8
D	1.0	2/3	1.0	1.0

The covered paths in the control flow graphs of each implemenation when executed with the test case set generated from solution A are shown in Figure 1 in the third column. The coverage value may not exceed a maximum value and should be calculated in relation to the maximal possible coverage. In our case, the maximum is always 1.0. For some coverage metrics, which may not be fully covered or implementations which contain unreachable, *dead* code, it may be less. The coverage value is therefore adjusted to

$$cov_{ij} = \begin{cases} \frac{c_{ij}}{max_i} & \text{if } c_{ij} < max_i \\ 1 & \text{else} \end{cases} \quad (1)$$

where max_i is the maximal possible coverage on implementation i. For a proper comparison of the implementations the framework needs to combine the coverage values for each implementation. If one implementation only covers the test case set of one other implementation well, this indicates that it may be designed especially for this competitor. Therefore, some very good and very bad coverage values are worse than average coverage values only. The combined coverage should also take into account how many test case sets have been combined. A combination of only two implementations should assign a higher weight to a single coverage value than the combination of 10 values. We propose the combined coverage c_i as

$$c_i = \prod_{j=1}^{n} (cov_{ij})^{\frac{k}{n}} \quad (2)$$

where n is the total number of implementations and k is the parameter to adjust the impact of bad coverage values on the combined coverage. The combined coverages for our example are presented in table 2 in the middle column. They

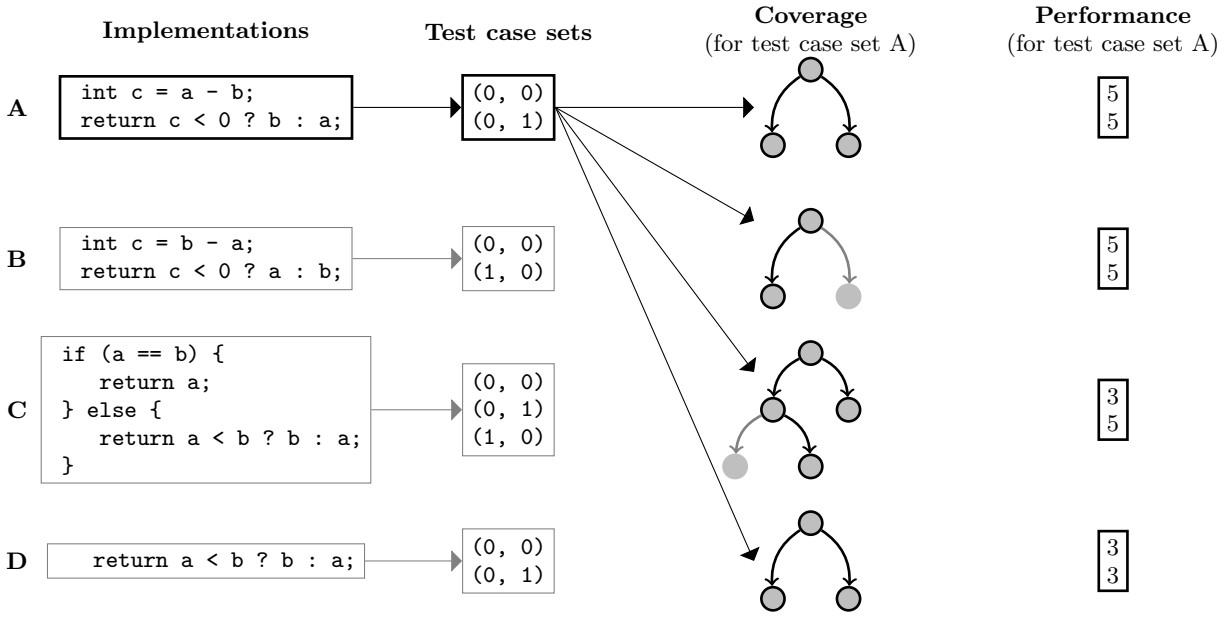

Figure 1: Overview over our framework. Sets of test cases with maximized coverage are generated from each implementation (first to second column). Each implementation is executed with each test case set and the coverage (third column) and performance (fourth column) are measured.

Table 2: Combined coverage and weighted performance with $k = 2$

implementation	combined coverage	weighted performance
A	0,816497	6,562500
B	0,666667	8,125000
C	0,715542	5,770833
D	0,816497	3,937500

can be interpreted as follows: Implementations A and D have the same control flow graph and therefore react equal on test cases. Implementation C has an additional test case, caused by the first statement, the test on the equality of parameters `a` and `b`. Therefore the test cases of C cover all other implementations, but C is not fully covered by the others. In contrast to A, C and D implementation B returns parameter `b` if it equals `a` and therefore has the least combined coverage. That means, that B can not be compared to the other implementations as good as A, C and D.
In the last step, the performance of each implementation is combined with the coverage for each test case set. In our approach a higher performance value is worse and the best possible performance is 1. To maintain a comparability we calculate the weighted performance p_i with the following definition

$$p_i = \left(\sum_{j=1}^{n} \frac{p_{ij}}{(cov_{ij})^k} \right) / n \tag{3}$$

where p_{ij} is the average performance of implementation i when executed with the test cases from implementation j. Similar to the coverage calculation, the way the performance of the implementations is measured, has an impact on the outcome of the evaluation. For our example we use the number of bytecode load and store operations on integers as performance values. The best performing implementation is assumed to be the one with the least number of those instructions. The performance of every implementation when executed with the test cases from A is shown in Figure 1 in the right column. One may consider other performance values like the number of executed cpu cycles, the wall clock time etc., but counting the bytecode operations is sufficient for our explanation. We also disabled Java's just-in-time-compilation which has to be taken into account outside of our simple example.
As for the combined coverage, the coverage values are weighted with the parameter k in order to punish a bad coverage more than an average coverage. The weighted performance of our example is presented in table 2 in the right column. It shows that implementation D is considered to have the best weighted performance. This means, that this implementation not only has a good average performance but also is covered best by the test cases of the other implementations.

3. DISCUSSION

From this first view on our procedure we can identify several uncertainties and open issues which will be discussed in this chapter.
The performance value p_{ij} treats every test case in the same way. It does not weight very unlikely or likely test cases. On the one hand this is not fair, if one implementation has a clearly better average performance but one very unlikely worst case and the other performs worse in average but has no worst case. On the other hand, the aim of our framework is to fully compare several implementations, and not build another benchmark generator. Weighting one test case more than another would also imply knowledge of the intended use and that disagrees our introductive claim to cover all aspects of an implementation equally. Therefore, a full comparison

needs to incorporate all cases in equal measure. Nonetheless, at least the graphical representation of our framework will highlight worst, best, and average case scenarios and identify often used paths and instructions. A prototype of the UI is shown in figure 2.

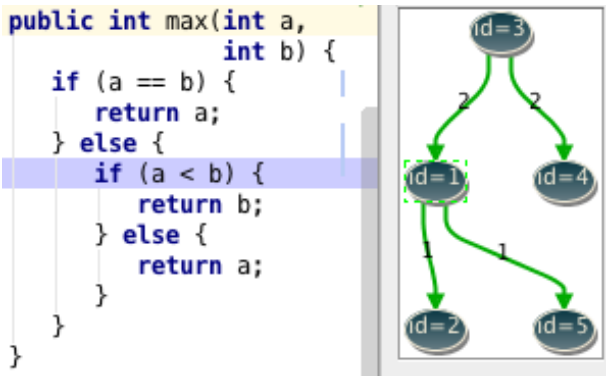

Figure 2: UI - excerpt of our framework. Source code (left) and control nodes (right) are linked to each other. After execution, the visited paths are highlighted and the edges are labeled with the number of accesses.

In the end, the developer who uses our tools, may adjust the weights for the specific purpose after getting a complete overview about the investigated implementations.
The chosen coverage metric is essential and clearly has an impact on the overall computation. For example, we also computed the combined coverage and weighted performance for four sorting algorithms with a quadratic average computation time: BubbleSort, InsertionSort, SelectionSort and ShellSort. When using basic block coverage as metric, each implementation is fully covered by each test case set and therefore also the combined coverage for each implementation is 1.0. We also used path coverage, with the constraint that every loop should be accessed at least twice for full coverage, as metric. Although this changes the combined coverage, it does not change the general outcome for the weighted performance. As the four algorithms are very similar it is only logical that their test cases cover each other and that there is no reasonable coverage metric which would alter the weighted performance such that the performance ranking of the algorithms would change. When confronted with very different implementations, the coverage has a greater impact than for similar implementations.
Not only the way the performance and coverage are calculated, but also the generation of the test cases affects the outcome. In the previous chapter we used JPF to create relatively universal test cases which cover all control flow paths. Changing the test cases of B to `(1, 0) | (0, 1)` already changes the coverage for A and D but not for B and C. This has an impact on the weighted performance and even on the performance ranking. But it is rather obvious that even a slight change on one of two test cases changes the outcome. Problems which require more complex implementations than the simple `max` example in this paper will also produce more test cases per test case set and therefore be less prone to minor changes in the test cases.

4. CONCLUSIONS AND OUTLOOK

In this paper, we present a new approach for comparing the performance of algorithms and data structures on the implementation level. We provide two metrics, the combined coverage and the weighted performance which can be used to compare the coverage and the performance of different implementations. We also discuss our work in progress and state that our calculations may be altered by using different performance weights, coverage metrics and test case generators but none of the alterations let the overall reasoning behind our metrics fail.
In the next steps, we need to apply our framework on a more complex domain. Complex data structures for indexing spatio-temporal data like the R^{ST}-tree [4] are generated on the prior analyzation of their predecessors and therefore are exactly the structures we aim at. We also need to compare our framework to the already existing benchmarks and test sets for those structures or find a way to incorporate them in our performance comparison.
For more complex implementations the usage of JPF is unpractical not only because of a possible state explosion but also because JPF is not able to handle input arrays of variable length. Beside the risk of getting stuck in a local optimum, evolutionary algorithms have to be set up for every problem domain anew. Especially the definition of the fitness function and the mutation operators is crucial for the success of the evolutionary algorithms. Therefore, we need to build a simple and more general framework for the generation of test cases via evolutionary algorithms, which takes one method call as a single gene and a chromosome as one test case. After an extensive test of this evolutionary computation framework, the user will be able to use predefined mutation operators and algorithms which have proven best in previous evaluations.

5. REFERENCES

[1] MOEA Framework. pages 1–210, Jan. 2015.

[2] J. Burnim, S. Juvekar, and K. Sen. WISE: Automated test generation for worst-case complexity. *2009 IEEE 31st International Conference on Software Engineering*, pages 463–473, 2009.

[3] S. Monpratarnchai, S. Fujiwara, A. Katayama, and T. Uehara. Automated testing for Java programs using JPF-based test case generation. *ACM SIGSOFT Software Engineering Notes*, 39(1):1–5, Feb. 2014.

[4] S. Saltenis and C. S. Jensen. R-Tree Based Indexing of General Spatio-Temporal Data. 1999.

[5] W. Visser, C. S. Păsăreanu, S. Khurshid, and S. Khurshid. Test input generation with java PathFinder. *ACM SIGSOFT Software Engineering Notes*, 29(4):97–107, July 2004.

[6] J. von Kistowski, J. A. Arnold, K. Huppler, K.-D. Lange, J. L. Henning, and P. Cao. How to Build a Benchmark. In *the 6th ACM/SPEC International Conference*, pages 333–336, New York, New York, USA, 2015. ACM Press.

[7] M. Woodside, G. Franks, and D. C. Petriu. *The Future of Software Performance Engineering*. IEEE, 2007.

[8] L. Zhu, N. B. Bui, Y. Liu, and I. Gorton. MDABench: Customized benchmark generation using MDA. *Journal of Systems and Software*, 80(2):265–282, Feb. 2007.