Usability of AspectJ from the Performance Perspective

Erik Šuta, Ivan Martoš, and Valentino Vranić

Institute of Informatics and Software Engineering
Faculty of Informatics and Information Technologies
Slovak University of Technology in Bratislava, Slovakia

suta.erik@gmail.com, martos.ivan@gmail.com, vranic@stuba.sk

***Abstract*—While performance is one of the most important attributes of computation intensive systems such as complex event processing, it is essential to success of mobile devices and embedded systems, in which providing seamless experience to end users is of uttermost importance. In this paper, we present a framework we designed to assess AspectJ performance both in desktop and mobile settings. We applied this framework to measure the performance of the current AspectJ version and to provide a comparison with its older versions. One of the important findings is that while in desktop settings vast aspect application does not generate significantly bigger performance overhead than their scarce application, in mobile devices it does, so it pays off to apply aspects rather to a small number of high time complexity methods than to a large number of low time complexity methods.**

***Keywords*—aspect-oriented programming; AspectJ; performance; Android; mobile applications**

I. Introduction

Aspect-oriented programming makes possible to separate concerns that otherwise would be entangled. Although it can be used right from the start of the software development process, it is more popular as a way of introducing changes into existing applications without having to change the existing code [1, 2]. However, the benefits of using aspect-oriented programming and AspectJ as a reference aspect-oriented language [3] have always been overshadowed by impaired performance.

While performance is one of the most important attributes of computation intensive systems such as complex event processing [4, 5], it is essential to success of mobile devices and embedded systems, in which providing seamless experience to end users is of uttermost importance. Applications written with performance in mind are also eco-friendly since they utilize CPU cycles better, thus reducing energy consumption.

In this paper, we propose a framework we designed to assess AspectJ performance both in desktop and mobile settings. We applied this framework to measure the performance of the current AspectJ version and to provide a comparison with its older versions.

The rest of the paper is structured as follows. Section II presents an overview of selected approaches to measuring AspectJ performance. Section III presents our measuring framework. In Section IV the performance measurement results are provided and discussed. Section V concludes the paper.

II. Selected Approaches to Measuring AspectJ Performance

The performance of aspect-oriented programming implementation is one of the most important indicators of approach maturity and readiness for production usage. AspectJ and its performance has been discussed in hundreds of papers. However, on a closer look, it is quite surprising that most of them lack concrete figures. Put in other words, the ratio of the papers just discussing performance and papers providing actual performance measurements and results is very small. The creators of *AspectWerkz*[1] framework have conducted one such measurement in December 2004 [6]. They focused on the comparison of existing approaches to aspect-oriented programming with the goal to identify advice or interceptor overhead. While in AspectJ the overhead ranged from 10 to 50 ns (with exception of the after throwing advice with the overhead of 3009 ns), Spring AOP exhibited the overhead from 275 to 445 ns.

Another interesting approach to measuring AspectJ performance overhead was introduced by Dufour et al. [7]. The proposed solution consists of introducing new metrics for capturing dynamic behavior of AspectJ applications. They also provided tools to capture and evaluate these metrics:

- A modified version of AspectJ compiler that was able to tag bytecode instructions and determine the cause of their generation (e.g., if the instruction was generated by an aspect intrusion or not)
- A modified version of the *J dynamic metrics collection tool composed of a JVMPI-based[2] generator and analyzer that propagates the tags and computes new proposed metrics

In addition to the above mentioned contributions, their work also contained a set of benchmarks to evaluate the performance of the AspectJ framework.

Avgustinov et al. [8] focused on optimizing a code generation strategy to increase the overall AspectJ performance. They addressed several issues. One of them was the compilation of the `around` advice, which is a very challenging task. Avgustinov et al. proposed a new compilation strategy that avoids previously used approaches, such as the usage of excessive inlining and closures. As shown in the benchmark results provided in the paper, the proposed approach leads to performance improvements. It was later accepted by AspectJ developers and integrated into the ajc compiler (version 1.2.1). Another issue was the optimization of the `cflow` pointcut. As stated by Avgustinov et al., previously used techniques were costly both in terms of space and time, so they introduced new techniques to minimize the overhead and improve the `cflow` pointcut performance. Also, Avgustinov et al. addressed the issue of structuring and optimizing compiler so that traditional analyses can be easily adapted to the aspect-oriented programming setting..

Of course, a great optimization effort has been conducted by the developers of the AspectJ language themselves. Many minor releases of AspectJ were actually aimed at performance improvements and introduced no new features. A particularly interesting area of optimization is the load time weaving. In

[1] http://aspectwerkz.codehaus.org/

[2] JVMPI – Java Virtual Machine Profiling Interface

AspectJ version 1.6.7,[3] load time weaving performance improvements were introduced. The source of improvements was mostly the optimization of include/exclude patterns, several of which have been optimized [9]: the exact name pattern (e.g., com.foo.Bar), trailing suffix (e.g., *Bar), types not in included the default package containing a string, and any type pattern (*). This significantly improved the statup time and heap usage.

III. Performance Measurement Framework

AspectJ is an open-source project maintained by the Eclipse Foundation. AspectJ is actually an extension to the Java programming language as any legal Java program is also a legal AspectJ program.

Due to the very specific nature of AspectJ and aspect-oriented programming in general it is not easy to choose an adequate approach to measure AspectJ performance. We decided to take an AspectJ developer perspective concerned with the exact performance overhead that may be expected from using aspects in order to make a qualified decision whether to apply aspects or not in each particular case.

To provide as exact as possible overhead prediction, we created ten tests that are focused on different aspects of Java:

- Ackermann function calculation[4] (deep recursion)
- Fibonacci sequence calculation (branching recursion)
- Large matrix computation (matrix operations)
- Nested loop execution (loop handling)
- Random generation of double numbers (random generation)
- Prime numbers calculation (arithmetic operations)
- Vast string concatenation (working with string values)
- Read of a long text file (working with I/O)
- Quicksort algorithm (sorting)
- Object instantiation (memory allocation)

Of course, the performance overhead of using AspectJ depends on the extent of aspect usage. In other words, it seems logical that the bigger the number of methods wrapped by aspects, the greater the performance overhead will be. To answer this question, we measured the overhead in two types of aspect application:

- Coarse-grained application, in which only the operations of high time complexity are being affected by aspects
- Fine-grained application, in which many calls to the operations of low time complexity are being affected by aspects, too

The tests are assembled into an AspectJ performance measurement framework.[5] Our framework targets all three basic advice types: around, before, and after. The performance overhead is also introduced by the code generated and executed in advices themselves. However, this overhead is not important for the evaluation of general performance cost of aspect usage. Thus, the advices used in our framework are empty.

IV. Measurement Results

The tests introduced in the previous section were conducted a million times to minimize the error rate of measurement methods. The target of measurement is the duration or the time cost of introducing aspects into a Java application. To measure this cost, we used standard measurement methods from the Java API, and the System.nanoTime() method in particular. The problem with using this method can be best expressed by quoting the documentation:[6] "This method provides nanosecond precision, but not necessarily nanosecond resolution (that is, how frequently the value changes)—no guarantees are made except that the resolution is at least as good as that of currentTimeMillis()." This problem can be eliminated by conducting a big number of tests. Of course, we cannot eliminate these inconsistencies completely, so our results inevitably embrace some inaccuracy. All the results introduced here were normalized.

A. Desktop Setting

The first tests were conducted on AspectJ 1.8.4. We focused on determining the cost of the before, after, and around advice. The data can be seen in Fig. 1. It can be clearly seen that the around advice is the one that adds the most performance overhead to the target application. This trend can be seen almost in all tests from our framework. Another observation is that the before advice adds slightly less performance overhead than the after advice.

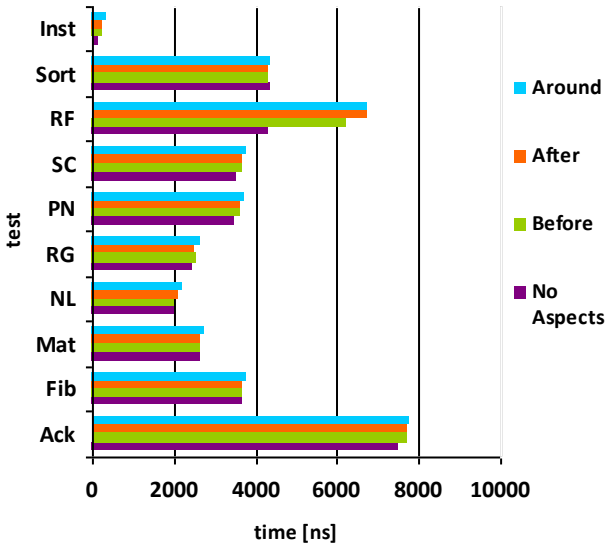

Fig. 1. The normalized measurement results for the after, before and around advice, as well as for code with no aspects.

[3] http://eclipse.org/aspectj/doc/released/README-167.html

[4] http://www.encyclopediaofmath.org/index.php/Ackermann_function

[5] available at https://github.com/eriksuta/AspectJ-Performance-Measurement-Framework

[6] http://docs.oracle.com/javase/7/docs/api/java/lang/System.html#nanoTime()

In the next phase of testing, we have focused on the overall performance improvements of AspectJ during its history. For these purposes, we have chosen the around advice, because since its usage adds the most significant performance overhead, the difference among AspectJ versions would be more visible. We tested AspectJ versions 1.5.0, 1.6.0, 1.6.7, 1.6.13, 1.7.4, and 1.8.4. As can be seen in Fig. 2, over time the AspectJ performance overhead had a decreasing tendency. Despite our great effort to minimize the error in measurements, we can clearly see that the results exhibit some inaccuracy, such as a newer versions performing slightly worse than an older version or a no aspects case performing worse than one with aspects.

We made a set of measurements aimed at differentiating the overhead generated by coarse-grained and fine-grained aspect application. The results can be seen in Fig. 3. This test provided probably the most surprising results. We expected fine-grained aspect application tests to generate much bigger overhead than the coarse-grained ones. These expectations were not met and as we can clearly see the overhead, while it is clearly present, is in general not that significant even in fine-grained tests.

B. *Android Setting*

We repeated the measurements on an Android mobile device with the ART virtual machine (a Java virtual machine type) with a clean Android installation (5.0.1). The original test suite had to be slightly altered for performance reasons. For example, we had to omit the test with the reading of a long text file. Again, measurement results were normalized. As can be seen in Fig. 4, while a rich use of aspects in code on desktop devices (devices with much higher computational capacity and resources) did not cause significant performance overhead, the situation is different on mobile devices. Especially in tests with recursion (*Ack*, *Fib*) or tests with vast number of method calls (*nested loop*, *instantiation*) the rich use of aspects can cause dramatic performance overhead.

V. Related Work

As has been pointed out in Section II, there is a lack of research providing concrete figures regarding performance overhead generated by AspectJ usage. One of early measurements was conducted by the creators of the AspectWerkz aspect-oriented framework in 2004 [6]. However, their approach is more focused on the performance comparison of different aspect-oriented frameworks.

Romanoff and Meyer [10] took a different approach. They measured the difference in performance between pure Java and AspectJ solutions to the set of common problems.

Dufour et al. [7] proposed eight AspectJ benchmarks based on their dynamic metrics. The benchmarks focus on parts of AspectJ not covered by our work, such as performance of inter-type declarations.

VI. Conclusions and Further Work

We conducted a series of performance measurements in AspectJ programming language. For this, we designed a framework that can be used to measure AspectJ performance on both desktop and mobile devices. The framework provides ten different tests each of which aims at capturing different aspects of the Java programming language and by this different situation for the use of aspects. We measured the overhead of all three basic advice types. We also addressed coarse-grained and fine-grained aspect application.

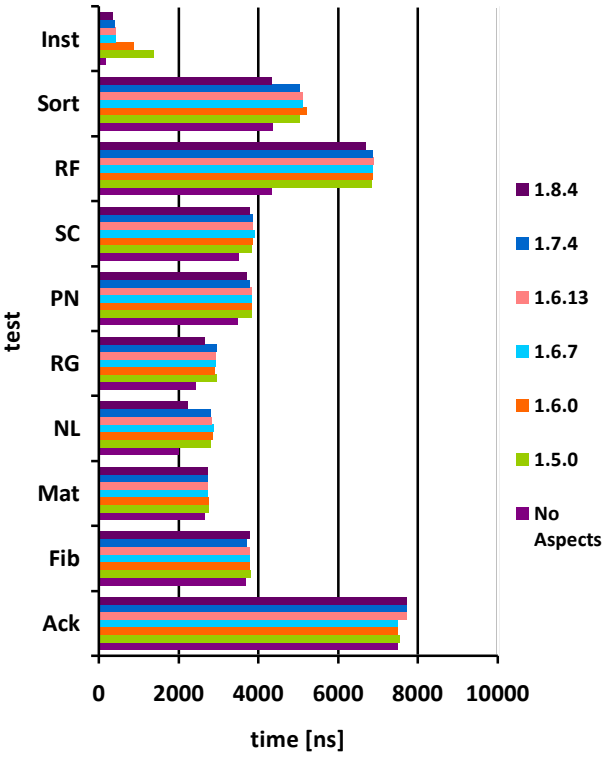

Fig. 2. Overhead in AspectJ over time.

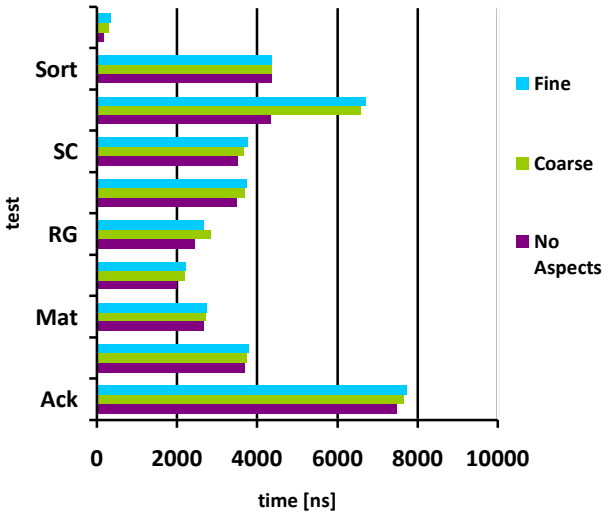

Fig. 3. The difference between fine-grained and coarse-grained aspect application.

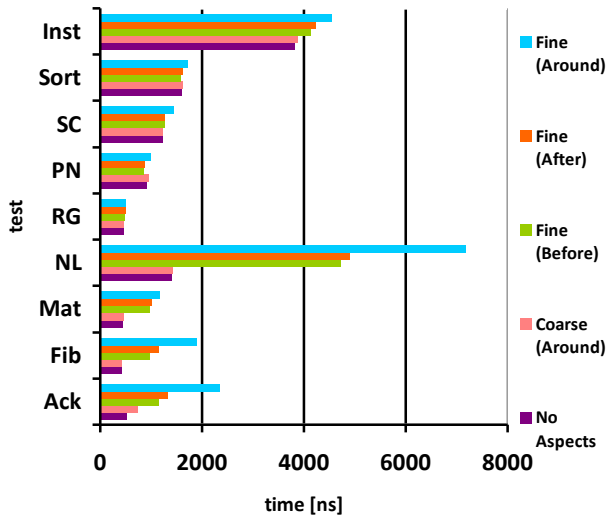

Fig. 4. Measurement results from test run on Android mobile device (AspectJ version 1.7.3).

From the measurement results, the following conclusions can be drawn:

- While in desktop settings vast aspect application does not generate significantly bigger performance overhead than their scarce application, in mobile devices it does, so it pays off to apply aspects rather to a small number of high time complexity methods than to a large number of low time complexity methods.
- The before advice generates less performance overhead than the after advice (on both mobile and desktop devices). As it was expected, the around advice generates the biggest performance overhead.
- In the AspectJ evolution, the developers clearly took performance overhead as a serious topic and invested a lot in its improvement. That can be clearly seen in our measurements with different AspectJ versions.

Our performance measurement framework can be applied to forthcoming AspectJ versions, too, so developers can easily track how is performance affected there. The framework can certainly be improved by addressing further AspectJ features, such as control flow pointcut or after returning and after throwing advice. Another direction of further work might be to embrace performance measurement in profiling efforts, in which aspect-oriented programming has already been applied to some extent [11].

ACKNOWLEDGMENTS

The work reported here was supported by the Scientific Grant Agency of Slovak Republic (VEGA) under the grant No. VG 1/1221/12.

This contribution/publication is also a partial result of the Research & Development Operational Programme for the project Research of Methods for Acquisition, Analysis and Personalized Conveying of Information and Knowledge, ITMS 26240220039, co-funded by the ERDF.

REFERENCES

[1] V. Vranić, R. Menkyna, M. Bebjak, and P. Dolog, "Aspect-Oriented Change Realizations and Their Interaction," e-Informatica Software Engineering Journal, vol. 3, num. 1, 2009, pp. 43–58.

[2] R. Menkyna and V. Vranić, "Aspect-Oriented Change Realization Based on Multi-Paradigm Design with Feature Modeling," in Proceedings of 4th IFIP TC2 Central and East European Conference on Software Engineering Techniques, CEE-SET 2009, Revised Selected Papers, LNCS 7054, 2009, Krakow, Poland, Springer, 2012.

[3] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," in Proceedings of 11th European Conference on Object-Oriented Programming, ECOOP'97, LNCS 1241, Springer, 1997, pp. 220–242.

[4] J. Lang, M. Jantošovič, I. Polášek, "Re-Usability in Complex Event Pattern Monitoring," in Proceedings of IEEE 10th Jubilee International Symposium on Aplied Machine Intelligence and Informatics, Herľany, Slovakia, IEEE, 2012, pp. 265–270.

[5] J. Lang, J. Janík, "Reactive Distributed System Modeling Supported by Complex Event Processing," in Proceedings of ECBS-EERC 2013, 3rd Eastern European Regional Conference on the Engineering of Computer Based Systems, Budapest, Hungary, IEEE CS, 2013, pp. 163–164..

[6] A. Vasseur, "AOP Benchmark." http://docs.codehaus.org/display/AW/AOP+Benchmark

[7] B. Dufour et al., "Measuring the Dynamic Behaviour of AspectJ Programs," ACM SIGPLAN Notices, vol. 39, num. 10, 2004.

[8] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble, "Optimising AspectJ," in Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, Chicago, IL, USA, ACM, 2005.

[9] A. Clement, "AspectJ 1.6.7 and Faster Load Time Weaving." http://andrewclement.blogspot.sk/2009/12/aspectj-167-and-faster-load-time.html

[10] E. Romanoff and J. Meyer, "The Performance of AspectJ," March 19, 2010. http://www.cs.rit.edu/~ear7631/aop/AOPWriteUpPDF.pdf

[11] J. Porubän, J. Kollár, and M. Vidiščak, "Aspect-Oriented Program Profiling," in Proceedings of 8th International Conference on Engineering of Modern Electric Systems, Felix Spa-Oradea, Romania, 2005, pp. 112–117.