On Performance of Delegation in Java

Sebastian Götz
Software Technology Group, Dresden University of Technology, Germany
sebastian.goetz@mail.inf.tu-dresden.de

Mario Pukall
Database Research Group, Otto-von-Guericke-University Magdeburg, Germany
mario.pukall@iti.cs.uni-magdeburg.de

Abstract

Delegation is an important design concept in object-oriented languages. In contrast to inheritance, which relates classes to each other, delegation operates on the level of objects and thus provides more flexibility. It is well known, that usage of delegation imposes performance penalties in the sense of delayed execution. But delegation on the level of sourcecode is optimized on bytecode level up to a degree, that imposes much lower performance penalties than expected. This paper examines in detail how big these penalties are.

Categories and Subject Descriptors D.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement—Dynamic Software Upgrade

General Terms Java, HotSpot, Inlining, Software Upgrade

Keywords Upgrade, Java, JVM, Delegation, Inlining

1. Introduction

Many approaches in the context of object-orientation base on the concept of delegation. For example, Pukall et. al.'s object-wrapping approach [6] enabling dynamic runtime upgrades of Java applications. Objects of updated classes are wrapped. Every further update introduces another wrapper, leading to deep delegation chains. In [3] Kniesel presents an object wrapping approach based on type-safe delegation, which also exhaustively utilizes delegation. Büchi and Weck [1] introduced *Generic Wrappers* for late composition of component systems, which enable generic aggregation of objects at runtime. Again delegation is keenly utilized. In [2] Şavga et al. present ComeBack!, an approach supporting clients to run with newer frameworks, than they were originally written for. The key idea is to put binary-compatible adapters between clients and frameworks, thus introducing further levels of delegation. PROSE[5] is a system supporting runtime method body changes. Changes are realized as separate Java classes to enable composition of changes, leading to delegation chains of considerable depth.

It is well known, that heavy usage of delegation imposes valuable performance penalties in the sense of delayed execution. But how big these penalties actually are, is unknown. Nevertheless, these performance penalties are often seen as drawbacks of approaches keenly utilizing delegation. As we will show in Section 3, source code is optimized on bytecode level and delegation imposes much lower performance penalties than expected, thus puts these drawbacks into perspective.

This paper presents empirical data about performance penalties, introduced due to delegation. Six of the most common Java Virtual Machines (Suns and Apples HotSpot 5 and 6 and Oracles JRockit Mission Control 5 and 6) were examined on three different operating systems (Windows 7, Linux and Mac OS X).

2. Experimental Setup

To measure the performance penalties implied by introducing levels of indirection, a set of classes, whose instances are connected to each other via method calls, needs to be created. That is, each class has at least one method, which invokes a method of another class. All methods connected to each other by invocation form a method chain.

To simulate real world methods a complex calculation is performed in each method. Four different numbers (a to d) between 0 and 100, are randomly generated for each method containing an invocation and are used in the following formula: $sin(a) * tan(b) + \sqrt{c^2 + d^2}$.

The randomly generated classes differ in their name, number and names of their methods, as well as number and type of arguments per method. This circumvents, that the just-in-time (JIT) compilers optimizer is able to see the coherency between the methods, i.e. the method chain, simply by looking at their names. The generated classes have the form depicted in Listing 1.

HotSWUp'09, October 25, 2009, Orlando, Florida, USA.
Copyright © 2009 ACM ISBN 978-1-60558-723-3/09/10…$5.00

```
class $C_i$ {
    private $C_{i+1}$ next;
    public String $call_i^1(t_1^1\ p_1^1, \ldots, t_j^1\ p_j^1)$ {
        if(next != null) {
            double x = Math.sin(a) * ...
            return next.$call_{i+1}^y(v_1^y, \ldots, v_k^y)$;
        } else return ”end”;
    }
    ...
    public String $call_i^k(t_1^k\ p_1^k, \ldots, t_q^k\ p_q^k)$ { ... }
    ...
}
```

Listing 1. General Class Format. Each Class Comprises k Methods. They Differ in Their Name and Number/Type of Arguments.

In order to measure the time an execution of the first method in the chain takes, a client is used, which instantiates the classes, connects all instances to each other, and calls the first method in the delegation chain. The method call is surrounded by calls of `System.nanoTime()`, which return the current system time in nanoseconds.

The method is executed several times. This is, because the JIT compiler needs a warm-up phase to optimize the running code. The more often the code is executed, the closer its execution time comes to execution times of code, which does not use delegation at all.

The class and client generation is parameterized by the number of classes to be generated, the maximum number of methods per class, parameters per method, and the number of execution times. In the following this test is called `TestJIT`.

Due to limitations of class files[4], the maximum number of classes is limited. This is, because the byte code of any method must not exceed the size of 65534 kilobyte. The client needs to create instances of all classes and connect them to each other. The maximum number of classes revealed to be approximately 2000.

In order to make a statement about the performance penalty of delegation, reference values are necessary. Hence the question is, how long would the same execution take, if delegation is not used? In consequence a single method, containing all calculations, which originally were spread over all methods in the chain, needs to be executed and the time this execution takes has to be measured. As for the measurement of delegation the method should be executed several times, so the JIT compiler is able to identify the hotspots. To get comparable total times the same numbers *a* to *d* as used in the generated classes are used in this method. This class is furtheron called `TestManual`.

3. Empirical Data

We run the clients described in Section 2 on different combinations of machines and Java Virtual Machines (JVM). The JVMs we used are Suns HotSpot, Oracles JRockit Mission Control 3.1.0 and Apples HotSpot. All VMs were chosen, because they are well-known and often used in productive environments. The HotSpot VM is the standard VM, which is delivered by Sun. Hence it is optimized for general purpose usage. Its name typifies its special feature. The HotSpot VM continuously analyzes the byte code to detect so-called ”hot spots”, statements which are frequently or repeatedly executed. As the VM compiles byte code to native code just in time, i.e. at runtime, it is able to optimize these hot spots based on runtime information. Oracles JRockit JVM allegedly has a better performance than Suns HotSpot.[1] Unfortunately Oracle does not provide information about how they achieve better performance. Apples HotSpot is similar to Suns implementation, but optimized for a single processor architecture.

All of these JVMs have been tested for Java Version 5 and 6. Always the latest release of the corresponding JVM has been used. The reason for testing on different JVMs is the assumption, that they will optimize the code in different ways. Version 1.4.2 of the JVMs has been omitted, because it does not provide methods to retrieve the current time at the granularity of nanoseconds, but just milliseconds.

Different operating systems and different editions of the JVMs have been used. The tests have been run multiple times, revealing a negligible deviation between the results of the same tests. Thus the results are reproducible. The machines used for the tests are:

- Windows 7 and Linux (2.6.28) on Intel Core 2 Duo T7700, 2.4GHz, 3GB RAM
- Mac OS X on MacBook Pro, Intel Core 2 Duo, 2.66GHz, 4GB RAM

3.1 Results for Windows 7

Figure 1 shows the results of `TestJIT` and `TestManual` for the Sun HotSpot 6 VM on the Windows 7 machine for a set of 1000 classes, having a maximum of 10 methods per class and a maximum of 10 attributes per method. The x-axis shows the number of execution times and ends at 10.000. The y-axis denotes the time a single execution took in milliseconds. The lightgray line depicts the times achieved by `TestManual`. Except for some outbreaks the time is almost all time around 2,5 milliseconds. Thus, by executing the same method over and over again, the JIT compiler is not able to optimize. The darkgray line depicts the times of `TestJIT`. Here 1000 instances of different classes are connected to each other, as described in Section 2. During the first 2000 executions the necessary time for a single execution varies around 7,5 milliseconds, but there are multiple

[1] http://www.oracle.com/technology/products/jrockit/

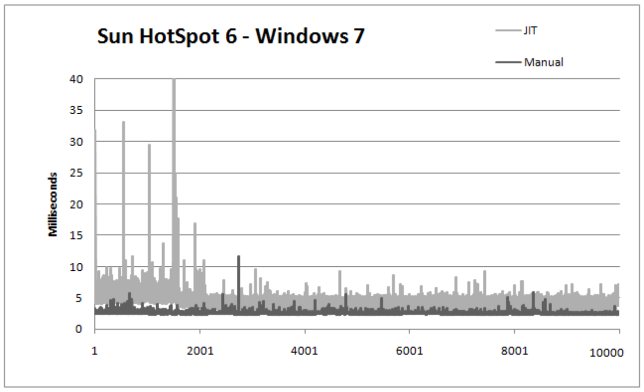

Figure 1. Total Execution Time by Number of Execution Times for Sun HotSpot 6 under Windows 7

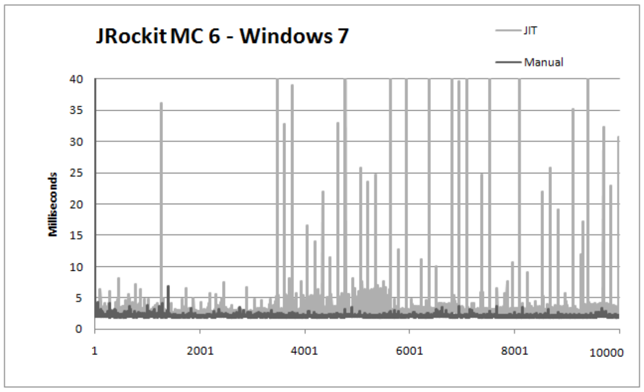

Figure 3. Total Execution Time by Number of Execution Times for Oracle JRockit MC 6 under Windows 7.

outbreaks taking around 32 milliseconds. After 2000 executions the necessary time for a single execution varies between 2,5 and 5 milliseconds, having an average of 3,5 milliseconds. In comparison to the time needed to execute the single method in `TestManual`, using the running average values over all execution times, delegation imposes a performance penalty of 50%.

The reason for the speedup is an optimization technique of the JIT compiler: inlining. If one method often calls another method, the content of the other method is inlined, i.e. copied, into the first method, thereby avoiding an expensive virtual call. In `TestManual` this has been done manually. In `TestJIT`, after approximately 2000 times of execution, all methods, which are chained together, have been called at least once. Thus, the JIT compiler knows, about all of them and is able to inline them. Therefore, the optimized code is able to achieve execution times even of 2,5 milliseconds, just like the single method. Thus modularization of source code, i.e. using delegation and inheritance, does not contradict fast execution.

Figure 2 shows the results for the same test on a Sun HotSpot 5 VM. In comparison to HotSpot 6 the times for the single method are approximately the same. The times needed for the chained methods have more breakouts, but in general are, too, approximately the same. The average time for the single method is slightly better at 2,3 milliseconds. The average time for the method chain degraded to 3,1 milliseconds. Suns HotSpot 5 VM showed a performance penalty for delegation of 46%.

Figures 3 and 4 show the same test run with Oracles JRockit for Java 6 and 5. The results for the single method are again stable, but slightly better at an average of 2 milliseconds. This holds for both, Java 6 and 5. Interestingly the first execution took more than 100 milliseconds. The results for the method chain differ from those of Suns HotSpot VMs in that much more breakouts occur, even after 2000 executions. The first execution even took almost 500 milliseconds. The y-axis is limited to 40 milliseconds for clarity. The many breakouts show, that Oracles JIT optimizer works in another way and gives the impression, that it performes worse than Suns optimizer. But this is not the case, as a look on the average time for an execution of the method chain shows. In average such an execution in Java 6 takes only 2,7 millseconds. In Java 5 it takes 2,8 milliseconds. The performance

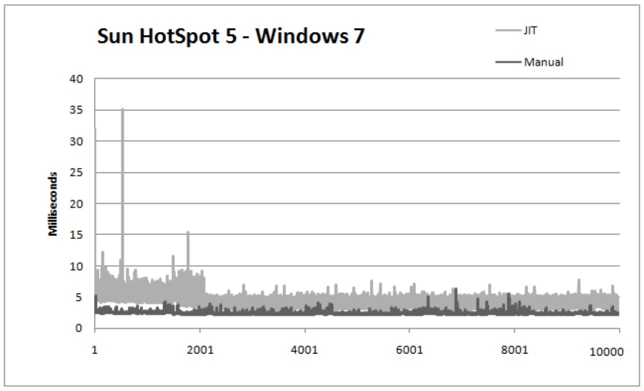

Figure 2. Total Execution Time by Number of Execution Times for Sun HotSpot 5 under Windows 7

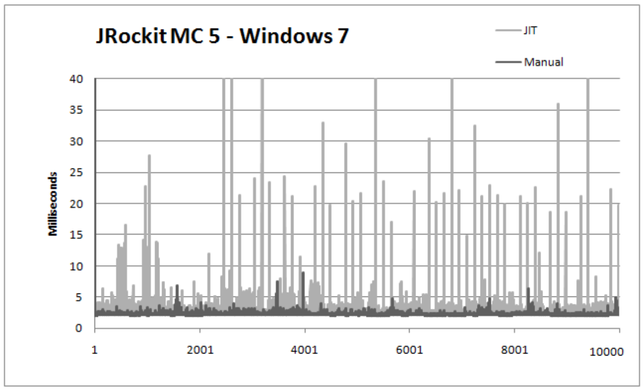

Figure 4. Total Execution Time by Number of Execution Times for Oracle JRockit MC 5 under Windows 7.

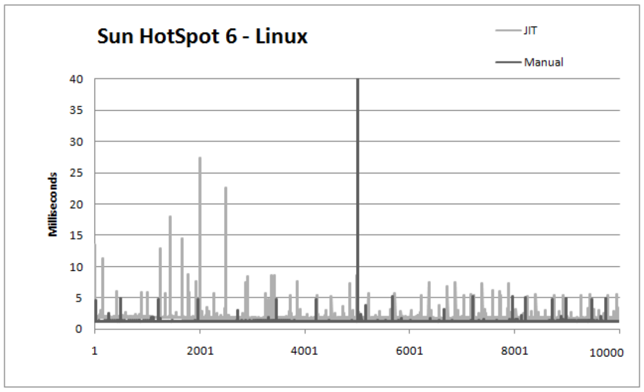

Figure 5. Total Execution Time by Number of Execution Times for Sun HotSpot 6 under Linux.

penalty introduced by delegation is thus 43% for Java 6 and 39% for Java 5.

3.2 Results for Linux

We run the test on the same machine, but under Linux (2.6.28). The results for Suns HotSpot 6 und 5, depicted in Figures 5 and 6, show, that the JIT compilers optimizer works in a slightly different way compared to its Windows 7 implementation. Althouh this VM is the HotSpot VM developed by Sun, too. But it is a separate port of it, which has its own characteristics. In version 6, after the first few executions, the time needed for a single execution goes down to approximately 3 milliseconds. Under Windows this only rarely happened before the first 2000 executions. But under Linux much more outbreaks occur, though they are relatively small. Also version 5 of Suns HotSpot slightly differs from its Windows counterpart, in that it too has much more outbreaks and no change occurs after 2000 executions. The performance penalties introduced by delegation are less than under Windows. Suns HotSpot 6 needs an average of 1,8 milliseconds for the method chain and an average of 1,3 milliseconds to execute the single method. This results in a penalty of 37%. Version 5 of Suns HotSpot needs an average of 1,33 milliseconds for the method chain and 1,4 milliseconds for the single method, thus imposes a performance penalty of 8%.

Comparing Suns HotSpot implementations for Windows and Linux gives the impression, that Linux is the better choice for applications making heavy usage of delegation. The JRockit results show, that this is indeed the case. Figure 7 and 8 show the results for version 5 and 6. They don't differ in their characteristics from the results of their Windows counterpart, but in the penalties imposed by delegation. Version 5 and 6 need an average of 1,2 milliseconds for the method chain, which is more than twice as fast as under Windows. The average time needed to execute the single method is 1,07 milliseconds, which is much faster than under Windows, too. The performance penalty imposed by delegation results in only 14% for Java 5 and 6. This is less than half of the penalty under Windows 7.

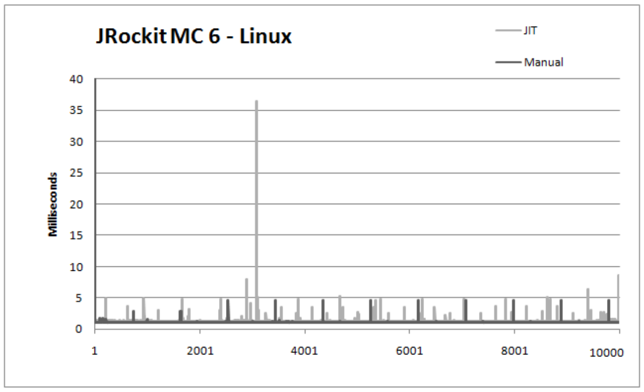

Figure 7. Total Execution Time by Number of Execution Times for Oracle JRockit MC 6 under Linux.

3.3 Results for Mac OS X

Finally we run the same test on Mac OS X. Oracle does not provide a JRockit Mission Control for Mac OS X, thus only Apples Java HotSpot implementations have been tested.

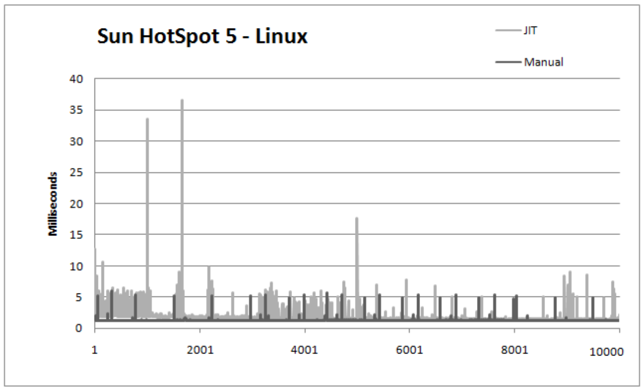

Figure 6. Total Execution Time by Number of Execution Times for Sun HotSpot 5 under Linux.

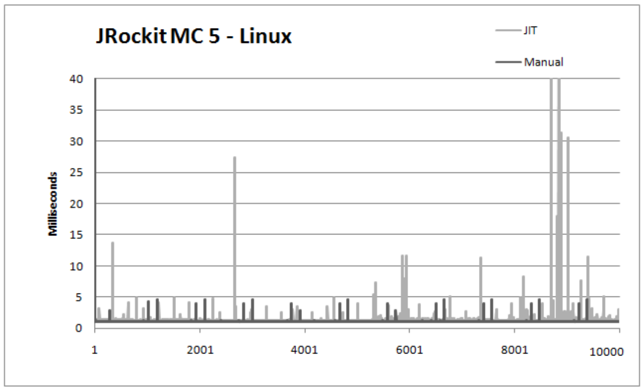

Figure 8. Total Execution Time by Number of Execution Times for Oracle JRockit MC 5 under Linux.

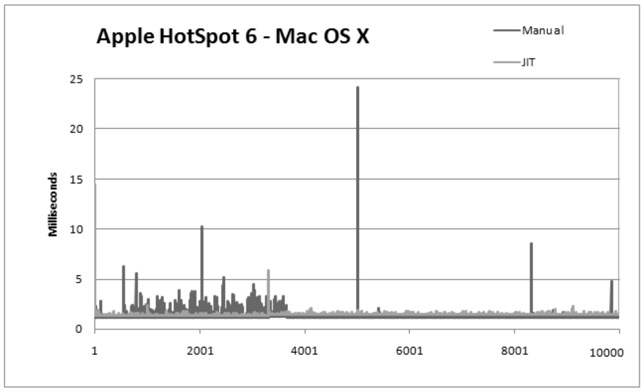

Figure 9. Total Execution Time by Number of Execution Times for Sun HotSpot 6 under Mac OS X.

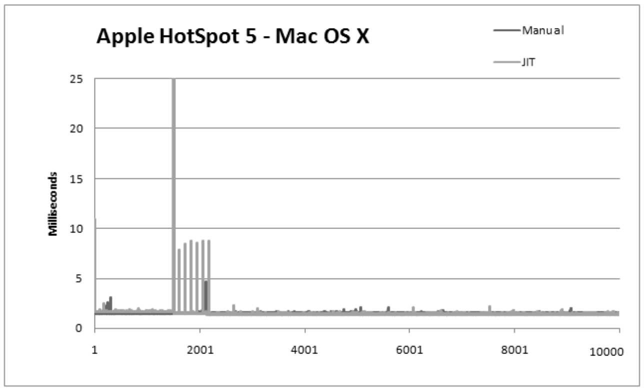

Figure 10. Total Execution Time by Number of Execution Times for Sun HotSpot 5 under Mac OS X. The y-Axis is Limited to 10 Milliseconds. The 1500th Execution had a Breakout of More Then 300 Milliseconds.

Figures 9 and 10 show the results. We used a slightly better equipped machine, thus tests run faster. For the single method Apples HotSpot 5 achieved an average of 1,5 milliseconds, version 6 of Apples HotSpot even 1,25 milliseconds. The figures emphasize, that under Mac OS X the single method sometimes takes longer than the method chain. Apples HotSpot 6 needs an average of 1,44 millseconds for the method chain and 1,25 milliseconds for the single method. Thus the performance penalty introduced by delegation is only 15%. Version 5 of Apples HotSpot is even better. Both, the single method and the method chain need an average of 1,5 milliseconds, thus in this case **no performance penalty** at all is implied by delegation. An important difference between Apples and Suns HotSpot, as well as Oracles JRockit is, that Apple was able to optimize their VM for a single processor architecture, whereas the other VMs need to support a variety of them.

3.4 Varying Workload

All results were measured with the same workload, that is the same calculation. This makes the results comparable,

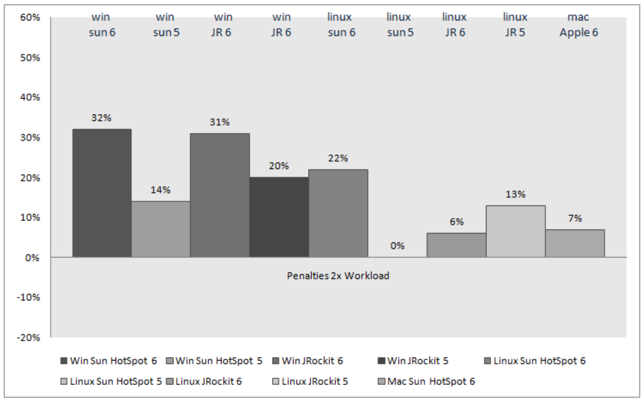

Figure 11. Penalties for Double Workload, Measured Under Windows 7, Linux and Mac OS X

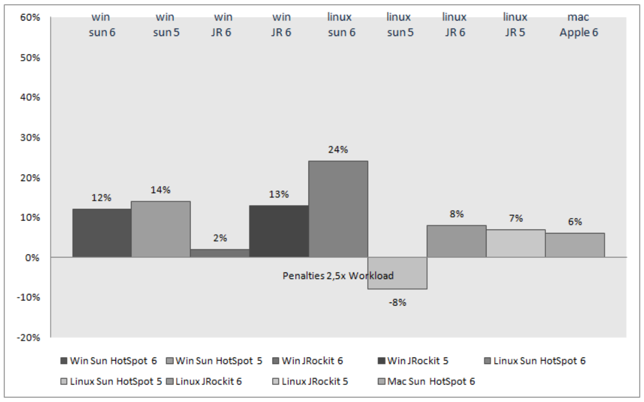

Figure 12. Penalties for 2.5 Times Workload, Measured Under Windows 7, Linux and Mac OS X

but doesn't take the influence of the workload into consideration. Hence, we run the same tests with double and 2.5 times workload. To simulate double workload, the formula was doubled, that is, the same formula is calculated twice, though with different values. For 2.5 times workload, half of the formula is added to the doubled formula. The reason for using 2.5 times workload, instead of triple workload is, that using triple workload, the number of classes is limited to less than 1000 classes.

The results of these tests showed a basic principle: *the more workload, the less penalty*. Figure 11 summarizes the resulting penalties for double workload. The penalties are never worse than 32% and go down to even 0%. For 2.5 times workload, depicted in Figure 12 the penalties are even lower. A maximum of 24%, going down to -8%. That is, the JIT optimized delegation code is faster, than the manually inlined code, which is optimized by the JIT, too.

Thus, delegation imposes less performance penalties for computation-intensive methods. Because sometimes delegation is even faster than manually inlined code, big methods should be avoided. Modular, reusable code, using delegation, is able to perform better.

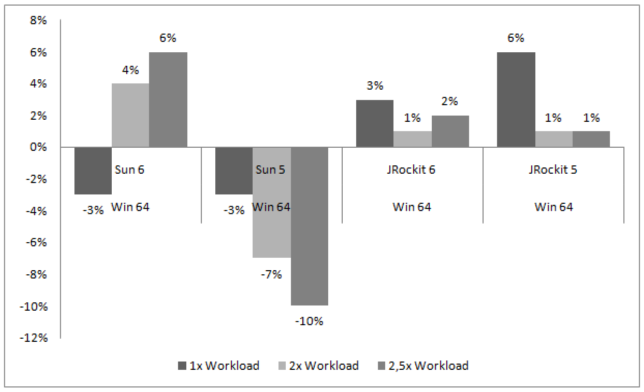

Figure 13. Penalties For Suns HotSpot and Oracles JRockit MC in Version 5 and 6, Measured on a 64bit Windows 7.

3.5 32bit vs. 64bit

An important difference between the two machines used for testing is, that the first machine, running Windows 7 and Linux, works with 32bit, but the second with 64bit. This could be an important difference, as different ports of the JIT compilers for 64bit, instead of 32bit, exist. Hence we run the same tests on a comparable 64bit machine for Windows 7 and Linux using the appropriate VM implementations.

The results show, that JVM implementations for 64bit perform much better, than those for 32bit. Figure 13 depicts the results for Windows 7 on a 64bit machine. Suns HotSpot VM for Java 5 is most performant. It starts, like Suns HotSpot for Java 6, with -3% for usual workload. Thus, even for small workloads the JIT compiler is able to optimize delegation code in a way, that it runs faster, than manually inlined code. For 2.5 times workload the penalty is even -10%, confirming the statement *"the more workload, the less penalty"*. A look at Oracles JRockit Mission Control shows very small penalties, too. They never exceed 6% and, for double and 2.5times workload go down to just 1%.

In comparison the 32bit results under Windows 7 are between 39% and 50% for usual workloads. The 64bit implementations of the VMs are at least 6 times as fast, as their 32bit counterparts.

4. Conclusion

The paper presented empirical data about performance penalties imposed by delegation. Two different tests were developed as described in Section 2. The first one simulates exhaustive delegation, the second one represents the same code, but without using delegation. The resulting times were the basis to calculate the running average, whose last values were used to calculate the performance penalty imposed by each virtual machine on the corresponding operating system. Figure 14 summarizes the derived penalties.

As has been shown in Subsection 3.4, the more workload the method chain contains, the less performance penalty is imposed by it. Big, unstructured methods do not necessarily perform better, than many modular and reusable methods, utilizing delegation.

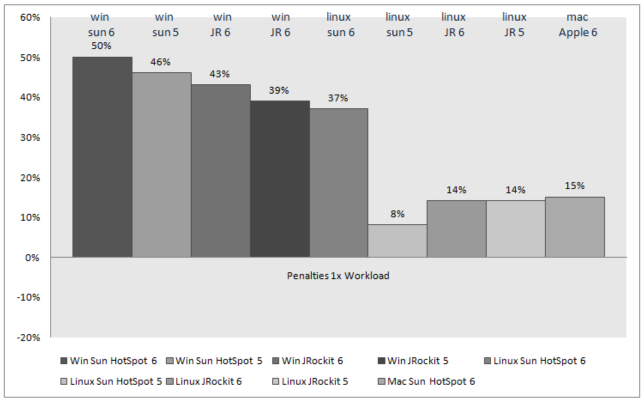

Figure 14. Performance Penalties Inmposed by Delegation in Percent, Relative to Manual Inlining. I.e. 50% means, code using delegation takes 1.5 times longer.

Furtheron 64bit implementations of all VMs used to collect the data, perform much better than their 32bit counterparts. They are at least 6 times faster.

In the worst case delegation imposes a penalty of 50%. The majority of Virtual Machines imposes for usual workload at most 40%. But increasing the workload even leads to penalties, below 0%! Thus, although delegation imposes performance penalties, these penalties stay in reconcilable borders.

References

[1] Martin Büchi and Wolfgang Weck. Generic wrappers. In *Proceedings of ECOOP 2000, LNCS 1850*, pages 201–225. Springer, 2000.

[2] Ilie Şavga, Michael Rudolf, Sebastian Götz, and Uwe Aßmann. Practical refactoring-based framework upgrade. In *Proceedings of GPCE 2008*, pages 171–180, New York, NY, USA, 2008. ACM.

[3] Günter Kniesel. Type-safe delegation for run-time component adaptation. In *Proceedings of ECOOP 1999, LNCS 1628*, pages 351–366. Springer, 1999.

[4] Tim Lindholm and Frank Yellin. *Java^{TM} Virtual Machine Specification, The, 2nd Edition*. Prentice Hall, 1999.

[5] Angela Nicoara, Gustavo Alonso, and Timothy Roscoe. Controlled, systematic, and efficient code replacement for running java programs. In *Proceedings of EuroSys 2008*, pages 233–246, 2008.

[6] Mario Pukall, Christian Kästner, and Gunter Saake. Towards unanticipated runtime adaptation of java applications. In *Proceedings of APSEC 2008*, pages 85–92. IEEE, 2008.