Towards Performance Measurements for the Java Virtual Machine's invokedynamic

Chanwit Kaewkasi
School of Computer Engineering
Suranaree University of Technology
Nakhon Ratchasima, Thailand 30000
chanwit@sut.ac.th

ABSTRACT

This paper presents a study of a Java Virtual Machine prototype from the Da Vinci Machine project, defined by JSR 292. It describes binary translation techniques to prepare benchmarks to run on the `invokedynamic` mode of the prototype, resulting in the `invokedynamic` version of the SciMark 2.0 suite. Benchmark preparation techniques presented in this paper are proven to be useful as the `invokedynamic` version of benchmark programs successfully identified strange slowness behavior of the `invokedynamic` mode of the server virtual machine.

Surprisingly, benchmarking results show that the `invoke-dynamic` mode with direct method handles on the server virtual machine is just 2-5 times slower than native Java invocations, except the Monte Carlo benchmark. But this mode on the client virtual machine still requires further performance tuning.

Categories and Subject Descriptors

C.4 [**Measurement Techniques**]; D.3.3 [**Programming Languages**]: Language Contructs and Features; D.3.4 Optimization, Code generation.

General Terms

Measurement, Performance, Languages

Keywords

Bytecode, invokedynamic, method invocation, benchmarks

1. INTRODUCTION

Java Virtual Machine Language (JVML) is an intermediate language defined by the Java Virtual Machine Specification [9]. JVML has been designed to be statically verifiable and symbolically linkable as its class file format contains type information enough to do so. Classes, methods and fields in JVML are strongly typed. Every symbol is linked before executing a referencing instruction [15]. In the JVM, there are four instructions for invoking different kinds of Java method calls. These instructions serve Java programs perfectly, but not that well for other languages that use the JVM as their runtime systems [4], [5], [8], [11], [12].

The Da Vinci Machine project [3], also known as the Multi-Language Virtual Machine (MLVM), is an effort to investigate and develop prototypes that support a new bytecode instruction, `invokedynamic` [15], to extend the Java Virtual Machine (JVM) to other languages, beside the Java language itself. Recently, several language implementers have adapted the JVM as a runtime system for their languages, which include JavaScript [11], JRuby [12], Scala [13], Groovy [8], Clojure [4] and Jython [5]. Dynamically typed languages among them have been trying to simulate dynamic invocation using the existing infrastructure provided by the JVM, and resulting in several magnitudes of performance degradation (see Chapter 7 in [6]). After years of drafting the Java Specification Request (JSR) 292 by its Expert Group, the referential implementation of `invokedynamic` has now been a part of the early access builds of Java Development Kit (JDK) and OpenJDK 7, while the Da Vinci Machine project has continued to implement several advanced features separately.

Reported by members in the development mailing list of MLVM, the performance of dynamic invocation in recent builds of OpenJDK has been around 10 times faster than invocation through Java's reflection APIs [1]. Thus, it is worth measuring the performance of the `invokedynamic` mode of the JVM, even in the early stage of its development, to help identifying possible bottlenecks in both JDK and OpenJDK implementations.

Following one of the design principles of JSR 292 that focuses on performance characteristics of the new bytecode [15], this paper aims at finding translation techniques for replacing all traditional Java invocations by equivalent `invokedynamic` instructions in available benchmark programs in order to measure performance of this new execution mode. This paper chose the SciMark 2.0 benchmark suite [14] as input programs for the translation process because of 1) its simplicity, 2) all benchmarks are CPU-bound, and 3) these benchmarks mainly contain primitive operations, thus refactoring them as method calls and replacing them by `invokedynamic` can represent how well MLVM performs execution and optimization for trivial final methods.

VMIL'10, 17-OCT-2010, Reno, USA

One of the problems this paper addresses is that benchmarks for a new runtime, like the `invokedynamic` mode, have not been widely implemented at this early stage, as the technology is relatively new and its adoption is just started. Another problem is that `invokedynamic`'s target users are language implementers. Implementing benchmarks in various languages is difficult to directly compare the performance of `invokedynamic` with native Java invocations. Other factors in these languages, such as target method selection algorithms, may affect the measurement.

The main contribution of this paper is to present the results of performance measurements for the JVM's `invokedynamic` using the refactored version of the SciMark 2.0 benchmark suite [14]. The hypothesis is that the performance of `invokedynamic` should not be too slow compared with native Java invocations. Other contributions are 1) a binary translator for compiling normal Java programs into equivalent `invokedynamic` programs, 2) an identification of a potential bottleneck in the server (C2) virtual machine (VM), and 3) a limitation of the bytecode verifier when taking `invokedynamic` into account.

This paper is organized as follows. Section 2 reviews the `invokedynamic` mode and its components. Section 3 describes a mapping from other invocation instructions to `invokedynamic`. Section 4 discusses benchmarks preparation, experiments and results. Section 5 continues on discussion about important implementation issues, namely, the use of `findSpecial` and bytecode verification when taking `invokedynamic` into account. Section 6 discusses related work, and then this paper ends with conclusion and possible future work in Section 7.

2. REVIEW `invokedynamic`

This section reviews the `invokedynamic` mode of the JVM. It consists of a newly defined bytecode, `invokedynamic`, and other key components, such as the bootstrap method, and method handles. However, adapter method handles are not discussed here as they are beyond the scope of this paper.

2.1 Bytecode

An `invokedynamic` is a 5-byte instruction. It has no scope type, because it is designed to be a replacement for all other invocation instructions. It symbolic reference is called name-and-type, where one can specify the name of the call site as well as parameter and return types. When referring to a dynamic call site, it is an instance of an `invokedynamic` instruction in a method body [15].

This instruction is linked for the first time when the call site is executed. The JVM will obtain a call site object for the current call site by calling the local bootstrap method.

2.2 Bootstrap Method

A bootstrap method is responsible to create a call site object, of type `java.dyn.CallSite`, upon request by the JVM. The JVM passes static information of the executing dynamic call site to this method. A compiler that generates `invokedynamic` instructions is also required to generate this method for each class. For example, a binary translator presented in this paper generates a bootstrap method and registering it to the JVM in the class initializer [15]. Recently, JSR 292 allows specifying a bootstrap method per call site, rather than having only one bootstrap method per class [16].

Before the bootstrap method returning a call site object, the call site's target must be assigned an instance of a method handle.

2.3 Method Handle

A method handle is a lightweight structure that has been designed to invoke a JVM method [15]. There are several kinds of method handles defined by JSR 292. However, this paper uses only the simplest one, which is called the direct method handle, to minimize runtime overheads that may affect the experiments. A direct method handle can be obtained by using an instance of `MethodHandles.Lookup`, which provides finder methods for each kind of JVM calls, for example,

- `findStatic` for static methods,
- `findSpecial` for inherited methods (super calls),
- `findConstructor` for constructors, and
- `findVirtual` for normal and interface calls.

3. TRANSLATION TO `invokedynamic`

From Section 2, finder methods are reviewed to show that different categories of method handles can be obtained from different kinds of them. This section discusses semantics when mapping from traditional invocation instructions to equivalent `invokedynamic` instructions using these finder methods. Semi-formal rules in Figure 1 are used to implement a binary translator that compiles class files into the `invokedynamic` version.

Static Method Call:

$$\frac{h=\text{findStatic}(C,m,D,\text{type}(\overline{e}))}{\text{invokestatic}(C,m,\overline{e}):D\rightarrow\text{invokedynamic}(I,h,\overline{e}):D}$$

Constructor Call:

$$\frac{c:C \quad h=\text{findConstructor}(C,\text{type}(\overline{e}))}{\text{invokespecial}(C,\langle\text{init}\rangle,c\bullet\overline{e}):V\rightarrow\text{invokedynamic}(I,h,\overline{e}):C}$$

Inherited Method Call:

$$\frac{h=\text{findSpecial}(C,m,D,\text{type}(\overline{e}),E)}{\text{invokespecial}(C,m,\overline{e}):D\rightarrow\text{invokedynamic}(I,h,\overline{e}):D}$$

Special `super()` and `this()` Call:

$$\frac{\text{this}:E \quad h=\text{findSpecial}(C,\langle\text{init}\rangle,V,\text{type}(\overline{e}),E)}{\text{invokespecial}(C,\langle\text{init}\rangle,\text{this}\bullet\overline{e}):V\rightarrow\text{invokedynamic}(I,h,\text{this}\bullet\overline{e}):V}$$

Virtual Method Call:

$$\frac{c:C \quad h=\text{findVirtual}(C,m,D,\text{type}(\overline{e}))}{\text{invokevirtual}(C,m,c\bullet\overline{e}):D\rightarrow\text{invokedynamic}(I,h,c\bullet\overline{e}):D}$$

Interface Method Call:

$$\frac{c:C \quad h=\text{findVirtual}(C,m,D,\text{type}(\overline{e}))}{\text{invokeinterface}(C,m,c\bullet\overline{e}):D\rightarrow\text{invokedynamic}(I,h,c\bullet\overline{e}):D}$$

Figure 1. Translation Rules.

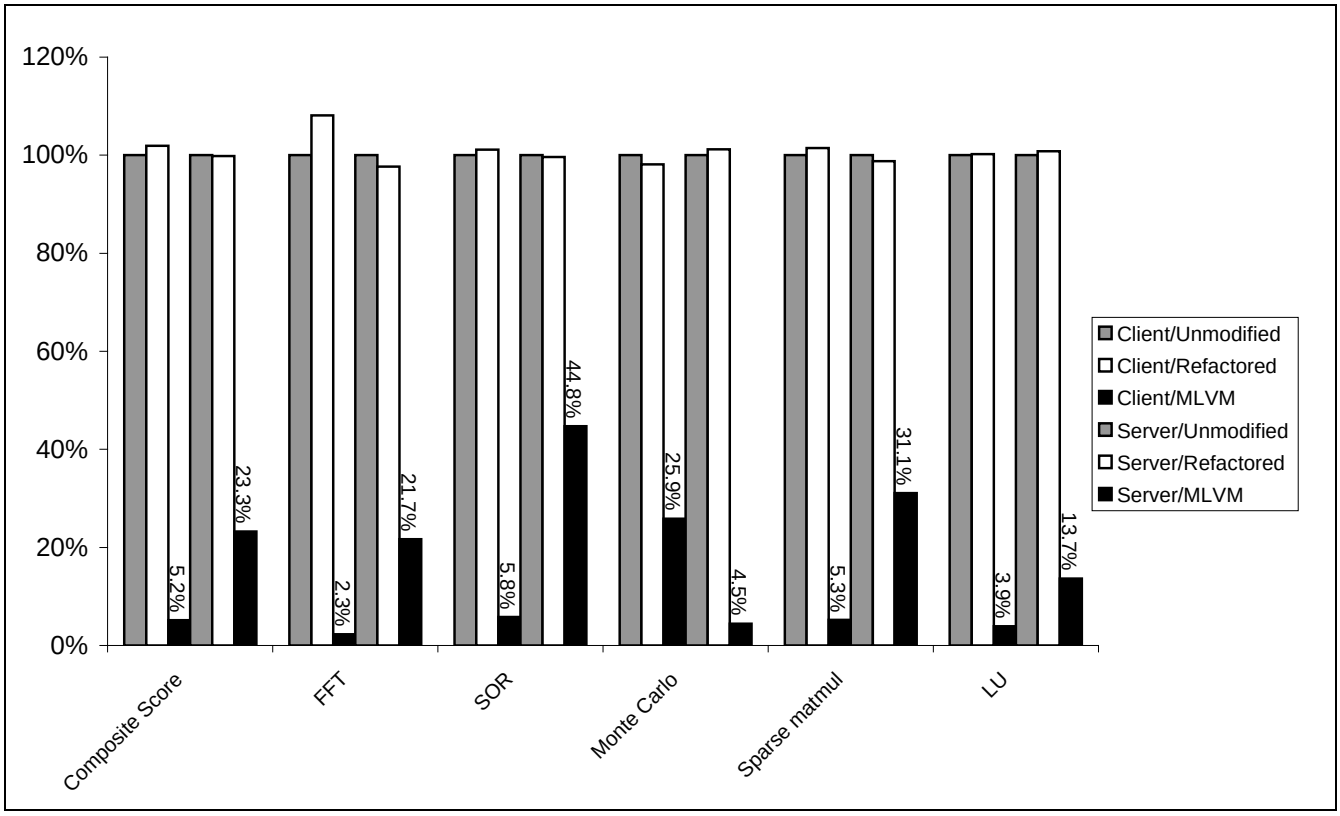

Figure 2. Normalized results of the refactored version (white bars) and the invokedynamic version (black bars with numbers) against the original version of SciMark 2.0 (gray bars).

According to JSR 292 [16], a dynamic call site, which is an instance of `invokedynamic` instruction, must contain name-and-type information. Its associated target method handle must also conform this type information; otherwise the execution will fail during runtime.

The rules described in Figure 1 use *C*, *D*, *E* to represent classes, while *I* represents `java.dyn.InvokeDynamic` and *V* represents `void`. They use *c*, `this` as an object, *m* as a method name, `<init>` is the constructor name, and *h* is a direct method handle. An expression is denoted by *e*, and $\bar{e}$ means a sequence of expressions e_0 … e_n. A dot notation means sequence concatenation. An arrow means translation. Finder methods have the similar semantics to those in JSR 292 [16]. Auxiliary function `type()` returns a sequence of types of expressions. Writing `invoke<*>(…)`: *D* means this instruction returns an object of type *D*, and of course writing *e*: *C* means *e* is an expression or an object of type *C*.

One may notice that semantics of `invokedynamic` in these rules use a method handle as a parameter instead of name-and-type. This is intended to express their related finder methods. It also allows omitting semantics of bootstrap method from the translation rules.

4. BENCHMARKS

The SciMark 2.0 benchmark suite is a set of micro-benchmarks for scientific and numerical computing [14]. Beside the original version of SciMark 2.0, there are two versions of these benchmarks used in this paper. The first one is the refactored version of SciMark 2.0, and the second one is the first one re-compiled by a binary translator according to rules described in Section 3.

Table 1. Experimental results.

Configuration	Benchmark score (Mflops)					
	Composite Score	**FFT**	**SOR**	**Monte Carlo**	**Sparse matmul**	**LU**
Client / Unmodified	547.08	436.59	634.91	98.65	627.85	937.41
Client / Refactored	557.48	471.93	641.89	96.79	637.05	939.72
Client / MLVM	28.47	9.98	37.13	25.52	33.00	36.71
Server / Unmodified	895.33	618.17	909.51	342.32	741.64	1865.00
Server / Refactored	893.68	603.63	906.24	346.29	732.64	1879.62
Server / MLVM	208.52	134.15	407.03	15.28	230.69	255.44

The main reason of choosing SciMark 2.0 is that it is simple, so that it can be manually refactored. Beside its simplicity, this

benchmark suite is fit for measuring performance of `invokedynamic` because it is CPU-bound, i.e. there is no benchmark program in this suite that performs input/output operations. More importantly, using this benchmark suite to measure `invokedynamic` reflects how well this new mode can execute and optimize trivial final methods that contain only a simple primitive operation.

The original version of SciMark 2.0 cannot be used to correctly measure `invokedynamic` because its benchmarks mainly use primitive operations. Thus, refactoring all primitive operations to method calls is required. In fact, there are two major reasons why refactoring is required for existing benchmark programs before `invokedynamic` translation can be applied First, converting all primitive operations to be method calls in order to make benchmark applicable for `invokedynamic`. Secondly, avoiding limitations of the current build of MLVM by manually changing every unsupported method signature. For every method to be called through `invokedynamic`, internal mechanism of MLVM requires reordering its primitive parameters to be after reference parameters for performance reasons. For example, `void m1(int, String)` is invalid to dispatch, while `void m2(String, int)` is a valid method signature. However, the automatic reordering algorithm has not been implemented in MLVM yet, at the time of writing this paper. A manual process is still needed to do so.

After refactoring, the first suite of benchmark programs is re-compiled with the binary translator to be the second suite. All invocation instructions are replaced by `invokedynamic`. Each benchmark class is also enhanced by adding a bootstrap method. Required method handles are obtained during each class initializer.

The experiments were conducted on an AMD64 machine running 32-bit Ubuntu 10.4 with Linux 2.6.32-24-generic. A custom version of MLVM was built using JDK 1.7-b103 as the imported runtime. MLVM's source code was tweaked to allow using `Lookup.findSpecial` on `<init>` methods. This modification will be discussed in Section 5.

Then original, refactored and `invokedynamic` versions of SciMark were run on two VM configurations, therefore there were six different configurations, namely, Client/Unmodified, Server/Unmodified, Client/Refactored, Client/MLVM, Server/ Refactored and Server/MLVM, in the experiments. The Client/Refactored is the client VM running the refactored benchmark programs, while the Client/MLVM configuration is the client VM running the `invokedynamic` version of benchmark programs. The Server/Refactored and the Server/MLVM configurations are similar to those two, but using the server VM rather than the client VM. Two configurations for the original version of SciMark benchmarks are denoted by Client/Unmodified and Server/Unmodified, respectively.

All configuration runs with `–noverify` switch because the bytecode verifier rejects the `invokedynamic` version of the benchmark programs, while it is perfectly valid to run. This limitation will also be discussed in Section 5.

Benchmarks ran ten times for each configuration. Two slowest and two fastest results were removed. Then remaining six results were averaged. Note that standard derivations of all six configurations are insignificantly small. SciMark 2.0 contains five benchmark programs. Performance is measured in Mflops unit. The first column, namely Composite Score, is an average of remaining five columns. Experimental results are shown in Table 1.

Performance results of `invokedynamic` on the server VM from FFT, SOR and Sparse matrix multiplication are faster than expected. They are about 22-44% of native Java invocations. The `invokedynamic` mode on the server VM performed strangely with the Monte Carlo benchmark as it is clearly slower than the client VM, while this benchmark on the normal server VM ran faster than that on the normal client VM. Figure 2 illustrates the normalized benchmark results of all six configurations against the original version of SciMark 2.0.

For the `invokedynamic` mode of the client VM, it is clearly that its performance is not that good due to the beginning of its implementation. Note that the MLVM development team has firstly implemented `invokedynamic` for the server VM, and then ported it to the client VM later.

5. IMPLEMENTATION NOTES

There are two additional issues found during the implementation of the binary translator for compiling benchmark programs. Firstly, the finder method `findSpecial` should be able to obtain method handles of `<init>` methods. Secondly, bytecode verification should be taking `invokedynamic` into account.

5.1 Finding Special Method Handles

The current design decision of JSR 292 [16] does not allow using method `Lookup.findSpecial` to retrieve method handles of constructors, namely `<init>`. They are allowed to obtain via `Lookup.findConstructor` only. Method `findSpecial` are used for obtaining handles of super class' methods. Programs in Figure 3 and Figure 4 demonstrate this issues using pseudo JVML codes.

```
abstract class  A { }
class B extends A { }
```

Figure 3. Declaration of abstract class A, and sub-class B.

```
.class B extends A
.method <init>()V
 aload 0
 invokespecial A.<init>()V
 return
.method end
```

Figure 4. Pseudo JVML codes for class B.

However, this situation requires a proper `invokedynamic` call, which links to `<init>`. The reason is that the default constructor of class B compiled by a standard `javac` compiler always contain an `invokespecial` instruction that calls `<init>` of class A. To chain object initialization correctly, a method handle used here is required to obtain from calling `Lookup.findSpecial` with `<init>`, and not `Lookup.findConstructor`. But putting an instance of `invo-`

kedynamic as the first outmost call in a constructor leads to another verification problem, described next.

5.2 Bytecode Verification Problem

According to the JVM specification [9], the first outmost call in a constructor of a class must be an `invokespecial` call to another constructor of super class or the class itself. However, replacing it with `invokedynamic` that lately binds with a target method handle obtained from `Lookup.findSpecial` gives exactly the same semantic.

Although, the bytecode verifier rejects such class, but it runs fine with `–noverify` switch, as demonstrated by the `invokedynamic` version of SciMark 2.0 in Section 4.

This could be an open question for the specification of `invokedynamic`. Should the verifier be relaxed to accept such class? To motivate this relaxation, one of the important and interesting use cases of putting `invokedynamic` there is that it allows to have dynamic call join points, in terms of aspect-oriented programming (AOP) [7], for `super()` and `this()` for free. If there were an `invokedynamic` version of an AOP system, it could advise these join points of a class to possibly change semantics of `super()` and `this()` at runtime, for example.

```
.class B extends A
.method <init>()V {
 aload 0
 invokedynamic InvokeDynamic.”0”()V
 return
.method end

.method static <clinit>()V
 ...
 aload lookup
 invokevirtual Lookup.findSpecial(
   A.class, ”<init>”,
   type(void.class) , B.class
 )
 aastore mh[0]
.method end

.method bootstrap(...)CallSite;
 aaload mh[0]
 new CallSite
 dup
 invokespecial <init>(MethodHandle;)V
 areturn
.method end
```

Figure 5. An invokedynamic version of class B.

6. RELATED WORK

There are several benchmark suites, such as Java Grande [10], SPECjvm [17], and DaCapo [2] that can be modified and used for measuring performance of `invokedynamic`. However, there is currently no benchmark suite that has been specially designed for it. The `invokedynamic` version of SciMark 2.0 developed by the work presented in this paper seems to be the first one that is publicly available[1].

Performance measurement of `invokedynamic` can also be done from the perspective of language implementers. For example, JRuby [12] is a language adopted and ship with `invokedynamic`. It comes with its own set of benchmarks with can run against other Ruby implementations. As previously mentioned, its sets of benchmarks cannot directly measure performance of `invokedynamic` and be compared side-by-side to the Java's native invocations. However, the JRuby team reported that they could substantially improve JRuby's performance using this new execution mode.

7. CONCLUSION AND FUTURE WORK

This paper has presented the results from running an `invokedynamic` version compared to the original and refactored versions of SciMark 2.0 on a recent build of the Da Vinci Machine project, a branch of JDK 1.7 that supports the new `invokedynamic` bytecode. The experimental results show that the new execution mode with direct method handles on the server VM is just 2-5 times slower than native Java invocations, except that Monte Carlo benchmark performed strangely on this configuration. Also from the results, this mode on the client VM still requires further performance tuning.

The `invokedynamic` version of the SciMark 2.0 suite has demonstrated that it can be used to directly measure performance of this new execution mode. One of its benchmark programs can identify the slowness behavior of the server VM, as discussed in Section 4.

Another contribution of this paper is an implementation of the binary translator for compiling normal Java programs into equivalent `invokedynamic` programs, in Section 3. In addition, this paper also identified a limitation of the bytecode verifier as the current verifier fails to accept some valid `invokedynamic` programs.

The development of the `invokedynamic` mode is moving fast to its final stage. During this development, robust suites of benchmarks are required to tackle performance issues. To reflect performance of `invokedynamic` in real world scenarios, the binary translator used in this paper is being improved to be able to compile the whole the DaCapo benchmarks suite [2].

8. ACKNOWLEDGEMENTS

This work was funded by Suranaree University of Technology. The author thanks anonymous reviewers for useful comments on the early version of this paper.

[1] The `invokedynamic` version of SciMark 2.0 is available to download from `http://github.com/chanwit/jdyn`.

9. REFERENCES

[1] Bodden, E. Invokedynamic slower than reflection? (ca. 2010) URL: http://www.mail-archive.com/mlvm-dev@openjdk.java.net/msg01720.html.

[2] Blackburn, S. M., Garner, R., Hoffmann, C., Khang, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. B., Moss, B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., and Wiedermann, B. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, USA, October 22 - 26, 2006). OOPSLA '06. ACM, New York, NY, 169-190. DOI= http://doi.acm.org/10.1145/1167473.1167488.

[3] Da Vinci Machine project, the. (ca. 2010). URL: http://openjdk.java.net/projects/mlvm.

[4] Hickey, R. et al. Clojure project (ca. 2010). URL: http://clojure.org.

[5] Hugunin, J. et al. Jython project (ca. 2010). URL: http://www.jython.org.

[6] Kaewkasi, C. *An Aspect-Oriented Approach to Productivity Improvement for A Dynamic Language using Typing Concerns*. PhD thesis, The University of Manchester, 2009.

[7] Kiczales, G., Lamping, J., Mendhekar A., Maeda, C., Lopes, C., Loingtier, J. and Irwin, J. Aspect-Oriented Programming, In *Proceedings of ECOOP'97*, Lecture Notes in Computer Science, Springer, vol.1241, pp.220-242, 1997.

[8] Koenig, D., Glover, A., King, P., Laforge, G., and Skeet, J. 2007 *Groovy in Action*. Manning Publications Co.

[9] Lindholm, T. and Yellin, F. 1999 *Java Virtual Machine Specification*. 2nd. Addison-Wesley Longman Publishing Co., Inc.

[10] Mathew, J. A., Coddington, P. D., and Hawick, K. A. 1999. Analysis and development of Java Grande benchmarks. In *Proceedings of the ACM 1999 Conference on Java Grande* (San Francisco, California, United States, June 12 - 14, 1999). JAVA '99. ACM, New York, NY, 72-80. DOI= http://doi.acm.org/10.1145/304065.304101.

[11] Mozilla Corp. Rhino: JavaScript for Java (ca. 2010). URL: http://www.mozilla.org/rhino.

[12] Nutter, C. et al. JRuby project (ca. 2010). URL: http://kenai.com/projects/jruby.

[13] Odersky, M. and Zenger, M. 2005. Scalable component abstractions. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (San Diego, CA, USA, October 16 - 20, 2005). OOPSLA '05. ACM, New York, NY, 41-57. DOI= http://doi.acm.org/10.1145/1094811.1094815.

[14] Pozo, R. and Miller, B. Java SciMark 2.0 (ca. 2010). URL: http://math.nist.gov/scimark2.

[15] Rose, J. 2009. Bytecodes meet combinators: invokedynamic on the JVM. In *Proceedings of the Third Workshop on Virtual Machines and intermediate Languages* (Orlando, Florida, October 25 - 29, 2009). VMIL '09. ACM, New York, NY, 1-11. DOI= http://doi.acm.org/10.1145/1711506.1711508.

[16] Rose, J. 2008. JSR 292: Supporting dynamically typed languages on the Java platform. http://jcp.org/en/jsr/detail?id=292.

[17] Standard Performance Evaluation Corporation. SPEC jvm98 Documentation, release 1.03 edition, March 1999.