MyJVM : a 100% pure Java parameterizable Java virtual machine

January 31, 2003

Abstract

Numerous research works propose Java extensions. Most of these works try to solve a precise problem and offer *ad-hoc* solutions to achieve their goal: language extensions, preprocessors implementation, modifications of the compiler, of the bytecode or of the virtual machine behavior. Most of this changes could be implemented in a common framework by extending a Java Virtual Machine (JVM). Unfortunately, most JVM are not freely available or difficult to manage and/or to extend, and explicit use of their particular features implies a loss of portability. In this paper, we present an open Java Virtual Machine (100% pure Java) designed to be easily extensible (statically or dynamically) by Java programmers.

1 Introduction

The popularity of the Java technologies is now well established. It is mainly due to its portability, to the simple and well defined semantics of the language, but also to the availability of numerous tools that ease application development, maintenance and extension. Despite of these qualities, numerous research works still propose to extend the actual Java framework. Most of these works focus on a precise feature and implement it in an *ad-hoc* manner: language extensions, preprocessors implementation, modifications of the compiler, of the bytecode or of the virtual machine behavior [10]. Since all these extensions are implemented in different frameworks, they are difficult to combine, reuse or compare. We think that most extensions could be implemented in a common framework with a Java Virtual Machine written in Java, designed to be extended, and freely available.

Moreover, Java is now largely adopted as the support language for many computer science courses. Then it is important for students to understand all the fundamental mechanisms hidden behind the scene: in the virtual machine. A practical way to reach this goal consists in modifying the standard behavior of the JVM. For instance, to change method resolution implementation or to change memory allocation scheme. Here again, the availability of free JVM written in Java would be valuable.

In this paper, we present the design and implementation of MyJVM, an open Java Virtual Machine (100% pure Java) designed to be easily extensible (statically or dynamically) by Java programmers. This work is still in progress, but the basic features of the JVM are now available.

The paper is organized as follows. In section 2, we review different kinds of extensions and modifications MyJVM should support. Related projects are then presented. In section 3 the global architecture of MyJVM is detailed. Then, in section 4, basic and dynamic extensions of the JVM are explained. Section 5 summaries our conclusions.

2 Objectives and related work

MyJVM is designed to clearly separates concerns and to come up to the requirements of different categories of modification of a virtual machine. We now give examples of such categories, and related works.

First, our virtual machine must allow fine control of the standard Java features. Indeed, predefined mechanisms offered by Java sometimes induce inescapable constraints. For instance, the garbage collector usually lighten the programmer's work, but can be a problem if he needs fine control on the garbage collector activation or algorithm.

On the contrary, some useful mechanisms are not integrated into the virtual machine. They must be easy to add in MyJVM. For instance, deadlock situations or *data races*, i.e. concurrent access to data, could be detected automatically by virtual machines [17, 13]. This kind of problems are actually solved by extensions of the language [13], or by adding the necessary mechanisms into the virtual machine (as, for example, in JanosVM [16]), or by writing programs analyzing the traces of the VM (example: JProbe).

Other needs for modifications of the JVM we want to support are bound to extensions or modifications of the language. Aspect Oriented Programming is a main example of the former: it offers to separate the functional concepts of a program from non-functional ones, increasing the ease of reusability, maintenance and parameterization of the program. In AspectJ [8] or HandiWrap [3] approach, aspect code and usual Java code are woven at compile time, and the program is run on a classic JVM. There also exists dynamic approaches [11] where programmers use wrappers of a dedicated library to manipulate aspects. Multi-methods is an example of the need of modification (not extension) of the semantic of the language. They allow method resolution to be done at runtime using both target type and "real" (dynamic) arguments types. Again, this problem can be solved by extension of the compiler, or by packages [5], or by modification of the method resolution algorithm into the virtual machine.

At last, some software automatically adapt themselves to their execution environment. As an example, software that modifies themselves at runtime according to the performances of their execution platform for optimization reasons. This suppose that the execution platform is equipped with an interface offering the functionalities needed by the application.

So the needs for modifications of the JVM are numerous. Some research projects develop JVMs with particular properties. For example, KissMe [15] and PEVM [9] have persistent objects, Jalapeño [1] and JRockit [2] are optimized for server applications, and JanosVM [16] has multi-processes support. Some other research projects develop open JVMs, without particular functionalities but with modifiable components. Most of them are written in C (SableVM [6], Jupiter [4]), some of them are written both in C and Java (Jikes RVM [1]) and some others, only a few of them, are written in pure Java (JavaInJava [14]). The use of C is justified by performance arguments, and the use of Java ease of modification arguments. The explicit use of all these

JVMs in a Java program implies the loss of portability of the program, which is a main quality of Java programs. Furthermore, a modification of one of these JVMs must be made before the execution of a program.

3 MyJVM architecture

We wanted MyJVM to be as easily understandable and modifiable as possible. Modifications can occur at compile-time of the VM or at runtime. Every component of MyJVM is a Java object. The architecture of MyJVM is designed to come up to the following requirements:

- before compiling the VM the programmer must be able to substitute the implementation of a component by an other one, or to add a new component. To achieve this goal MyJVM distinguishes the *type* of the component, i.e. its functionalities inside the VM, from its *implementation*. The type of the component is given by a Java interface, its implementation by a class. The association between types and implementation is described in a static component (repository) of the VM. In order to change (or add) a component the programmer just has to implement an interface (possibly by subclassing its default implementation) and to notify the change (or the addition) of implementation in the repository ;
- at run time any component of the VM must be able to obtain the current implementation of any other component. This goal is again achieved using a repository ;
- at run-time the application the VM is running must be able to interact with the VM's components. This is done by an introspection mechanism that reifies each component of the VM into the heap of the application.

Figure 1 gives a simplified view of the structure of MyJVM.

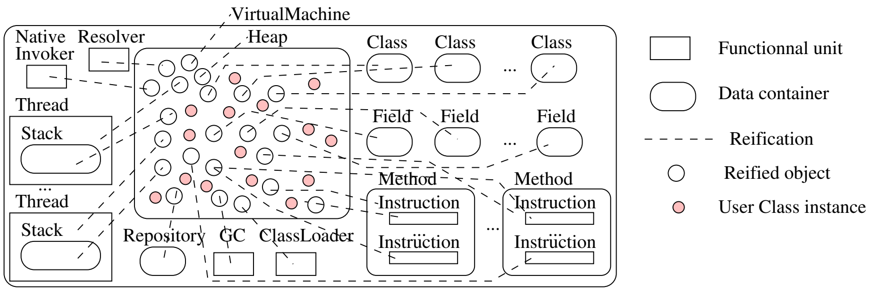

Figure 1: The architecture of MyJVM.

The main component of MyJVM, *VirtualMachine*, is the repository that knows any Java object representing a component of the VM (itself included) and which is able to associate a component type with its implementation class. At start time MyJVM first creates the implementation of the components from their types. Then, it makes any component learn all other components its needs: this is the initialization step of the virtual machine. Once components are all ready, *VirtualMachine* asks the default classloader to load a class. This creates a *Class* component, containing *Method* components composed of *Instruction* components (even instructions are Java objects). MyJVM then creates a thread component, and starts this thread on the *Method* representing the `main` method of this *Class*. The thread is responsible for the sequential execution of instructions and each *Instruction* object is responsible from its own execution.

An important feature of the architecture of MyJVM is to offer the application using it a reflection mechanism in which any component of the virtual machine is entirely accessible. As soon as a component is created by *VirtualMachine*, a *proxy* representing this component is created into the MyJVM user's heap. This object is then accessible by the application MyJVM is running *via* both structural and behavioral reflection. The reflection mechanism enables in particular dynamic changes of the virtual machines, discussed in Section 4.2.

4 Modifications of MyJVM

Modifications of MyJVM are briefly described in this section. Compile-time changes are illustrated by a simple extension of MyJVM and run-time modifications are discussed.

4.1 Example of compile-time extension

We illustrate the extension of MyJVM before compilation by implementing a VM with an additional `complex` primitive type.

The addition of a new primitive type in MyJVM implies the following modifications:

1. the extension of the data storage areas objects concerned with this new type, in order to equip them with services specialized in the manipulation of data of this type. Data of type `complex` can be stored in two areas in the VM: in the stackframes and as fields of objects;
2. once the data areas are extended with appropriates services, the VM must be equipped with new bytecode instructions. These new instructions are defined using the new services of the data areas;
3. the reflection mechanism of Java associates a class object with any primitive type (e.g. `int.class`). A class must be added to MyJVM to define how `complex.class` is built;
4. the addition of a new entry in the constant pool for the literal `complex`.

Step one is achieved for the stack frames by writing a new class that extends their default implementation, and containing four new methods, `pushComplex`, `popComplex`, `storeComplex` and `loadComplex`. The access of `complex` fields into objects must be defined by writing a new class. Steps two is reached by writing a new class per new instruction, and step three and four is done by writing two new classes (one by step). All these new classes must implement an existing interface of MyJVM identifying their category (for example, the interface `Instruction` for classes implementing new instructions) and a new interface representing the component's type for the repository (for example, an empty interface `InstructionComplexAdd` for the class implementing the new instruction `ComplexAdd`). The latter interface must extends the former. The opcodes and number of operands of each new instruction must be declared in a specific repository for opcodes.

Steps one, three and four require each about 10 lines of Java code. Step two also requires about 10 lines of Java code per new instruction. Once they are done, the repository containing the association between the components types and their implementations must be updated. MyJVM is then ready to run with the `complex` primitive type.

4.2 Dynamic replacement of a component

The application MyJVM is running can do introspection on MyJVM's components *via* a programming interface. This programming interface reifies any of MyJVM's component into the application's heap: the application can thus consider these components as objects of its own. Both structural and behavioral reflections are allowed: the application can adapt itself to the runtime platform's characteristics and even can change a component if it does not fit its requirements. As an example, reflection can be used to dynamically change the heap by one that persistently stores newly created instances of particular classes.

The highly reflective capability of MyJVM poses safety and security problems, solved by an event notification mechanism. A dependence graph of the components is implicitly built at start time of the VM: a component `c1` depends on an other `c2` if `c1` uses `c2`. Each component registers an event listener on every components it depends on. When a request for introspection of a component is made by the application MyJVM is running, the request (containing its characteristics) is propagated as an event through the reverse dependence graph. A component receiving the event can agree or disagree with the request. If one component disagrees the introspection request is rejected. The cause of the reject is communicated to the request source, which can adapt its request to the requirements contained in the cause.

An other problem posed by the replacement of a component is the transfer of the knowledges of the old component to the replacing component. When the data should be known from the component independently of its implementation this is done by declaring the transfer services in the interface representing the abstract type of the component. Otherwise, the replacing component is responsible to obtain informations from existing components.

5 Conclusion

At the moment MyJVM is still a work in progress. Our practical experience of its architecture shows that MyJVM is easily understandable, and that most of modifications at compile-time are simple: the replacement or the modification of a component only takes a few lines. The implementation of MyJVM is developed enough to run reasonable sized test programs: the main features of a JVM are implemented. Some branches of MyJVM, like the dynamic behavioral introspection features, are still in development.

As MyJVM is written in pure Java it can be considered as a part of applications. Its explicit use by applications does not imply a loss of portability.

The execution time tests show that the choice of writing MyJVM in pure Java was not an utopia. Even if we tried to reduce execution times, performances are not a critical concern for us: our JVM is to be used as a research and teaching platform. Today our VM is about a hundred times slower than HotSpot. This relatively slow speed allows non-trivial (but reasonable sized) test programs to run. We are working on automatic optimization techniques based on bytecode generation that would multiply by ten (in an average situation) the speed of MyJVM. This work should make MyJVM's speed comparable to a standard JVM.

Some students currently use MyJVM to deepen their knowledges on the Java technology. Our JVM is used as a support for educational projects.

The Java source code of our virtual machine will be soon available on the Web under a GPL license, for people who want to use it as a research or teaching test platform.

References

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P.Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Virtual Machine. *IBM System Journal*, 39(1), February 2000. Jalapeño is the old name of the Jikes RVM Virtual Machine.

[2] B. Antonsson, J. Dahlsteclt, M. Joëlson, M. Lagetgten, S. Latsen, D. Lindholm, O. Lindholm, P. Lönnebting, M. Vidstedt, and C. Wilung. JRockit - The Faster Server JVM. White paper.

[3] J. Baker and W. Hsieh. Runtime aspect weaving through metaprogramming. In *Proceedings of the 1st International conference on Aspect-Oriented Software Development*, 2002.

[4] P. Doyle and T. Abdelrahman. Jupiter: A modular and extensible JVM. In *Proceedings of the Third Annual Workshop on Java for High-Performance Computing, ACM International Conference on Supercomputing*, pages 37–48. ACM, June 2001.

[5] R. Forax, E. Duris, and G. Roussel. The Java MultiMethod Framework. In *TOOLS Pacific'00 Proceedings*, November 2000.

[6] E. Gagnon and L. Hendren. SableVM: A research framework for the efficient execution of Java bytecode. In *Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM '01)*, April 2001.

[7] M. Golm and J. Kleinöder. MetaXa and the Future of Reflection. In *OOPSLA Workshop on Reflective Programming in C++ and Java*, pages 1–5, 1998.

[8] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, 2001.

[9] B. Lewis, B. Mathiske, and N. Gafter. Architecture of the PEVM: A High-Performance Orthogonally Persistent Java Virtual Machine. In *Ninth International Workshop on Persistent Object Systems: Design, Implementation and Use*, September 2000.

[10] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2 edition, 1999.

[11] R. Pawlak, L. Duchien, G. Florin, and L. Seinturier. Dynamic wrappers: Handling the composition issue with JAC. In *TOOLS 2001*, 2001.

[12] I. Piumarta, B. Folliot, L. Seinturier, and C. Baillarguet. Highly configurable operating systems: the VVM approach. In *Proceedings of the ECOOP'2000 Workshop on Object Orientation and Operating Systems*, June 2000.

[13] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. In *16th ACM Symposium on Operating Systems Principles*. ACM, 1997.

[14] A. Taivalsaari. Implementing a Java Virtual Machine in the Java programming language, 1998.

[15] S. J. Tjasink. PLAVA: a persistent, lightweight Java Virtual Machine. Master's thesis, University of Cape Town, Faculty of Sciences, Departement of Computer Science, February 1999.

[16] P. Tullmann, M. Hibler, and J. Lepreau. Janos: a Java-oriented OS for Active Networks. *IEEE Journal on Selected Areas in Communications*, March 2001.

[17] A. J. Van Engen, M. K. Bradshaw, and N. Oostendorp. Extending Java to support shared resource protection and deadlock detection in threads programming. *ACM Crossroads*, 2001. http://www.acm.org/crossroads/xrds4-2/dynac.html.