Towards Small Portable Virtual Machines

Noury Bouraqadi[a,b,c,*], Luc Fabresse[a,b,d]

[a] *Université de Lille Nord de France*
[b] *Ecole des Mines de Douai*
[c] *http://car.mines-douai.fr/noury*
[d] *http://car.mines-douai.fr/luc/*

Abstract

Due to the increasing number of new devices, porting software from one platform (hardware and OS) to another one becomes an even more critical task. Virtual machines (VMs) provide a first level of abstraction by making programs platform-independent. But, VMs which are developed in low-level languages (mainly in C) still require to be ported. To tackle this issue and ease the VM porting task, we propose to restrict VM features, and thus reduce the amount of code written in low-level languages. Our vision is to remove almost all platform specific code from a VM, except a Foreign Function Interface mechanism (FFI). We argue that, based on this FFI, it is possible to write platform specific efficient libraries in a high-level language (Smalltalk). We experiment this approach by replacing the network plugin of the Squeak VM by a library developed based on the Alien FFI. Our first benchmarks show that this solution has no significant impact on performance.

Keywords: Virtual Machines, Portability, Foreign Function Interface

1. Introduction

One of the benefits of a Virtual Machine (VM) is that it clearly decouples the application code from the operating system (OS) and the hardware. The abstract layer represented by the VM makes the application code independent of any platform (OS and hardware) and particularly the one it has been compiled on. However, a VM itself is by definition platform dependent. The well-known Java slogan "compile once, run everywhere" requires that a virtual machine has to be developed for every existing platform (hardware and OS). However, porting a VM is a difficult task mainly because it is written in a low-level languages (typically the C language). We think that this process should be simplified.

In this paper, we present an approach to ease VM porting. We advocate that a VM should be as minimal as possible, but should at least integrate a

*Corresponding author

Foreign Function Interface (FFI) to interoperate with the underlying platform. Based on the FFI, we argue that it is possible to build other platform dependent functionalities using a high-level language and IDE (Smalltalk). We experiment this idea through the implementation of OCEAN, a free portable networking library based on the Alien-FFI library in Pharo. Our main goal is to provide a first validation to our approach for making smaller VMs by replacing the socket VM support by the OCEAN library. OCEAN also servers as a vehicle to study the issues of testing low-level features, identifying a clean design of a library that complies with our approach while still ensuring performance. First benchmarks show that this approach's performance is as good as the original VM-based solution.

The remainder of this paper is organized as follows. Section 2 describes the architecture of the Squeak VM through our experience on porting of this VM to a robotic platform. Section 3 explains our vision of how to make a smaller and easy to port VM. In section 4, we present the design and the implementation of the OCEAN library. We provide an evaluation of the library compared to the socket VM support. The two last sections discuss the related work and conclude the paper mentioning some interesting perspectives.

2. Porting the Squeak VM: a Hurdle Race!

In the WifiBotST[1] robotics project, we had to port the Squeak VM to the Wifibot[2] robotic platform. We have to admit that this task was not so easy. Before explaining the problems that arose, we describe hereafter the Squeak VM source organization and the build process.

The Squeak VM is basically written in C. One part of this code is hand written, while the rest is generated from code written in Slang ([Gre02]). Slang is a subset of Smalltalk, that maps directly to C constructs ([IKM+97]). While it allows one developing with the comfort of the Smalltalk high-level IDE, Slang remains a low-level language. Object-oriented feature are forbidden!

The handwritten C code contains OS specific functionalities organized as *plugins* (either static or dynamic libraries) such as SoundPlugin, SocketPlugin or FilePlugin. Generally, this code is platform specific but it may be partially reused across different platforms (e.g POSIX compliant plugins). The Slang code contains the core of the VM such as the interpreter as well as some plugins code.

Figure 1 shows the two major steps in the compilation process of a Squeak VM:

Step 1: *C code generation from Slang code.* The VMMaker tool automatizes the generation of C code from Slang code. The generation process can be parametrized and a plugin may be put as *internal* or *external*. An

[1] `http://vst.mines-douai.fr/WifiBotST`
[2] `http://www.wifibot.com`.

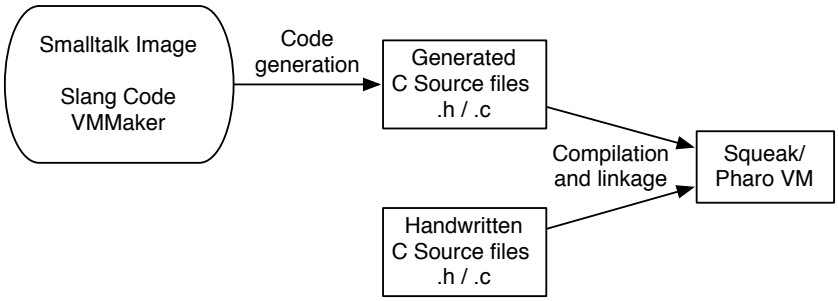

Figure 1: Build steps of the Squeak VM

internal plugin is statically linked in the Squeak VM executable whereas an external plugin is compiled as a dynamically loadable library (.dll, .so, .dylib, ...).

Step 2: *C code compilation and linkage.* This step requires a compilation toolchain dedicated to the targeted platform. In our experiment, we used a *cross-compilation* toolchain, i.e the Squeak VM for the Wifibot robot (ARM processor), was built on a different machine (PC with Intel processor).

We faced severals problems during this port. Our first plan was to build the smallest VM possible by removing most plugins because of the resource constraints on the Wifibot (ARM processor clocked at 400MHz, with 64MB of RAM and 32MB of flash storage). But, some plugins that aren't required by our application such as the Regular Expression plugin (RePlugin), were still mandatory to build a working VM . The next issue has been to decide if a plugin should be integrated as an internal one or an external one. At compile time we figured out that some plugins could not compile when made external. Yet another issue is that dependencies between plugins are hidden inside the C header files. Unsatisfied dependencies was not revealed until the execution of the produced VM, were we noticed that some dynamic libraries were missing. Such issues were tedious to tackle, since the Wifibot has no screen, and the log file only indicates that a primitive has failed.

We can summarize the questions and/or problems related to the VM porting task:

- What is the minimal set of plugins?
- What is needed in the compilation and/or execution environments?
- How to ease error analysis and debugging?

In the following section, we propose an approach to solve these problems and therefore ease the porting task.

3. Our vision of a Small Portable VM

The main idea is to pull up functionalities from the VM to the image side as shown in Figure 2. We suggest to replace VM plugins by Smalltalk code.

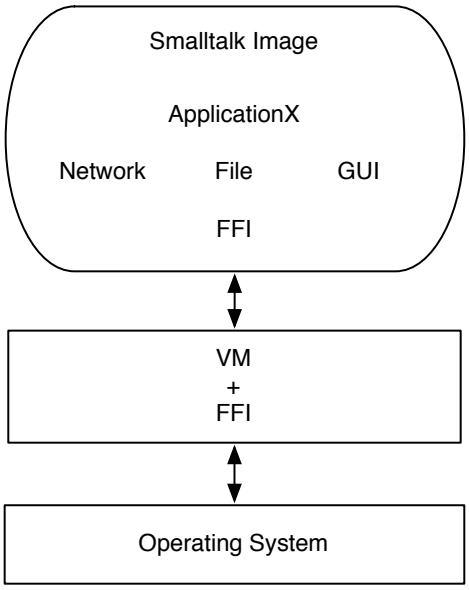

Figure 2: Replacing VM plugins with regular Smalltalk code based on a FFI library

Easier portability is a major interesting consequence of this code shifting. The VM gets smaller and therefore there is less code, written in a low-level language (typically the C language). When porting the VM to a new device, developers will have to spend less effort since even platform specific code will be written in Smalltalk. All Smalltalk powerful constructs, its libraries, tools, and facilities (such as SUnit, Refactoring Browser) can be used.

Easy shrinking is another interesting consequence of pulling VM functionalities to the image level. By switching low-level functionalities to Smalltalk code, where we can use the Smalltalk code package manager (such as Monticello). Thus, functionalities that are useless for some application can be simply unloaded from the image. For example, only the code for current platform the image is running on, can be loaded. This easy shrinking is a considerable benefit knowing that one needs to recompile the VM to remove internal plugins.

Three questions arise from our proposal:

- How to structure platform specific code?
- How to access operating system low-level functionalities?
- How to ensure performance?

Regarding platform specific code, we suggest to adopt a solution based on the Abstract Factory and the Bridge design patterns ([GHJV95]). The Abstract Factory pattern "*provides an interface for creating families of related or dependent objects without specifying their concrete classes*". The Bridge pattern "*decouples an abstraction from its implementation so the two can vary independently*". This approach is already in use in the Smalltalk library. An example is the FileDirectory that has one subclass per target platform. The subclass appropriate to the hosting operating system can be retrieved by sending the activeDirectoryClass message to FileDirectory. The corresponding method and others rely on the instance of SmalltalkImage that reifies the image. Indeed, SmalltalkImage does implement several methods to retrieve information about the hosting platform such as isLittleEndian (from the *endian* category), and platformName (from the *system attribute* category).

However, some functionalities still require accessing the operating system or external libraries (such as OpenGL). We propose to rely on a Foreign Function Interface (FFI) such as Alien ([Mir07]). The idea is to write as much as possible code in Smalltalk to benefit from its portability, and from the development tools it provides (SUnit, Debugger, Refactoring Browser). Only, third party (OS, external libraries) functionalities are called through Alien.

The last question is the impact of the solution on performance. Indeed, performance and more specifically computation speed is one important reason that makes traditional VM development rely on low-level languages. Although we believe that this question of performance is important regarding the usability of our proposal, we decided to approach the problem in an agile-like fashion. We first focus on the feasibility of the approach and its design. The performance question is deferred to the last step of a project life-cycle by evaluating the computation cost and memory footprint of a cleanly designed solution. This approach was adopted in our first experiment described in section 4. Figures we obtained so far show that our approach is promising, since it is faster than the existing C-based implementation. Furthermore, we envision that performance is likely to improve if we use a JIT compiler.

4. Experience Report: Replacing of the Socket VM Plugin

In this section we describe our ongoing project Ocean[3], that aims at providing a free replacement[4] under MIT license, to the networking plugin of the Squeak VM by Smalltalk code. We started this project by focusing on the central concept in networking, namely sockets. Currently, we have a fully tested partial implementation of TCP sockets under Mac OS X.

[3] Ocean stands for Object-oriented, Cross-platform, and Effective API for Networking.
[4] Ocean is available on `http://www.squeaksource.com/Ocean.html`

4.1. Design

In the existing implementation of socket in Squeak and Pharo all socket functionalities are merged into a single class that calls primitives provided by the networking plugin. The design of Ocean (see figure 3) differs from this approach at two levels. First, we adopted a more OO design, where we have each kind of socket represented by a different class. Second, we decided that the socket classes should be platform-independent. Platform specific code goes into a class representing the OS networking library. There will be as many library classes as supported platforms. By means of a bridge design pattern, an Ocean socket will use the class that matches the hosting OS. The socket then sends the appropriate messages to the library to perform low-level actions (e.g. connecting, sending and receiving data).

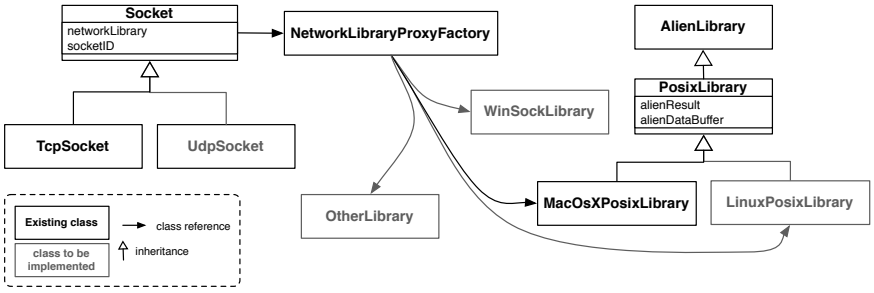

Figure 3: Ocean Design

In our current experiment, we relied on Alien to access networking functionalities provided the Mac OS socket dynamic library sys/socket.h[5]. We choose to wrap the sys/socket.h because it is a POSIX library, also available on other Unix-like platforms such as Linux.

The use of Alien is optional. Other approaches or FFI support might be used. So, network libraries could not share a common superclass. This is why we introduced NetworkLibraryProxyFactory. All classes representing network libraries should be registered in a collection hold by NetworkLibraryProxyFactory. On initialization, a socket requests from this factory the library to use.

It worth noting that in the Alien-based approach, every socket holds an instance of the Alien networking library class. This decision significantly speeds-up Ocean. Indeed, allocating and freeing Alien variable is very costly. So, we decided that the Alien variables that hold functions results (alienResult) and exchanged data (alienDataBuffer) should be created only once and stay allocated during the whole socket lifetime. Because, these variables are specific to a particular implementation of the platform specific part, we decided to store them in the library side. Thus, every socket should have its own instance of the

[5]Functions and constants of the sys/socket.h library were made accessible through instance methods of MacOsXPosixLibrary.

Alien networking library to avoid data incoherence due to concurrent reads and writes. This 1-1 relationship is fully hidden in the library class. The factory requests from the library class an instance. It is the library implementor that chooses whether to answer a singleton or a different instance on each request.

Another decision that impacts performance is related to the data exchange buffer (instance variable alienDataBuffer in the MacOsXPosixLibrary). When sending large amounts of data, a small buffer turns into a bottle-neck. But, allocating a large buffer by default can result in wasting memory when only little data is exchanged. This is why we decided to have buffer which size can adapt to the amount of sent data. By default, its size is about 2KB similar to the buffer size in the Squeak / Pharo Socket. But, in Ocean this size is increased when sending larger amounts of data.

4.2. Test

One of the biggest challenges we faced is related to testing. Indeed, being used to Test-Driven Development (TDD), we started by writing tests that describe the behavior of a socket. However, sockets are infrastructure entities managed by the operating system. Objects that represent sockets in the Smalltalk image are merely facades to these low-level entities. So, we need to collect information about these underlying entities to assert that a test succeeded.

To address this problem, we decided to interact with the operating system. We do use the OSProcess package[6] that reifies the operating system processes. By this means, we automated access to networking administrative common utilities such as *lsof* and *nc*. The *lsof* utility does list open files including sockets[7] and displays their status. The *nc* utility enables setting up arbitrary TCP and UDP clients and servers. It is useful to test socket data sends and receptions. These utilities are initially meant for interactive use. We wrapped them into Smalltalk objects to use them in an automated testing process.

Utilities we used so far for test are specific to the Unix world. For other platforms such as Windows, we need to find alternative solutions. It worth noting that the use of *lsof* and *nc* allowed us to test our approach at early stages. Platform independent tests can be envisioned now using only our implementation and OS library calls. Such tests would be platform agnostic and thus would support port efforts.

4.3. First Evaluation

To evaluate our approach, we conducted a benchmark for network communications. Our experiment were made using Pharo 1.0 under Mac OS X 1.6.3. We compared the performance of the Ocean implementation of sockets with the plugin-based one that comes with Pharo.

[6] `http://www.squeaksource.com/OSProcess.html`

[7] In the Unix world sockets are special kinds files.

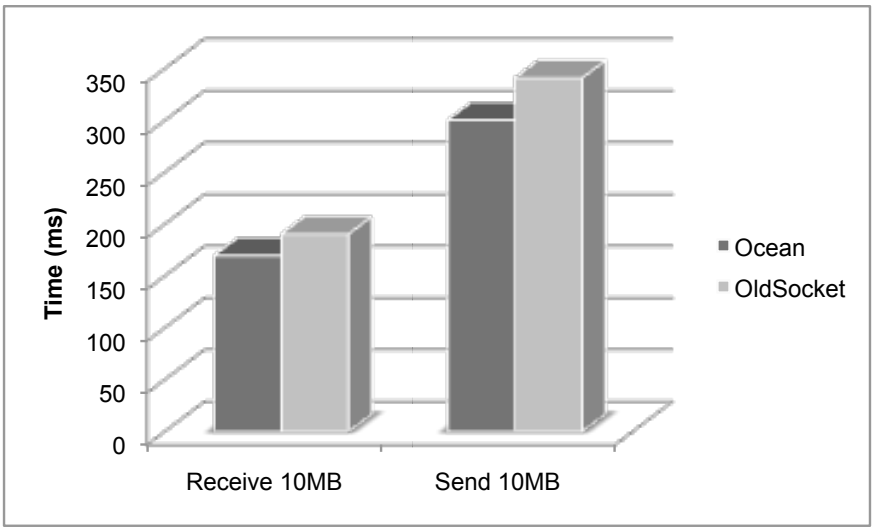

Figure 4: Ocean early sends and reception benchmarks

Our comparison focused on data sends and reception (see figure 4). We computed the average time to send 10MB to a TCP server located on the same machine. We also evaluated the average time to receive 10MB from a TCP server located on the same machine. The result is that Alien-based sockets are about 10% faster than plugin-based sockets for sending data. We conclude that there are no significant differences between the two approaches. Indeed, in a real setting, the network latency is much greater than the time required by socket operations.

Regarding our implementation, as mentioned earlier, we relied on Alien FFI. For purpose of analyzing the impact at the Operating System level in tests, we also used OSProcess that requires its own VM plugin. Nevertheless, we believe that this plugin can be replaced by Smalltalk code combined with FFI calls as done with sockets.

5. Related work

Porting a VM can be more or less difficult depending on how it is performed. We distinguish three families of approaches:

- Hand-coding
- Code generation
- Minimizing the VM functionalities set

This first family is about VMs that are coded by hand, often using low-level languages. As an example, there is the open source Java VM provided by

Sun written in C and C++. Another example is NXTalk [BHH09] whose VM is entirely written in C. The portability process of these VMs is clearly complex since it requires to deal with low-level languages and tools. Moreover, extending the VM to support a specific feature of the target environment would also be difficult.

We already mentioned the Squeak VM work done by the Squeak Central team leaded by Dan Ingalls ([IKM+97]). This work pioneered the generative approach family where the virtual machine C code is generated. This solution allows developers to use the Smalltalk IDE to develop the VM. The VM code is written in Slang, a low-level subset of Smalltalk that directly maps to C, but at the cost of banning all object-oriented features. Another criticism to this approach is the difficulty to test and debug. Since Slang is a subset of Smalltalk, the code corresponding to the VM under test can be run inside the development image. Nevertheless, this execution is significantly slow (about 450 slower than the actual VM ([IKM+02])). Yet another issue results from the fact that Slang supports C code inlining in Slang methods. Such code can not be tested. However, there is a construct that provides the Smalltalk counterpart for tests. Still, the Smalltalk code can not be fully equivalent to the C one. So, developers still have to test the VMs built from C generated C code because testing the generation process is not enough. Last, the generative approach can also be criticized for being static. With platform specific code on the image side, updates and new OS bindings could be directly loaded without stopping the VM.

Pypy ([BR07]) is another instance of the generative approach in the context of the Python language. A custom translation toolchain compiles the interpreter developed in RPython a subset of Python to a full VM. Criticisms made above about the current Squeak VM approach applies to Pypy since it also relies on a generative approach. However, contrary to Slang, RPython is a high-level language that supports type inference.

The proposal we describe in this paper belongs to the last family of approaches, i.e. those that minimize the set of VM functionalities. OpenJIT ([OSM+00]) is one of the few instances of this family, that focuses only on the JIT compiler part of the VM. Actually, OpenJIT is a compiler framework to be used for developing different platform specific JITs. An interesting feature of OpenJIT is that it applies to itself by compiling its own java byte-code into native code. It worth noting that the the Exupery [8] project pursues a similar objective in the context of Squeak Smalltalk.

6. Conclusion and Future Work

Porting VMs is a tricky task that currently requires dealing with complex code written in a low-level language. We proposed in this article to tackle this issue by downsizing the VM. Our ideally small VM merely includes a byte-code

[8] `http://wiki.squeak.org/squeak/3842`

interpreter, an object memory, a Just in Time (JIT) compiler, and a Foreign Function Interface (FFI) enabling advanced communications capabilities with the hosting OS. Even these components may be –at least partially– pulled up to the image level as demonstrated by the OpenJIT work ([OSM+00]).

Based on such a VM, it is possible to fully develop in Smalltalk platform specific features. We demonstrated this idea by implementing Ocean, a networking library based only on Alien FFI. Benchmarks show that Ocean sockets exhibit equivalent performances as compared to current VM-based sockets. Even if further analysis are needed, we believe that the same technic can be applied to other platform specific features with little impact on performance. However, there exist some features (such as Matrix operations) where a re-implementation in Smalltalk will be slower that its counter part at the VM level. Nevertheless, we believe that the speed-up resulting from a JIT compiler is likely to compensate this slow-down.

Regarding future work, our first goal is to finish implementing of our Alien-based socket library and porting it to different Operating Systems. Besides, we plan to continue our study by reimplementing in Smalltalk/Alien-FFI other parts of the VM. A first candidate is the plugin that enables OSProcess to interact with the OS. We envision that some tools are yet to be invented to make this process easier. An example is a tool that automatically reifies any system dynamic library given its *.h* C language header file.

References

[BHH09] Martin Beck, Michael Haupt, and Robert Hirschfeld. NXTalk: Dynamic object-oriented programming in a constrained environment. In *Proceedings of the International Workshop on Smalltalk Technologies (IWST)*, pages 38–49, Brest, France, August 2009. ESUG, ACM DL.

[BR07] Carl Friedrich Bolz and Armin Rigo. How to not write virtual machines for dynamic languages. In *3rd Worshop on Dynamic Languages and Applications in conjunction with ECOOP*, Berlin, Germany, July 2007.

[GHJV95] Erich Gamma, Richard Helem, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Gre02] Andrew C. Greeberg. *Squeak: Open Personal Computing and Multimedia*, chapter 9 – Extending the Squeak Virtual Machine, pages 263–291. Prentice Hall, 2002.

[IKM+97] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Allan Kay. Back to the Future. The Story of Squeak, A Practical Smalltalk Written in Itself. In *Proceedings of OOPSLA'97*, pages 318–326, Atlanta, Georgia, October 1997. ACM.

[IKM+02] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. *Squeak: Open Personal Computing and Multimedia*, chapter 5 – Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself, pages 151–170. Prentice Hall, 2002.

[Mir07] Eliot Miranda. *Alien Foreign Function Interface User Guide*, 2007.

[OSM+00] Hirotaka Ogawa, Kouya Shimura, Satoshi Matsuoka, Fuyuhiko Maruyama, Yukihiko Sohda, and Yasunori Kimura. OpenJIT: An Open-Ended, Reflective JIT Compiler Framework for Java. In Elisa Bertino, editor, *Proceedings of ECOOP*, number 1850 in LNCS, pages 362–387. Springer, 2000.