Silicon Virtual Machines

Alberto Manuel Brandão Simões

Departamento de Informática, Universidade do Minho
4710-057 Braga, Portugal
albie@alfarrabio.di.uminho.pt

Abstract This communication looks into some virtual machine implementations, from the common stack-based architecture to the newest register-based one. It evaluates the feasibility of their implementation on silicon and how this approach can improve performance while maintaining language flexibility. Silicon virtual machines are not successful in the CPU market, mainly due to the inherent limitations of a virtual machine architecture: these are developed too centered on a language, making them very efficient for that language and very slow executing applications developed for other languages. This communication concludes with the statement that the best virtual machine we can make is a simple processor, similar to the ones we use nowadays.

1 Introduction

Virtual machine is a wide spread term but is not a recent concept. A virtual machine is nothing more than an emulation software, some code that translates from one language to another. This first language is, normally, a high level language. The target language is an assembler language or is interpreted for a more low level language or a different paradigm language.

This is the virtual machine definition we will focus on this article. Meanwhile, other Other virtual machine concepts are appearing not truly related with this level of translation from one language to another. For example, on [5] we can see PVM (Parallel Virtual Machine) which objective is to make parallel computation possible abstracting the type and number of nodes in the system. In this case, we call it a virtual machine because it is not really only one machine. As said in [6], we can turn the web into a computer. In this case, we see the web as the virtual machine. Other concept is programmable active memories. For example, on [7], they call virtual machine to programmable active memories because while they are not really a processor they can process data between the store and fetch of the data.

Back to the topic, we will start looking into the Java Virtual Machine. Some time ago, *Sun* come up with a new language that would be revolutionary and become the new programming language standard.

This object oriented language, with a source code very familiar to C and C++ programmers, was developed to be compiled to a *bytecode* that would be interpreted later, on top of another application: the virtual machine. This bytecode would be portable to any architecture where we could find a Java Virtual Machine (JVM for short) binary.

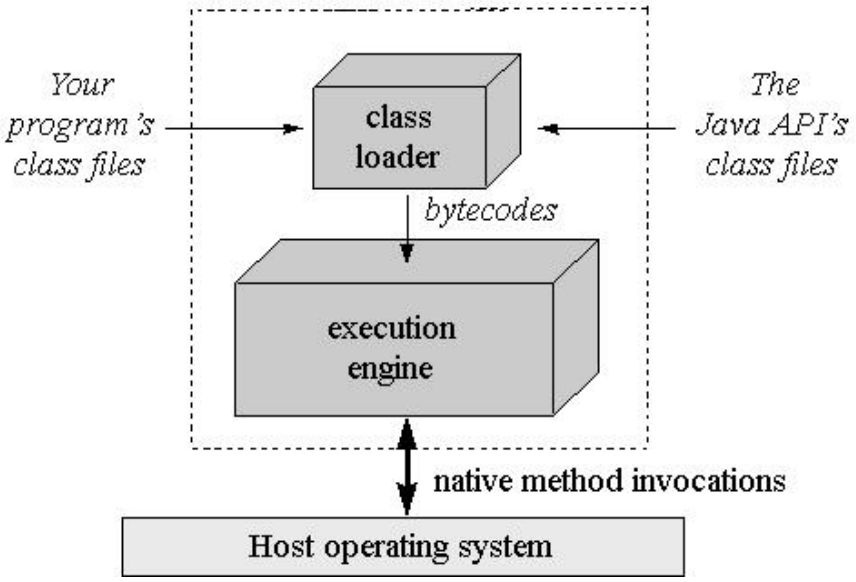

Figure 1: Java Virtual Machine Architecture

This is the story of the term Virtual Machine but, this idea exists for years. If we think a little more about specific languages, we can remember the old ideas to develop *language-centered* machines. Prolog and Lisp are examples of languages that had processors designs to make them faster, running on specific hardware solutions. These monolithic machines could be very fast for that language but completely unusable for common programming.

The virtual machine solution is based in this same idea. Create a specific virtual machine for a language. This virtual stuff means that we will not find hardware for that language. Instead, it will be develop an application that, running over traditional hardware, takes some advantages about the kind of the language it is processing.

Many languages used and continue using virtual machines with this purpose. With most of them, programmers don't notice that code they write is compiled to an intermediate representation language that is interpreted. This code is maintained in memory only for efficiency purposes. Examples for such languages are Perl, Python, Forth or Haskell.

Now, big companies like Microsoft appear in the market with *.net*[1], a new solution for all programmers, running all languages! The Perl community, for instance, is developing a virtual machine (Parrot) to interpret Perl 6.

This is the era of virtual machines! But, why to make these all virtual? Can't we create a generic processor to make all these languages faster without this intermediate layer? On the other hand, could we simply add some instructions to the instruction set of common processors to make them faster for virtual machines?

In this communication we will look to some virtual machine architectures, their features, why they are useful and what's the problem regarding their silicon implementations.

[1] nothing more than a virtual machine

2 Virtual Machine Architectures

This section will cover some virtual machine properties. First we will compare the stack versus the register based approach for operations, following some information about their instruction set.

2.1 Stack vs. Register based

Although almost all virtual machine have registers, they are full stack based. All operands and instructions are manipulated directly in the stack, with the registers being reserved for instruction pointers, stack pointers and miscellaneous flags. This means we will store all temporary information on the stack, needing at least five memory accesses to perform an operation: fetch the instruction, fetch the operators from the memory, put them in the stack, get the result and store the result in the memory, again! Of course some of these accesses can be omitted, for example if the result of the operation will be needed as an operator in the next step.

To solve this fetch hell, Java developers introduced an array concept that is no more than one cache. This array and the stack must be sufficiently polymorphic to handle the eight data type present on JVM: byte, short, int, long, char, float, double and reference. This last is used as a pointer to a generic object in memory. Because each thread uses a different stack, the JVM uses a head shared among all threads.

Current stack implementations examples are JVM[2] and Perl 5.

More recently, appeared another virtual machine concept, like Parrot[2][1], that uses generic registers. In this virtual machine, like on real processors, all operations take place on registers, accessing the memory as standard processors, to fetch instruction and data operands. Parrot contains 32 registers for each data type: integers, numbers, unicode strings and PMCs. This last type is an high level object type that can be instantiated with any object defined. Parrot authors are taking advantage of knowledge regarding register based code optimization, applying them to the parrot assembler, making parrot code run faster.

This register approach is very nice, but it requires a stack as well, like standard operating systems. On the other hand, cyclic operations could be made to fit the system's registers.

What the best approach depends on the situation. The stack approach is the more used because it is the traditional way, and because stack instructions can be very compact. While on a register based approach we have to refer the arguments for an *add* instruction, on a stack based approach the machine knows that the two arguments are in the top of the stack.

2.2 Instruction Set

Virtual Machines has instruction set like real processors, but at different levels. JVM for example, has a standard instruction set, with some more instructions for class hierarchy, exceptions and code robustness. JVM was developed for high quality code, prohibiting the user to access directly the memory. This machine has 201 op-codes. This can look a CISC (complex instruction set computer) machine but, really, most operations are not polymorphic meaning that we have a *load*, *store*, *restore* and such

[2]This virtual machine is being developed for Perl 6 internals, claiming to be generic sufficient to be used for any imperative language. We will focus this language internals because is the only open-source register based virtual machine known.

00 nop	01 aconst_null	02 iconst_m1	03 iconst_0	04 iconst_1
05 iconst_2	06 iconst_3	07 iconst_4	08 iconst_5	09 lconst_0
10 lconst_1	11 fconst_0	12 fconst_1	13 fconst_2	14 dconst_0
15 dconst_1	16 bipush	17 sipush	18 ldc	19 ldc_w
20 ldc2_w	21 iload	22 lload	23 fload	24 dload
25 aload	26 iload_0	27 iload_1	28 iload_2	29 iload_3
30 lload_0	31 lload_1	32 lload_2	33 lload_3	34 fload_0
35 fload_1	36 fload_2	37 fload_3	38 dload_0	39 dload_1
40 dload_2	41 dload_3	42 aload_0	43 aload_1	44 aload_2
45 aload_3	46 iaload	47 laload	48 faload	...

Table 1: Extract from the JVM instruction set

instructions for each type of data available. Looking more in depth, JVM has at most 30 instructions. This low level of instructions and the fact that it uses a stack approach makes the instruction set fit a 8 bit cell. All other values using more space than one cell are split on 2, 4 or 8 cells.

On table 2.2 we can see that instructions were designed to take almost no arguments. This is the reason there are operations indexed with a scalar that represent the index of the cache array.

On the other hand, Parrot has a high level assembler language using a lot of space for instructions op-codes. Examples of this high level is the unicode support directly on the virtual machine, registers using strings instead of characters, trigonometric functions, regular expression support and many more features. Parrot is still under development and current version has 173 different polymorphic operations. This means that there are really a lot of instructions in this virtual machine.

A sample code for Parrot:

```
# Trivial example
print "Hello world!\n"
end
```

Comparing these two machines we can say JVM is RISC and Parrot CISC. Parrot is taking advantage of the virtual approach, implementing operations we won't see in hardware soon. So, if we would like to make it silicon, we will need an auxiliary operating system with libraries for all this stuff.

3 Generic Virtual Machines

All virtual machines seen so far are, like old oriented computers, focused on specific languages. Recently various projects started to make a generic virtual machine. This generic would mean that we can compile any language to produce virtual machine bytecode.

Parrot is one of these project, aiming a general purpose virtual machine for scripting languages. Other more audacious project is *Microsoft .net*. At first this affirmation can sound silly but looking more depth at *Microsoft .net SDK* and looking to what they call *intermediate description language*, we have *assembler code*.

Microsoft have running examples for a lot of languages, from the Microsoft C# to Haskell, Prolog, Standard ML, Perl or Python. They have examples of some specific

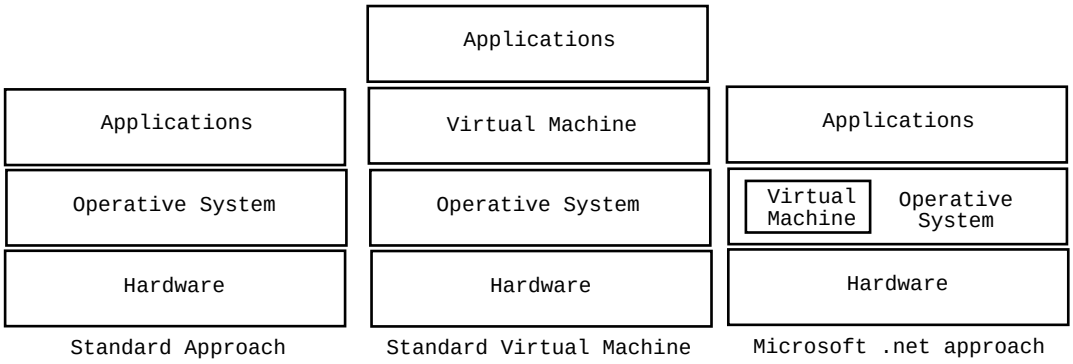

Figure 2: Graphic representation for common virtual machine architectures

languages running in this platform, too. Indeed, *.net* is integrating all languages making them work aside each other. This is the main advantage for using a virtual machine. All these languages generate the same type of language that can be debugged on a same application and run on top of the same virtual machine.

All these languages are working but traditional interpreted languages, like Haskell, Prolog or Perl are not being full implemented. On some cases, some features are being slightly changed or removed.

3.1 Problems regarding generality

Microsoft don't make available much documentation about implementation of the virtual machine (only the intermediate description language syntax), so we will look more deeply to JVM and show why we can't use one silicon processor for Java like picoJava Sun implementation for generic languages.

Let's take a first approach: make other languages produce Java bytecode and run them on JVM. This is possible, but not really a good choice. JVM is very good for Java and object oriented languages, but is totally inefficient for script and functional languages.

On [4] we can read some JVM problems when used to implement a functional language like Scheme. One of the problems referred in this text is object boxing. On JVM, all variables are objects, stored in data memory with the instructions in the bytecode simple pointing to the memory address. In Scheme, for example, there is a huge need of atoms variables that would be placed in memory as standard objects. This would make any operation very slow because of the need for argument fetching.

The author points a solution: we know that memory addresses bellow some address are operative system reserved. In this case, remaining bytes can be used to encapsulate the variable itself. This is a patch, not a solution to be used.

Other problem with generic virtual machines are the type of computation associated with each type of programming language. For example, a virtual machine for Prolog would need a backtracking module, while a Perl virtual machine had to be reflexive.

4 Making virtual real

If generality is difficult, making a silicon virtual machine is impossible. Each virtual machine has it's own architecture, instruction set and different processing rules, making

a silicon virtual machine the agglomeration of all these functionalities.

We can see i8086 as a virtual machine Instruction Set. Modern processors like K6 map this old instruction format to their core instruction format. But, while we can consider this a virtual machine, and a silicon one, this is hard wired and only works for this low level language. This is not what we intend by a virtual machine.

In the processing approach, we have a problem to solve regarding stack oriented machines: we need a quick stack, near the processor. The lack of registers make it necessary to access the memory too much times. There are some approaches to solve this problem. Some create a set of registers that work simulating the stack. These registers will have the n top cells of the stack. Other approach creates a standard *cache* used as a stack.

Both approaches are interesting, but became a really bad solution for threaded Java programming. In Java, each thread has it's own stack meaning how we increase the number of processes, we decrease the velocity for each one.

Regarding the instruction set architecture, and as said before, some virtual machines have a too high level operations that aren't suitable for silicon implementations. For example, there is not a good solution for trigonometric or regular expression implementations.

There is yet another level for code: *firmware*. We can develop a processor with the possibility to add some new instructions in a code cache that is never deleted. This mean that a virtual machine initialization could load a *firmware* to make most common instruction faster.

Crusoe[3] is an approximation to his kind of processors. It is a two level architecture, with a kernel processor and an adaptation layer that translate general code (x86, PowerPC) to the instruction set used by the kernel processor. This is done with a flash ROM that stores this virtual machine code. In this case, direct access to the kernel is not possible. Such a processor could be used as a generic virtual machine if this code layer could be reloaded on run-time when changing application requirements.

5 Conclusion

This communication presented various aspects about virtual machines, their problems and solutions. It doesn't want to take real conclusions but, instead, reflect over them.

First of all, we are using virtual because it is useful not only like Java for portability but for reflexibility, backtracking and many other features needed on specific languages. So, it's a good choice to implement a virtual machine if we need to make a new compiler for a newly created language.

Making a generic virtual machine will join all specific virtual machine modules in one bigger virtual machine (like Microsoft .net). On the other hand, it is traditional that when joining people to develop some sort of standard, many sub-standards arrive!

Finally, most of the virtual machines can't be used for different kind of languages, and almost all are difficult if not impossible to implement on silicon.

References

[1] Simon Cozens. Parrot: Some assembly required. *O'Reilly Perl.com*, September 18, 2001.

[2] Tim Lindholm and Frank Yellin. *The Java(tm) Virtual Machine Specification*. Addison-Wesley, 1997. Second edition.

[3] Vasco Nuno Barreiro Capitão Miranda. Crusoe: An approach for the new era computing. In 3rd Internal Conference on Computer Architecture (ICCA'02), editor, *ICCA'02*, January 28, 2002.

[4] Olin Shivers. Supporting dynamic languages on the java virtual machine. *MIT Artificial Intelligence Laboratory*, April 25, 1996.

[5] V. S. Sunderam, G. A. Geist, J. Dongarra, and R. Manchek. The PVM concurrent computing system: Evolution, experiences, and trends. *Parallel Computing*, 20(4):531–545, 1994.

[6] A. Vahdat, M. Dahlin, and T. Anderson. Turning the web into a computer, Technical report, University of California, Berkeley, 1996.

[7] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard. Programmable active memories: Reconfigurable systems come of age. *IEEE Transactions on VLSI Systems*, 4(1):56–69, 1996.