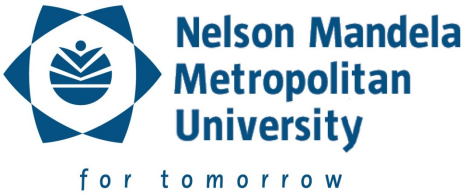
A JIT-Less Type-Mapped Dynamic-ISA Virtual Machine for Many-Instance Applications

March 1, 2015



Author: Douglas Bentley
Supervisor: Kevin Naudé

Submitted in partial fulfilment of the degree Baccalaureus Scientiae (Honores) in Computer Science at the Nelson Mandela Metropolitan University

Acknowledgements

Abstract

Declaration of Own Work

I declare that the entirety of the work contained in this treatise is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Nelson Mandela Metropolitan University will not infringe any third party rights that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Signature:..... Date:.....

List of Figures

List of Tables

Table of Contents

Contents

1 Introduction

1.1 Virtual Machines

To best describe the virtual machine presented in this treatise, a quick summary of virtual machines follows. The heirarchy of virtual machines is presented and our virtual machine is situated within it. Some important virtual machine terminology is introduced.

1.1.1 What is a Virtual Machine?

A *virtual machine* or *VM* is a computer program that "executes software in the same manner as the machine for which the software was developed" [1, pg9]. This program is designed to run on some real machine. We call the real machine that a virtual machine is running on the *host* and the virtual machine the *guest.* The guest machine provides an execution environment for software that is designed to run either on the guest itself or on an actual machine that the virtual machine is emulating. This means that we can use a virtual machine to run programs that are incompatable with the host. The virtual machine allows this by providing a mapping of its state to the state of the host machine on which it is running [1, pg4].

1.1.2 The Types of Virtual Machines

Virtual machines come in two varieties: *process* virtual machines and *system* virtual machines.

A process virtual machine is "capable of supporting an individual process"[1, pg9]. This means that the host runs the guest for as long as a process on the guest machine needs it. Once the process has completed its execution the guest machine terminates[1, pg9]. An example of a process virtual machine is the Java Virtual Machine or JVM. All java programs run on the JVM. An instance of the JVM is started when you execute a java program and killed when its execution is complete.

A system virtual machine "provides a complete system environment" "[1, pg9]. This means that it can support an entire operating system on the guest machine and the many processes that the guest operating system executes. A system virtual machine will terminate when the system is shut down.

1.1.3 The Types of Process Virtual Machines

Since this treatise outlines a process virtual machine we shall look at the different types of process virtual machines and ignore the finer

details of system virtual machines. Process virtual machines can be divided into two categories: *multiprogrammed systems* and *dynamic translators*. These are divided along whether or not the guest machine uses the same instruction set architecture as the host machine.

With multiprogramming the guest and host use the same instruction set. Multiprogramming is supported by most operating systems and allows a user to run many processes at once by making each process think it has access to an entire machine instead of only part of a machine "[1, pg13]. The OS creates an environment per process that it terminates when that process ends execution.

With dynamic translators the instruction set of the host and guest generally do not match. The virtual machine translates blocks of instructions meant for the machine it is emulating and translates them into instructions to be run on the host. Not all code is translated in a dynamic translater. Only code that is used often enough will be translated as there is an overhead involved with translating code. The code that is not translated is interpreted. Interpreted instructions are read, their meaning interpreted and then executed. This interpretation step must happen each time a piece of code is executed so code that is executed enough times will be dynamically translated and cached so later execution is faster. Dynamically translating in this manner is known as ju*st-in-time compilation* (JIT compilation).

1.1.4 Dynamic vs Statically Typed Programming Languages

A dynamically typed language is one in which the type information is associated with values [4, pg4](REF lua VM intro). An example of a dynamically typed language is javascript where the var keyword is used for variables and the type is inferred from the value stored into a variable. A statically typed language is one in which the type information is associated with the variable. An example of this is java where variables are declared using keywords that declare their type (*int*, *string* etc).

1.1.5 Where Our VM is Situated

The virtual machine described in this treatise is a process virtual machine. This process machine will have a new instruction set architecture and hence will differ from any host machine's ISA. The virtual machine does not make use of JIT compilation. It is entirely interpreted. It is also dynamically typed.

1.2 A Virtual Machine for Many Instance Applications

Modern process virtual machines make use of JIT compilers. Both JVM and .Net make use of a JIT compiler [2, 3]. For applications that run many concurrent instances, like a web server which which starts up a new process for every client, a virtual machine that makes use of a JIT compiler defeats the benefits of read-only memory sharing. Since memory sharing is not possible when using a JIT compiler, if we wish to take advantage of it a JIT-less virtual machine is needed. However the JIT compiler compiler is an important feature of a VM that allows for code given to it at runtime to be compiled into machine code that can run directly on the host. This compiled code is cached and allows for future execution to be faster. Sacrificing the JIT compiler to allow memory sharing presents us with a need to find alternative ways to let our programs execute quickly.

1.2.1 Type-Mapping

The idea to be explored for a more performant JIT-Less VM for a dynamically typed language is to make use of a *finite state space*. The VM has a small number of registers and instructions that it can perform. For a dynamically typed VM these instructions may be able to take arguments of different types. For instance an instruction to add may take two integers or an integer and a float or two floats. The instruction has to discover the types of the arguments and perform the correct action. A finite state machine can be used to keep track of the types of the values in all of the registers. Whenever the type in a register is changed, the finite state machine makes a state transition that keeps track of that change. This means we can use the state of the finite state machine to jump to specialised versions of each instruction for each combination of inputs. This is an untested approach that may improve the performance of our VM.

1.3 Project Scope

The project is to develop an implementation of a VM of this nature and run predetermined benchmarks on it against an alternate version of it that does not make use of type-mapping. This experiment should show what benefits the approach has. The The VM will only have around 35 instructions and 3-4 types. There will be no compiler implementation required and the VM will not be required to perform garbage collection. The VM will also not have to be robust in how it interacts with the operating system it is running on nor in how it handles errors.

1.4 Risks

The risks of the project are mainly about its structure and how it can be criticised. The VM will be running predefined benchmarks so there if it performs those well at the expense of its general usefulness the project can be criticised. All of these sorts of criticisms can be mitigated so long as the goal of the project is kept in mind at all times. Learning as much as possible about how well the idea works should always take precedence over trying to make results look good. Whether the VM performs better or not is important knowledge and being as careful as possible about discovering that accurately is extremely important.

1.5 Overview of Treatise

To be completed

2 Literature Review and Existing systems

References

[1] Ravi Nair James E. Smith. *Virtual Machines*. Elsevier Inc., 2005.

[2] MSDN.

[3] Oracle.

[4] Waldemar Celes1 Roberto Ierusalimschy, Luiz Henrique de Figueiredo. The implementation of lua 5.0.