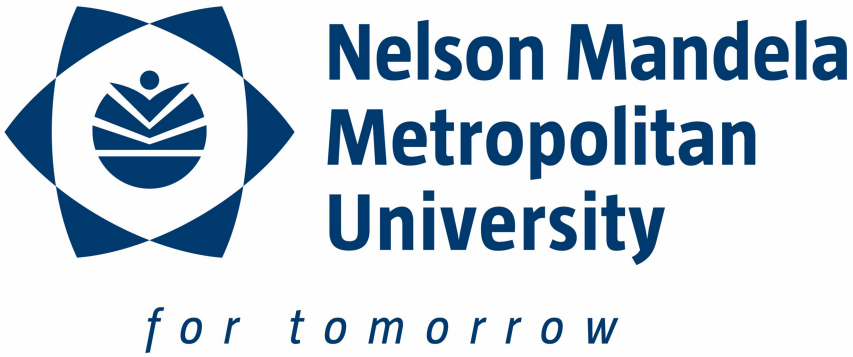
A JIT-Less Type-Mapped Dynamic-ISA Virtual Machine for Many-Instance Applications

August 2, 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

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

List of Figures

List of Tables

Table of 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" [?, 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 [?, 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"[?, 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[?, 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" "[?, 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 "[?, 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 [?, 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 define 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 [?, ?]. 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 VM will only have around 35 instructions and 2 types: integer and pointer. There will be no compiler or assembler 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. 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, Existing systems and Modern Processors

2.1 Modern Processor Architecture

In order to create an efficient VM, we need first understand how modern processors work and what some of the bottlenecks for our VM's execution might be. For example, *threading techniques* (not to be confused with threads in application programming) are commonly used in VM design. These take advantage of a feature of modern processors called the *Branch Target Buffer* or BTB. The BTB exists to aid a process called *branch prediction*. Branch prediction itself can only be explained once we know about how modern processors make use of *pipelining* to increase their throughput. As you can see, an understanding of modern computer architectures is needed before we can begin a discussion of VM design.

2.1.1 Pipelining

Figure 1:

Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

Pipelining is a process in which a processor's instruction processing cycle is shared by many consecutive instructions at once. The instruction processing procedure is split up into smaller stages that can execute simultaneously. An example of this is the classic RISC pipeline[x] which to divides instruction processing into the following steps: Instruction Fetch (IF), Decode (ID), Execute(EX), Memory Access (MEM) and Writeback (WB). Each instruction passes a stage and leaves that part of the processor free to perform that stage on the next instruction. Thus many instructions (as many as there are stages) can be processed at once, instead of each instruction having to be completely processed before the processor is free to move onto the next instruction. The process is analogous to an asssembly line, where many cars can be assembled at the same time, each in a different stage of assembly.

In *fig1* we see that the first instruction as currently performing MEMory access while the second EXecutes and the third is being Decoded. Up to 5 instructions can be processed simultaneously.

When pipelining an interesting situation occurs in the case of flow control instructions. These are instructions that cause execution to move to some other point in the program. The point at which we know which instruction comes next (called *branch resolution*) is usually later in the pipeline. Thus the processor cannot queue up the next instruction as it does not know which instruction will execute next. Branch prediction is used in modern processors to try to keep the pipeline as full as possible and not waste time waiting for this information to be known.

2.1.2 Branch Prediction

Instead of filling up the pipeline with no operation instructions until we know where to branch, with branch prediction we guess which branch will be taken and place the instructions from the predicted branch into the pipeline. When we know where the branch operation should have taken execution we either throw out our newly pipelined instructions (this is called *pipeline flushing*) if the prediction was incorrect or continue execution if it was correct. The more stages the pipeline has before branch resolution, the more of a performance benefit correct predictions become for programs with many control flow instructions. A Virtual Machine is such a program. This is because a VM must branch to the correct code for each instruction it executes. Writing code that allows for increased branch prediction accuracy is thus very important for VM efficiency.

Predicting a branch means we must predict if that branch is taken or not and what the target of that branch is if it is taken. Modern processors have a Branch Target Buffer where the target adresses of previously taken branches are cached. So upon arrival at a branch that has been previously taken, we guess if it will be taken and if it is we begin speculatively executing from the address stored in the BTB.

2.1.3 Cache

Modern processors make use of different levels of cache to allow for faster memory access. Ideally we would like to have an infinite amount of memory with no time cost for accessing it. The reality is that the larger memory is, the slower it becomes to access it [???]. In order to get closer to the ideal modern processors make use of cache. It takes around 100 cycles for Intel's Intel i7-4770 (Haswell)

architecture to access memory. Cache allows us get get closer to the ideal case by keeping commonly accessed memory closer to the CPU in levels of increasingly smaller, faster and more expensive (to manufacture) memory. The Haswell has 32KB of L1 data cache and 32KB of L1 code cache. These are located on the CPU itself [x] and can be accessed in around 4 cycles[x]. It also has 256KB of L2 cache and 8 MB of L3 cache. Caching works on the principal of locality of access. If you access memory in more or less the same area a lot, that acess is much faster than if you hop around from place to place. We should try to take advantage of cache in our VM's design.

2.2 Traditional Implementation of High Level VMs

2.2.1 Dispatch and Threading

Dispatch is the process of fetching, decoding and starting the next instruction to be run by a virtual machine [x].

```
typedef enum { add /* ... */ } Inst;

void engine()
{
        static Inst program[] = { add /* ... */ };
        Inst *ip = program;
        int *sp;

        for (;;)
                switch (*ip++)
                {
                case add:
                        sp[1]=sp[0]+sp[1];
                        sp++;
                break;
                /* ... */
        }
}
```

This approach is problematic because the switch statement means that there is only ever a single entry in the Branch Target Buffer used for dispatch . This means that the number of mispredictions will be large as the target of the branch will change for each new instruction we branch to. This problem may even be magnified by the fact that this specific VM implementation has far more instructions than usual as it requires several specialised versions of every instruction.

```
typedef void *Inst;

void engine()
{
        static Inst program[] = { &&add /* ... */ };
        Inst *ip = program;
        int *sp;

        goto *ip++; /* dispatch first VM inst. */

add:
        sp[1]=sp[0]+sp[1];
        sp++;
        goto *ip++; /* dispatch next VM inst. */

}
```

```
typedef void *Inst;

void engine()
{
        static Inst lookup[] = { &&add, &&sub /*...*/ }
        static int program[] = { 0, /* ... */ };
        int *ip = program;
        int *sp;

        goto *ip++; /* dispatch first VM inst. */

add:
        sp[1]=sp[0]+sp[1];
        sp++;
        goto lookup[*ip++]; /* dispatch next VM inst. */
}
```

Direct and inderect threading make much better use of the branch target buffer. Instead of a single entry in the BTB, with direct threading we have an entry per instruction, so instructions that commonly follow each other have a better chance of being predicted. All code in this section is from or adapted from [**?**].

2.2.2 Registers VS Stacks

In a stack virtual machine, instructions act on members of the stack. Arguments and return values for instructions are often implicit and thus instructions can be smaller. For instance a stack implementation of a = b + c would first push b and c onto the stack, then call the add instruction which has no arguments. Add pops b and c off the stack and pushes b+c back onto the stack. This value is then popped off the stack and stored.

For a register machine, a similar piece of code would have values for b and c in registers and an add instruction is called with a, b and c as arguments. This instruction would compute b+c and store it in a regsiter.

Widely used process virtual machines make use of a stack architecture. Both the Java Virtual Machine (JVM) and Microsoft's Common Language Runtime (CLR) make use of stack virtual machines[x].

2.2.3 JIT Compilation

Both the JVM and CLR make use of JIT compilation [x]. JIT compilation allows for code that is executed often enough to be compiled into native machine code at runtime. JIT compilation does not preclude the bytecode of a many-instance application being shared but each instance of the application is JIT compiling that code. So it is likely that the same bytecode is being compiled in many different processes at the same time.

2.3 VM Interpreter Reseach

James R. Bell introduced the concept of threaded code in 1973. The Fortran IV compiler was written to generate threaded code.[?]

Yuhne Shi[?] found that a well-implemented register VM is a more efficient option when speed of execution is more important than the size of the code to be executed.

Ertl and Gregg found that in their benchmarks 3.2%–13% of all executed instructions were indirect branches and that switch dispatch on an architecture with a BRB resulted in 81%-98% branch prediction misses [?]. They also found that direct threading resulted in 50%-63% branch prediction misses.

2.4 Our Implementation

2.4.1 Virtual Machine Design Details

The virtual machine described in this treatise is a register machine. It has 6 general purpose registers and 3 special purpose registers. The VM is for a dynamically typed language. Each register stores 'values' and all instructions act on those values. The type of a value is stored together with the data for that value. These are implemented in C as tagged unions. The VM has only two types: integer and pointer.

The 3 special purpose registers are:

1. The program counter (pc), which keeps track of where we are in the program
2. The frame pointer (fp), which points to the bottom of the latest stack frame
3. The type state (ts), which keeps track of the current state of types in the registers

2.4.2 Type-Mapping

The virtual machine does not make use of the usual opcode and arguments arrangment used in register machine interpreters. Usually, the interpreter reads in some instruction stored as an opcode and the arguments for that opcode at the same time. In *fig 2* we see a 16 bit instruction composed of a 4 bit opcode in the high bits, followed by two 6 bit operands or aguments. The interpreter performs the necessary shifts and bitmasking to get the values out of the fields.

In our VM instructions for programs are stored as opcodes only with no arguments. That is because we have a version of each instruction for every combination of registers as inputs. Because we have 6 registers and a maximum of two arguments per instruction that means that we can have a maximum of 6^2 versions of each instruction. This saves us the process of extracting arguments at the expense having to have more versions of the instruction[why are we doing it this way actually?]. The opcode implicitly stores the argument information.

The opcode is not the only information the VM uses to select code, however. The *ts* register keeps a record of the type of every register. It is a 6 bit number where each bit tells us the current type of a register. Thus for every opcode, there are 2^6 different states the VM could be in. With this information we know the arguments and

types and can jump to specialised code that acts on arguments of those types. A good way to think about it is to imagine it as a 2D lookup table (even though the implementation is 1D). On the y-axis we have the opcodes for all the versions of the instructions and on the x-axis we have the current state from 0 to 63. At the intersection of these we store the address of the code we jump to for that instruction.

Now since we only have integers and pointers it may, at first glace, seem pointless to have specialised instructions since most of the time using pointers in an instruction meant for integers is illegal. However this method elliminates the type checking involved in instructions for a dynamic-ISA VM. Normally for each instruction we would have to check the validity of the arguments first before we can perform the instruction, but with this method we already know the types and so we can just jump directly to code for those types or an error if the arguments are illegal.

2.4.3 Indirect Threading

This VM will make use of indirect threading. Though indirect threading is slower than direct threading [x] because it must first complete the lookup step before it can branch to the next instruction, the program code used by the VM is intended to be shared by many instances of the VM. This cannot be achieved with direct threading as the addresses of each instruction in each instance may be different[x]. Also because our dispatch is based on the runtime state of the virtual machine (the st register) even if addresses were the same between instnces we can't represent code in terms of those addresses as we don't know the state information until runtime so we can't choose which address should be used to replace the opcode. With Indirect threading's lookup table we can share the code in its opcode form and perform the correct jumps for each instance of the VM by consulting the state register and looking up the addresses in the lookup table.

3 Solution Design

Three virtual machines are presented as solutions. These VMs are spaced along a scale for how much they expand out the behaviour of each instruction to match the state of the virtual machine. At the lowest end of the scale is the control virtual machine which shall be referred to as the Conventional VM. One step up from that is a VM that expands out all cases of arguments for each instruction. This will be referred to as the Hybrid VM. A further step takes us to the Type-State virtual machine which, like the hybrid VM, expands out all cases of arguments for each instruction, but then further expands

out for every case of the combinations of types those arguments could take on.

On the lowest end of this scale the VM is simpler to implement and uses fewer instructions, but each instruction is less specific and so the complexity lies within the instructions. This is the conventional design for this type of VM.

On the highest end of the scale the VM is more difficult to implement and uses more instructions, but each instruction is very specifically geared to exact state of the VM so each instruction is very simple.

3.1 Common Elements

The three virtual machines are designed to be as similar as possible except for the aspects that the experiment examines. Those elements common to all VMs shall be expanded upon here and those specific to each shall be examined seperately.

3.1.1 Values

```
typedef struct ValueStruct value;
struct ValueStruct
{
        int tag;
        union
        {
                int64_t i;
                void *p;
        };
};
```

As this is a dynamic ISA virtual machine, types are associated with values instead of variables [lua VM]. The implementation for a value is simple enough to include here

A value represents one of four types depending on the tag. The value is either an integer (tag = 0), a pointer to a value or NULL (tag = 1), a pointer to an object (tag = 2) or a pointer to a buffer (tag = 4).

3.1.2 Objects

Objects are a simple built-in data structure for storing ordered fields of values. Objects are implemented as a struct with an array of values and an integer field for storing the length of the array.

3.1.3 Buffers

Buffers are a data structure that exposes the native byte to the VM so that string handling may be performed without having to create a value for each character. The implementation is, once again a struct with an array of bytes and a size field that holds the length of the array.

3.1.4 Registers

All virtual machines make use of 6 general purpose registers. These registers are implemented as an array of 6 values. A further two registers are found in all VMs, that is the program counter, which will be referred to as PC and the frame pointer which will be referred to as FP. These are a 64 bit integer and a value pointer respectively.

3.1.5 Stack Frames

All VMs make use of identical Stack Frames for subroutine calls. Each stack frame stores:

- The previous FP, PC and an additional register that goes unused except in the Type-State VM.
- The general purpose registers except the first and last
- The local variables used in that stack frame

Registers are saved upon calling a subroutine and restored when the subroutine returns. The first and last general purpose registers are used for return values.

3.1.6 Addressing

3.1.7 Instructions

All virtual machines support an identical set of instructions. How these are implemented is quite different for each VM but the functionality of each instruction does not change between virtual machines.

Instruction words are 16 bits long. Each instruction may have a maximum of two operands. If the instruction has multiple operands and a result, the result will be stored in the first of the two operands.

Instruction operands refer to one of the 6 general purpose registers. In some cases the next instruction word is a 16 bit constant value that is used by the instruction and in others the next instruction word is used as a 16 bit PC-relative address for a 64 bit constant value. These will be denoted as const16 and const64 respectively and their implementation will be expanded upon later.

In some cases, specific argument combinations for an instruction have been removed as they produce trivial results. For instance the subtract operation cannot be used with the same register for both arguments. Since the result of this operation is always zero and there already is an instruction to set a register to zero, there is no need to include this case. This has little bearing on the conventional VM (which can do these operations) but plays a role in the other two VMs.

The set of instructions is as follows:

Arithmetic Instructions

Arithmetic instructions provide some basic arithmetic operations for registers that contain integers. If a register that does not contain an integer is selected, an error is signalled and the VM halts. All arithmetic instructions increment the program counter. Those that make use of a constant that is looked up in the next instruction word increment the program counter again so that the PC points to the next instruction and not the data contained in the next word. The PC is incremented after the operation performed by the instruction.

$add(g_i, g_j)$	$g_i \longleftarrow g_i + g_j$
$addc(g_i, const_{64})$	$g_i \longleftarrow g_i + const_{64}$
$sub(g_i, g_{j \neq i})$	$g_i \longleftarrow g_i - g_j$
$csub(const_{64}, g_i)$	$g_i \longleftarrow const_{64} - g_i$
$mul(g_i, g_j)$	$g_i \longleftarrow g_i \times g_j$
$mulc(g_i, const_{64})$	$g_i \longleftarrow g_i \times const_{64}$
$div(g_i, g_{j \neq i})$	$g_i \longleftarrow g_i \div g_j; g_0 \longleftarrow g_i \textbf{ mod } g_j$
$divc(g_i, const_{64})$	$g_i \longleftarrow g_i \div const_{64}; g_0 \longleftarrow g_i \textbf{ mod } const_{64}$

The functioning of these operations is obvious except for the two division instructions where the first argument is g0. In this case g0 mod gj (or const64 as the case may be) is stored in g0 as the final result. The implementation used to bring this about is:

- find the modulus first and store it in a temporary variable

- perform the division, storing the result in register gi

- store the temporary variable in gi

Bit Manipulation Instructions

$and(g_i, g_{j \neq i})$	$g_0 \longleftarrow g_i$ **and** g_j
$andc(g_i, const_{64})$	$g_0 \longleftarrow g_i$ **and** $const_{64}$
$or(g_i, g_{j \neq i})$	$g_0 \longleftarrow g_i$ **or** g_j
$orc(g_i, const_{64})$	$g_0 \longleftarrow g_i$ **or** $const_{64}$
$xor(g_i, g_{j \neq i})$	$g_0 \longleftarrow g_i$ **xor** g_j
$shl(g_i, g_j)$	$g_0 \longleftarrow g_i << g_j$
$shlc(g_i, const_{16})$	$g_0 \longleftarrow g_i << const_{16}$
$cshl(const_{64}, g_i)$	$g_0 \longleftarrow const_{64} << g_i$
$shr(g_i, g_j)$	$g_0 \longleftarrow g_i >> g_j$
$shrc(g_i, const_{16})$	$g_0 \longleftarrow g_i >> const_{16}$
$cshr(const_{64}, g_i)$	$g_0 \longleftarrow const_{64} >> g_i$
$sar(g_i, g_j)$	$g_0 \longleftarrow g_i >>> const_{16}$
$sarc(g_i, const_{16})$	$g_0 \longleftarrow g_i >>> const_{16}$
$csar(const_{64}, g_i)$	$g_0 \longleftarrow const_{64} >>> g_i$

These instructions are for performing bitwise logical operations on integer operands and constants. The PC is incremented in the same manner as for the arithmetic instructions. The shr and shl instructions are the logical shifts whereas sar is the arithmetic right shift. The arithmetic left shift is the same operation as shl.

Data Movement Instructions

These instructions allow for values to be moved between registers.

$mov(g_i, g_{j \neq i})$	$g_i \longleftarrow g_j$
$movc(g_i, const_{64})$	$g_i \longleftarrow const_{64}$
$null(g_i)$	$g_i \longleftarrow NULL$

Memory Access

$getl(g_i, const_{16})$	$g_i \longleftarrow mem[fp + const_{16}]$
$setl(const_{16}, g_i)$	$mem[fp + const_{16}] \longleftarrow g_i$
$geto(g_i, g_j, g_{k \neq j})$	$g_i \longleftarrow mem[g_j + header + g_k * scale]$
$seto(g_i : g_j, g_{k \neq j})$	a
getb	a
setb	a

These instructions allow for getting and setting of locals (getl, setl), objects (geto, seto) and buffers (getb, setb).

Control Flow Instructions

jmp	a
jmpf	a
swtch	a
jcmp	a
jcmpc	a
jnullp	a

Function Call and Return

call	a
ret	a

Pragmatic Higher-Level Instructions

newo	a
newb	a
err	a
in	a
out	a
print	a

3.2 Conventional VM

3.2.1 opcode

The