Stacking them up: a Comparison of Virtual Machines

K John Gough *
j.gough@qut.edu.au

Abstract

A popular trend in current software technology is to gain program portability by compiling programs to an intermediate form based on an abstract machine definition. Such approaches date back at least to the 1970s, but have achieved new impetus based on the current popularity of the programming language Java. Implementations of language Java compile programs to bytecodes understood by the Java Virtual Machine (JVM). More recently Microsoft have released preliminary details of their ".NET" platform, which is based on an abstract machine superficially similar to the JVM. In each case program execution is normally mediated by a just in time compiler (JIT), although in principle interpretative execution is also possible.

Although these two competing technologies share some common aims the objectives of the virtual machine designs are significantly different. In particular, the ease with which embedded systems might use small-footprint versions of these virtual machines depends on detailed properties of the machine definitions.

In this study, a compiler was implemented which can produce output code that may be run on either the JVM or .NET platforms. The compiler is available in the public domain, and facilitates comparisons to be made both at compile time and at runtime.

1 Introduction

1.1 Abstract Stack Machines

The idea of using an intermediate form within a programming language compiler, as a means of communication between the front-end and back-end, dates back at least to the 1970s. The idea is quite straightforward. Language dependent front-ends compile and semantically check programs, passing an intermediate language representation of the program to the code-generating backend. In an ideal situation the frontend would be entirely independent of the target hardware, while the backend would be sensibly independent of the particular language in which the source program was written. In this way the task of writing compilers for N languages on M machine architectures is factored into $N + M$ part-compilers rather than $N \times M$ complete compilers.

Many of these intermediate language representation were based on abstract stack machines. One particular representation, *P-Code*, was invented as an intermediate form for the ETH Pascal Compilers[1], but became pervasive as the machine code for the UCSD Pascal System. What had been noted by the UCSD people was that a program encoded for an abstract stack machine may be used in two ways: a compiler backend may compile the code down to the machine language of the actual target machine, or an interpreter may be written which *emulates* the abstract machine on the target. This interpretative approach surrenders a significant factor of speed, but has the advantage that programs are much more dense in the abstract machine encoding. In the case of UCSD Pascal the code was so compact that the compilers could be run on the 4k or so of memory available on the very first microcomputers. As a consequence of this technology high-level languages became available for the first time on microcomputers. As an added benefit, the task of porting a language system to a new machine reduced to the relatively simple task of creating a new interpreter on the new machine.

The use of abstract machines as compiler intermediate forms has also had its adherents. For example, the Gardens Point compilers all use a stack intermediate form (D-Code) for all of the languages and platforms supported by the system[2]. Although most implementations are fully compiled, a special lightweight interpreted version of the system was written in about 1990 for the Intel *iapx86* architecture, allowing users with a humble IBM XT to produce the same results as the 32-bit UNIX platforms that the other implementations supported[3]. As a measure of the complexity of the virtual machine emulator, the interpreter was about 1k lines of assembly language, with the floating point emulator a further 1k lines.

A largely failed attempt to leverage the portability properties of stack intermediate forms was the Open Soft-

* Queensland University of Technology, Box 2434 Brisbane 4001, Australia

ware Foundation's *Achitecture Neutral Distribution Form* (*ANDF*). The idea behind ANDF was to distribute programs in an intermediate form, and complete the task of compilation during an *installation* step. The ANDF form was code for an abstract stack machine, but one with a slight twist. Generators of intermediate forms such as D-Code know enough about the target's addressing constraints to be able to resolve (say) record field accesses to address offsets. In the case of ANDF the target is not yet determined at the time of compilation, so that all such accesses must remain symbolic. It has been suggested that this incorporation of symbolic information into the distributed form was considered to be a threat to intellectual property rights by the major software companies, and was a factor in the failure of the form to achieve widespread adoption.

In the late 1990s Sun Microsystems released their Java[4] language system. This system is, once again, based on an abstract stack machine. And again, like ANDF, relies on the presence of symbolic information to allow such things as field offsets to be resolved at deployment time. In the case of Java and the Java Virtual Machine[5] (*JVM*) the "problem" of symbolic content turned out to be a virtue. The presence of the symbolic information is the thing which allows deployment-time and runtime enforcement of the type system via the so-called *bytecode verifier*. These runtime type safety guarantees are the basis on which applet security is founded. As things now stand, *JVM*s are available for almost all computing platforms, and Java tells a program portability story which transends almost all other vehicles.

In mid-2000 Microsoft revealed a new technology based on a wider use of the world wide web for service delivery. This technology became known as the *.NET* system. The technology has many components, but all of it depends on a runtime which is object-oriented and fully garbage collected. The runtime processes an intermediate form which, like the *JVM*, is based on an abstract stack machine. Apart from this common structure, the detailed design of the two machines is quite different.

During 1999 the author had explored the applicability of the *JVM* as a target for languages other than Java. As a result of this a prototype compiler for the language Component Pascal[6] was written. This compiler translates Component Pascal programs into *JVM* bytecodes. The prototype was written in Java. During the first half of 2000 Paul Roe and the author were given the opportunity to work under NDA on the then un-announced *.NET* platform. Building on the experience of the prototype, an entirely new Component Pascal compiler was written, this time implemented in Component Pascal. The new compiler has two separate code emitters. One produces *JVM* byte-codes, while the other produces *.NET* intermediate language. The compiler may be bootstrapped on either platform. The existence of these two parallel code generators allows side by side comparisons to be made between the two platforms.

The remainder of this paper is organised as follows: Section 2 gives a brief overview of the Java Virtual Machine, while Section 3 gives an overview of the *.NET* execution engine. Section 4 discusses the detailed differences between the two abstract machines, and introduces some performance comparisions. Finally, Section 5 draws some conclusions and offers some tentative predictions of future directions.

2 The Java Virtual Machine

The underlying execution mechanism of the *JVM* is an evaluation stack, and a set of instruction which manipulate this stack. As a first example, the code required to take two local integer variables, add them together and deposit the result in a third local variable would be –

```
iload_1    ;  push local int variable 1
iload_2    ;  push local int variable 2
iadd       ;  add the two top elements
istore_3   ;  pop result into variable 3
```

Note the use of the i-prefix on all of these instructions, encoding the fact that these all operate on integers. Notice also that in this case the index of the local variable is encoded into the instruction, at least for the lowest numbered few variables. We might therefore expect the code to be quite dense with each of the instructions requiring only one byte.

The instruction set of the *JVM* is designed with the sole purpose of representing Java programs. There is thus direct support for the object model of Java, and for the various kinds of method dispatch that are required. In particular, the instruction set allows for classes to inherit behaviour from just one superclass, but to declare that they implement multiple fully abstract class specifications (i.e. "interfaces").

At runtime, data is represented in just two ways. Scalar data may exist as local variables, in fields of structures, or on the evaluation stack of the abstract machine. Aggregate data exists only in dynamically allocated objects which are automatically collected when they are no longer accessible. References to these objects may be stored in local variables, in fields or on the evaluation stack as with other scalars. There is no union construct.

During the execution of a method, the evaluation stack consists of a finite stack of "slots" the depth of which is statically determined by the compiler. Each of these slots may contain an object reference or a 32-bit scalar value. Long integers and floating point double values use up two slots.

2.1 Class Files

At deployment time a Java program is represented by a set of one or more dynamically loaded *class files*. These files contain a specification of the behaviour of the class, including its external contracts. The features of the class are named in an indexed *constant pool*, and all references to these features are mediated via references to the constant pool indices.

This symbolic information allows for a significant degree of type-checking to take place at load time, with a small amount of runtime checking still required. This, together with the absence of instructions which manipulate addresses ensures that programs encoded for the *JVM* will be free from certain kinds of type errors at runtime. This is a necessary foundation for the kind of security which users require before executing code from untrusted sources.

As it turns out, the presence of symbolic information increases the code density of programs so that typically they are comparable in size with native code object files for complete programs. Although each instruction requires only one byte, or one byte plus a constant pool index, the constant pool itself takes up a significant amount of space. Of course, class files tend to be textually repetitious, so that they compress quite readily for data transport.

2.2 Parameter Passing

There are four different method invocation instructions, for static methods, virtual methods, interface methods, and virtual methods invoked statically. In each case the method may take any number of parameters, passed by value. In all but the static case, the methods take a **this** receiver, which appears as the zero-th parameter to the callee. Methods may return a single result.

In all cases, actual parameter values are pushed onto the evaluation stack prior to the call, and a returned result appears on the top of the evaluation stack on the return. Incoming values appear as the first n local variables in the callee.

Since only scalar values and references may be pushed on the stack, it follows that these are the only possible parameter types. However, since both arrays and structures only exist as dynamically allocated objects accessed by reference, this is no limitation for Java programs.

As has been noted elsewhere[7, 8] the semantics of parameter passing in the *JVM* create a limitation in the efficiency with which languages other than Java can be implemented on this machine.

3 The *.NET* Execution Engine

The underlying execution mechanism of *.NET* is an evaluation stack, and a set of instruction which manipulate this stack. To take the same example, the code required to take two local integer variables, add them and deposit the result in a third local variable would be –

```
ldloc.1   ;  push local variable 1
ldloc.2   ;  push local variable 2
add       ;  add the two top elements
stloc.3   ;  pop result into variable 3
```

These instructions are all generic, with the type of the “add” being determined from the inferred type of the stack contents. In this case the type will be known from the declared types of the local variables. Indeed, the instruction sequence would be identical if the three variables were all declared as floating point double type. Notice that for this platform also, the index of the local variable is encoded into the instruction, at least for the lowest numbered variables.

The instruction set of *.NET* is designed with the objective of supporting multiple languages, and thus needs to support all of the constructs of what Microsoft calls the *Virtual Object System* (*VOS*). The object model supports several kinds of method dispatch. There are three kinds of methods: static methods and instance methods which may be either virtual or non-virtual. As with Java, *VOS* reference classes are permitted to inherit behaviour from just one superclass, but may declare that they implement multiple fully abstract class specifications (i.e. “interfaces”).

At runtime data exists as scalars, as references, and as instances of value classes. There is a fundametal distinction made between value and reference classes. Value classes do not inherit behaviour, and cannot have virtual methods. As the name implies assignment of such values has value (non-aliassing) semantics. In the newly announced language C#, `structs` are implemented as value classes, while `classes` are implemented as reference classes in *VOS*. Value class instances may be statically allocated, automatically allocated at method entry, or be boxed in dynamically allocated objects. The instruction set has support for boxing and unboxing of such values. Dynamically allocated objects are garbage collected when no longer accessible. As with the *JVM* there is no union construct.

During the execution of a method, the evaluation stack consists of a finite stack of abstract values. The depth of the stack is statically determined by the compiler. Each abstract stack element may contain any value, including an instance of a value class. Unlike the *JVM*, no value uses up multiple elements.

3.1 Assembly Files

At deployment time a *.NET* program is represented by a set of one or more dynamically loaded *assembly files*.

These files typically contain a specification for a single module, which may contain a large number of separate classes. There is provision for versioning information and cryptographically strong signatures for such assemblies. As with the *JVM* the assembly contains a significant amount of symbolic information.

This symbolic information allows for a significant degree of type-checking to take place at load time, with a small amount of runtime checking still required. This, together with the absence of instructions which manipulate addresses ensures that programs encoded for *.NET* will be free from the same kinds of type errors that are absent in programs executed on the *JVM*.

The code density of executable programs in *.NET* is comparable with the same program encoded for the *JVM*.

3.2 Parameter Passing

There are only two different method invocation instructions in *.NET*. One, `callvirt`, performs a virtual dispatch, and is used for both virtual methods and interface implementation methods. The other, plain `call`, is used for static calls and also for non-virtual dispatch of instance methods. In each case the execution engine must determine the semantics of the invocation from the signature of the callee.

Methods may take any number of parameters, which may be passed by value or by reference. Reference parameters may be marked as “out”, with significance for the computation of value liveness. In all but the static case, methods take a **this** receiver, which appears as the zero-th parameter to the callee. Methods may return a single result.

Actual parameter values are pushed onto the evaluation stack prior to the call, and a returned result appears on the top of the evaluation stack on the return. Incoming values appear as a list of arguments distinct from local variables.

Because a wide range of values may be pushed onto the stack, it follows that a wide range of parameter passing semantics may be created. In particular `structs` may be passed by value, and formal values mutated without affecting the corresponding actual parameter value.

The wide range of parameter passing semantics is intended to support a range of languages. This goal appears to be well met. Note however that arrays values in *.NET* are always references, and that it is these array *references* that are able to be passed either by value or by reference. In order to obtain non-aliassing behaviour for array parameters with this platform an array copy must be explicitly encoded either in the caller or in the callee prolog.

4 Comparing the Virtual Machines

4.1 Overall Philosophy

There are some clear differences in the underlying philosophies of design for these two virtual machines. However, it might be well to first highlight the design choices that are in common.

The two designs both rely on a virtual machine specification which reveal an underlying object oriented model. In the case of the *JVM* this model is closely isomorphic to the Java language, while for *.NET* the model is the somewhat more general *VOS*. An alternative would have been to have a lower-level model which relied on explicit operations for computing target addresses for such things as virtual method dispatch. However, such a choice would give up the chance to perform load-time verification of type safety, and would thus miss out on one of the main attractions of the VM approach. In any case, this choice is in common, as is the single-inheritance, multiple interface implementation choice. In each case, as might be expected, the designs rely on garbage collection for memory management.

The most striking design difference is that the *.NET* machine designers seem to have been willing to surrender the option of interpretative execution. This choice is signalled by a number of the details. Perhaps the first hint is the presence in the instruction set of generic instructions such as `add` with no specified data type. In order to interpret such an instruction it would be necessary for the interpreter to track the data type of the top of stack element. This would appear to require more computation than the rest of the interpreter fetch execute cycle, thus extracting crippling performance penalties. Just to choose one other example, in both virtual machines the compiled code declares the number of local variables which each method uses. In the case of the *JVM* this is the maximum number of 32-bit slots that the method uses. The code may reuse slots either singly or in pairs for any data type. In the *.NET* machine, by contrast, local variables are logically distinct, and each has a declared data type. Variables may only be recycled for another datum of the same type. Any overlaying of data of different types in the same stack frame, if it happens at all, is performed by the *JIT*. Recall also that in *.NET* local variables may be entire structs, rather than being restricted to 32 and 64-bit data, as is the case in the *JVM*. As a consequence, in *.NET* the size of the stack frame is not directly specified in the bytecode.

By contrast, interpretation of *JVM* byte-codes was an explicit goal, and one that is still preferred in some contexts where the runtime of a program does not adequately amortise the high cost of *JIT*-compiling. The historical origins of the Java language lie in a project at Sun Microsystems which was specifically aimed at small footprint, embedded machines for network appliances.

In the conclusion, the possibility of interpreting a pre-processed version of *.NET* intermediate language is considered.

As might be expected, there are a number of detailed differences in the byte-code format which are of importance to the compiler writer, but do not indicate a difference in expressivity in the languages. As an example, consider the instructions for loading the integer field "`fld`" of the object whose reference is on the top of the stack. They are –

```
getfield   ClassName/fld I         // I means int32
ldfld      int32 ClassName::fld
```

Apart from the order of operands, there is a subtle difference: in the first example the class name is the class of the object on the stack. In the second case the class name is the name of the class from which the object *inherits* the field.

4.2 Code Density

Neither of the machines has a significant edge over the other in program density. As a rough spot-check, the two encodings of the Gardens Point Component Pascal compiler were compared. The compiler consists of 29k lines of code in 31 modules. The 117 class files of the *JVM* version occupy 550k bytes, while the 31 dynamically loaded libraries of the *.NET* version occupy 480k bytes.

4.3 Sharing out the Bit-budget

Both of these virtual machines have approximately 250 instructions defined, fitting with the concept of being "byte-codes". The way in which this budget is shared out among the various functions is somewhat different. In *JVM* there are multiple versions of arithmetic instructions, and four rather than two method invoke instructions. There is also a substantial number of codes which are used for "quick" versions of other instructions. The idea behind these instructions seems to have been an attempt to allow loaded class files to be overwritten once some of the more costly operations have been performed. This would allow subsequent executions to use the updated instructions and avoid repetition of the costly operation. The widespread adoption of *JIT*-compiler technology probably means that these codes are a wasteful legacy obligation.

In the *.NET* encoding there are some savings due to the generic operations, but this is lessened to some extent by the need for both trapping and non-trapping versions of the integer arithmetic operations. Furthermore, there are a number of extra instructions required to deal with reference parameters, and with the pushing and popping of instances of value classes. The need to use different instructions to access arguments and local variables also burns up some extra space here. As in the case of the "quick" instructions of the *JVM*, there is a trace of a false start in the design of the *.NET* instruction set. In the encoding there are a number of instructions which define annotations for the control flow graph. In principle this would have allowed front-ends to assist the *JIT* in constructing a static single assignment representation of each method. It now seems that the *JIT*s would rather rely on their own computations of such attributes, rather than trust a third party frontend processor. In the current release, these instructions are no-ops.

In each case, most of the leftover space in the budget of 256 instructions has been used up with specialised versions of the common instructions. Examples of this have been seen in Sections-2 and 3, where access to the low-numbered local variables use a single byte instruction, without the need for an index. Both machines use these for loading and storing locals, and for loading a number of commonly occurring literal values. Curiously, the *JVM* instruction set allows for at least two implicit literal loads for long and floating point constants (0 and 1, predictably), while the *.NET* platform uses the resource for four-byte integers only. It is not clear whether this decision is based on careful simulation or a wet finger in the wind.

4.4 Multi-language Support

The *.NET* platform is explicitly intended to support multiple languages, and at the initial announcement of the platform a variety of language implementations were demonstrated by both research and commercial language groups. These languages included contemporary logic, functional and object-oriented languages. Indeed, Fujitsu demonstrated an implementation of Object Oriented COBOL interworking with Microsoft's visual studio languages.

The case with the *JVM* is less clear, since multi-language support was never a goal. There are a number of implementations of at least subsets of other languages on the *JVM*, but these need to overcome a number of difficulties[7, 8]. In particular, the absence of reference parameters leads to a number of contortions which will be familiar to anyone who has had to translate standard algorithms to Java. The boxing and unboxing of values required to simulate reference parameters are strikingly demonstrated in the Hennessy benchmarks in Figure 1

These times are for the same program compiled with Gardens Point Component Pascal, but using either the *.NET* or the *JVM* code emitter. These are thus the same programs compiled with the same compiler and run on the same machine with the two different virtual machines. This Figure gives the results for a number of classic but idealised algorithms. It will be noted that for some of these non-object oriented codes the *JVM* platform is faster than *.NET*. Notice however that for the first program, `perm`, the *JVM* is worse by a factor more than ten. This overhead arises in a single

Benchmark	gpcp/NET	gpcp/JVM
perm	2.8	38.1
towers	4.2	4.7
queens	2.7	2.6
intMM	5.0	10.2
puzzle	15.3	13.7
quick	3.3	3.5
bubble	4.7	2.7
trees	4.2	5.3

Figure 1. Time in seconds for the Hennessy integer benchmarks

call site in the program, where a `swap()` procedure swaps the values of two reference parameters. The need to box and unbox one of these (the other secretly sneaks back as the function result) causes the observed blowout. (It might also be added that this example gives yet another powerful argument against the use of toy benchmarks in which a single statement can dominate the entire runtime.)

It should be noted that these figures compare a relatively well-tuned *JVM* code emitter running on a second-generation virtual machine with a prototype *.NET* code generator running on a pre-beta VM. We would predict a closing of the gap between these figures as **gpcp** climbs the *.NET* learning curve.

There are some difficulties with some language constructs, even for the *.NET* platform. For example, methods with covariant return types are not directly supported by the *VOS*, and require some encoding magic to emulate. This is even more the case for some of the more exotic languages, although an experimental implementation of most of the Eiffel language showed that generic code can be supported, with some effort.

5 Conclusions

The comparison of these two virtual machines shows clear evidence of quite different original design goals. In the case of the *JVM* the evolution of the whole Java movement has taken the platform well beyond the original design context. All of this has taken place in the full glare of public scrutiny, and in a context where backward compatability at the VM level has almost certainly become a costly constraint.

By contrast, the designers of the *.NET* VM have had the luxury of a postponed decision on the freezing of the instruction set. Presumably they have also learned from the Java experience, insofar as the design goals overlap.

For the *JVM* the most important evolutionary decision will be whether or not the machine can be extended so as to efficiently implement some of the desirable language extensions. These include parametric polyporphism, and direct support for local or nested classes. (At the moment the *JVM* does not provide support for nested classes, and instead depends on a particularly ugly name-mangling convention.) Parametric polymorphism (or generics) is a likely extension to the Java language, and could be implemented without VM changes. However, there are significant advantages in changing the VM to directly support this. Perhaps we may see the disappearance of the “quick” instructions in favour of some of this support for advanced language features.

In the case of *.NET* similar considerations apply, at least for generics. However a more pertinent consideration may be the issue of small footprint embedded versions. As has been argued above, the *.NET* intermediate language does not lend itself to interpretation. However, it would be possible to perform an offline pre-processing step which converts to a different form which is more interpreter-friendly. Particularly if the conversion accepts responsibility for the type safety issues, the form could be trimmed of its symbolic information, and achieve much higher code density. This would be perfectly acceptable for real “embedded” systems which did not have to resolve the issues of dynamic loading of modules, and runtime type safety and security model checks. There is actually a double gain to be had along this path. Not only is there a significant opportunity to increase code density, but the footprint of the VM itself can be substantially shrunk. The figures quoted for the interpreted version of **gpm**[3] give a reasonable limit for just how small a general purpose VM can be.

It may be noted that some lightweight implementations of the *JVM* involve such a philosophy, although in that case the option of dynamic loading is retained and thus the opportunity to trim all of the symbolic information is not taken.

The project to create Gardens Point Component Pascal has been an interesting one. It has enabled us to make a very detailed side-by-side comparison of the two VMs which will fight it out for market share on the world’s desktops. The compiler itself is open source software, under the Free Software Foundation’s General Public Licence. We expect to use the compiler as a platform for further language research, with the implementation of parametric polymorphism the first goal in the second half of year-2000.

6 Acknowledgment

Parts of this project were supported by ARC Grant A49700626. Continuing work on the *.NET* platform is supported by Microsoft, and by the CRC for Enterprise Distributed Systems Technology.

References

[1] U Ammann, 'Code Generation for a Pascal Compiler', chapter in *Pascal the Language and its Implementation*, Editor: D W Barron, Wiley 1981.

[2] K John Gough, 'Multi-language, Multi-target Compiler Development: Evolution of the Gardens Point Compiler Project' *Joint Modula Languages Conference* JMLC1997, Linz, March 1997, LNCS 1204, Springer.

[3] K J Gough, C Cifuentes, D Corney, J Hynd and P Kolb. 'An Experiment in Mixed Compilation/Interpretation' *Proceedings of the Australian Computer Science Conference* ACSC-14, Hobart, Australia, 1992. Australian Computer Society.

[4] J. Gosling, B. Joy and G. Steele, *The Java Language Specification*, Addison-Wesley, Reading MA, 1997.

[5] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, Addison-Wesley, Reading MA, 1997.

[6] Oberon Microsystems, 'Component Pascal Language Report' available at — `http://www.oberon.ch/resources`

[7] K John Gough, 'Parameter Passing for the Java Virtual Machine' *Australian Computer Science Conference* ACSC2000, Canberra, February 2000, IEEE Press.

[8] K John Gough and Diane Corney, 'Evaluating the Java Virtual Machine as a target for Languages Other than Java' *Joint Modula Languages Conference* JMLC2000, Zurich, September 2000, (to appear in LNCS, Springer).