JVM versus CLR: A Comparative Study

Jeremy Singer
University of Cambridge Computer Laboratory
William Gates Building, 15 JJ Thomson Avenue
Cambridge, CB3 0FD, UK

jeremy.singer@cl.cam.ac.uk

ABSTRACT

We present empirical evidence to demonstrate that there is little or no difference between the Java Virtual Machine and the .NET Common Language Runtime, as regards the compilation and execution of object-oriented programs. Then we give details of a case study that proves the superiority of the Common Language Runtime as a target for imperative programming language compilers (in particular GCC).

1. INTRODUCTION

Sun's Java Virtual Machine (JVM) and Microsoft's .NET Common Language Runtime (CLR) have recently emerged as fierce rivals for the accolade of the premier next-generation computing platform.

We say little in qualitative terms about the overall relative merits of the two machines or their designers' philosophies. Those matters have already been considered elsewhere [8]. Instead, we make a quantitative comparison of the performance of the JVM with the CLR, using a standard Java benchmark test. Performance is defined in terms of code size and code speed, since both factors are important for would-be ubiquitous distributed computing platforms. From this general comparison, we see how the two virtual machines are evenly matched, when executing programs written in idiomatic modern object-oriented style.

Then we move on to address the issue of how well the two virtual machines serve as targets for the imperative programming style. We consider the problem in both qualitative and quantitative terms, by applying it to code generation in the GCC compiler.

We note in passing that Gough [8, 7] has undertaken a similar investigation in the context of his Component Pascal compiler.

2. JAVA VIRTUAL MACHINE

Java [1] is a simple general-purpose object-oriented architecture-neutral programming language. Java compilers convert Java source code into Java bytecode, which is stored in Java `class` files. These are a platform-independent binary distribution format. The bytecode is executed by interpretation or just-in-time compilation (JIT), at runtime.

The runtime system that handles Java bytecode is known as the Java Virtual Machine (JVM) [13]. The JVM has been designed with the explicit aim of supporting Java.

The JVM . . .

- is stack-based

- is secure—type safety is guaranteed by preventing explicit pointer manipulation

- has automatic memory management (garbage collection)

- is object-oriented, with primitive instructions for creating objects, accessing object members, etc.

Although the JVM is primarily designed to be the target for Java high-level language compilers, there are many other compilers which target the JVM [9, 21].

3. COMMON LANGUAGE RUNTIME

The .NET Common Language Runtime (CLR) [14] is designed to be a language-neutral architecture. This is one respect in which the CLR differs from the JVM. However, there are many similarities: The CLR (like the JVM) . . .

- is a stack-based virtual machine

- has a platform-independent bytecode format [12]

- is secure and type-safe

- is garbage collected

- has object-oriented primitive instructions

Thus the CLR is a good target for compilers for modern object-oriented languages like Java and C# [11]. However, the CLR has many other features which are included to provide support for other language styles.

One extreme is the imperative programming style. Consider the example of C [10], which allows arbitrary pointer manipulation. On the CLR, such behaviour is permitted, but we lose our secure, type-safe properties. Also, garbage collection doesn't work any more. Explicit memory allocation and deallocation is required.

The opposite extreme is the functional programming style. Consider the example of Standard ML [15]. Tail recursive function calls are essential for efficient execution, and the CLR provides a tail call instruction. SML .NET [2] takes full advantage of this support.

benchmark	JVM		CLR	
(datasize)	time/s	size/KB	time/s	size/KB
crypt (c)	46	9.8	20	11.7
fft (b)	83	10.0	74	11.7
heapsort (b)	75	8.2	66	9.7
lufact (c)	77	10.6	80	12.3
sparse (c)	192	8.7	194	10.8

Table 1: Java Grande benchmark results

4. GENERAL PERFORMANCE COMPARISON

Size of the bytecode (`class` files for JVM, portable executable files for CLR) reflects on the quality of the source-language compiler. In general, smaller bytecode is better, since it is quicker to load into the virtual machine, and quicker to transmit across slow networks. The time taken to execute programs is the other important consideration. This reflects on the quality of the JIT compiler (JVM or CLR). Faster run-times are always desirable.

Of course, we will not make the absurd claim that code size and code execution time are independent. Sometimes, large code runs slower, because it is not optimal. However, it is just as true that sometimes, large code runs quicker—due to loop unrolling, for example. So, the relationship between code size and code speed is a complex one, which we leave for another day.

We used a selection of programs from the standard Java Grande benchmark suite [4]. The programs chosen are all single-threaded and computationally intensive. The Java Grande benchmark suite has been used as a basis for comparing the performance of different Java execution environments [4], and also as a basis for comparing the performance of programs written in Java with equivalent programs written in other languages [3]. Thus this benchmark suite seems eminently suitable for comparing JVM with CLR, and Java with C#.

Before conducting any tests, it was necessary to port the Java Grande benchmark source from Java to C#. We endeavoured to do this by changing as little source code as possible. Java and C# are remarkably similar languages. We found that on average, less than 5% of the source code needed modification to turn a valid Java program into an equivalent valid C# program.

After this, we compiled the Java benchmarks using the J2SE 1.4.1 `javac` compiler. We compiled the C# benchmarks using Visual Studio .NET v7.0.9466. The compiled Java bytecode was then executed using Java HotSpot Client VM v1.4.1-01. The compiled .NET bytecode was then executed using .NET Framework v1.0.3705. All tests were carried out on a lightly loaded 1.4GHz i686, Win2K Pro system.

From the results in table 1, it is evident that Java/JVM and C#/CLR give nearly identical results for equivalent programs. The Java compiler produces marginally smaller bytecode than the C# compiler, but this appears to be a consistent difference.

The JIT compilers also show similar performance. In two cases (fft, heapsort) CLR gives slightly better execution times, in two cases (lufact, sparse) JVM gives slightly better execution times. The crypt result is an anomaly. CLR executes this benchmark in half the time taken by JVM. This is presumably because crypt does not run for too long, so we encounter JIT warm-up effects, and JVM takes longer to warm-up than CLR. These details are carefully explained in the HotSpot white paper [20].

5. PORTING GCC

The GNU Compiler Collection (GCC) [19] is a world-class optimising compiler. It has been at the heart of the open-source movement for the last twenty years. GCC is a completely modular compiler. It currently boasts front-ends for C, C++, Objective-C, Fortran, Ada and Java. GCC has been ported to all common hardware platforms, and to many more exotic platforms too.

The usual way to port GCC to a new platform [16, 18] involves creating a new compiler backend. Each backend is defined by a set of "machine description" files, which define how GCC's internal Register-Transfer-Language representation maps onto actual target machine instructions.

5.1 egcs-jvm

In 1999, Trent Waddington implemented a GCC backend (egcs-jvm) [22] that targets the JVM. This formed part of the University of Queensland binary translation project [5]. egcs-jvm is able to compile a subset of the C language to JVM bytecode. Waddington experienced tremendous difficulty in implementing many features of C, because they were not supported natively in the JVM, so he had to provide cumbersome workaround tricks to enable . . .

- a linear untyped view of memory, with explicit pointer manipulations

- static global data and its initialisation

- full support for both unsigned and signed integer types

The workarounds are generally inefficient, bloating the code size and slowing down execution time. There are more features of C which egcs-jvm does not even attempt to support, because they are just infeasible on the JVM, e.g. indirect jumps. At present, by Waddington's own admission, development on egcs-jvm is at a stand-still.

5.2 GCC .NET

In 2003, we began to design and implement a GCC backend that targets the CLR (GCC .NET) [17]. This work is still at a preliminary stage, but we are certainly able to compile at least as many programs as egcs-jvm. Of course, we were able to take full advantage of the extra coverage of the CLR instruction set, which includes direct support for the three problems listed above, and many more.

6. GCC PERFORMANCE COMPARISON

We chose a set of simple C benchmark tests from the GCC Benchmarks Collection [6]. These programs only use the basic features of C, all of which should be supported by both egcs-jvm and GCC .NET.

However, while egcs-jvm could be persuaded to compile the benchmarks, the resulting Java bytecode was not executable. First the JVM complained about code verification errors, so we turned off the code verification feature. Then the code began to execute, but threw runtime errors before completing. So, egcs-jvm produced code that left much to be desired.

	JVM		CLR	
benchmark	time/s	size/KB	time/s	size/KB
ack		7.2	<0.1	3.6
ack2		7.0	1.4	3.6
takeuchi2		7.7	0.8	4.1

Table 2: GCC benchmark results

In contrast, GCC .NET compiled the simple benchmark programs, and the resulting .NET bytecode could be executed faultlessly at once.

In table 2, we present the performances of code generated by egcs-jvm and GCC .NET. Notice that the code size is smaller for GCC .NET generated code, since it can handle imperative features directly, without having to resort to ungainly workaround code like egcs-jvm.

As for execution times, we leave the egcs-jvm column blank, a sad reflection on the utter abandonment of this project.

7. CONCLUSIONS

We have demonstrated that there is little performance difference between the JVM and the CLR, for standard object-oriented style programs.

The feature of the CLR that gives it an advantage over the JVM is its ability to handle other language paradigms than just modern object-oriented style. This is what distinguishes the CLR from the JVM. There is no need to employ ugly workarounds to support alternative language styles. Instead, such support is present natively in the CLR instruction set.

8. REFERENCES

[1] K. Arnold and J. Gosling. *The Java Programming Language*. Addison Wesley, second edition, 1997.

[2] N. Benton, A. Kennedy, and C. Russo. SML .NET, 2002. `http://www.cl.cam.ac.uk/Research/TSG/SMLNET`.

[3] J. Bull, L. Smith, L. Pottage, and R. Freeman. Benchmarking Java against C and Fortran for scientific applications. In *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, pages 97–105, 2001.

[4] J. Bull, L. Smith, M. Westhead, D. Henty, and R. Davey. A Benchmark Suite for High Performance Java. *Concurrency: Practice and Experience*, 12(6):375–388, 2000. `http://www.epcc.ed.ac.uk/javagrande/publications.html`.

[5] C. Cifuentes, M. V. Emmerik, and N. Ramsey. The Design of a Resourceable and Retargetable Binary Translator. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, pages 280–291, Atlanta, USA, Oct 1999. IEEE, CS Press.

[6] GCC Benchmarks, 1999. `http://savannah.gnu.org/cgi-bin/viewcvs/gcc/benchmarks/`.

[7] J. Gough. *Compiling for the .NET Common Language Runtime*. Prentice Hall, 2002.

[8] K. J. Gough. Stacking them up: A Comparison of Virtual Machines, 2001. `http://sky.fit.qut.edu.au/~gough/VirtualMachines.ps`.

[9] K. J. Gough and D. Corney. Implementing Languages Other than Java on the Java Virtual Machine, 2000. Presented at Evolve 2000, Sydney, March 2000.

[10] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, 1988.

[11] J. Liberty. *Programming C#*. O'Reilly, 2001.

[12] S. Lidin. *Inside Microsoft .NET IL Assembler*. Microsoft Press, 2002.

[13] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, second edition, 1999.

[14] E. Meijer and J. Gough. Technical Overview of the Commmon Language Runtime, 2001. `http://research.microsoft.com/~emeijer/Papers/CLR.pdf`.

[15] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[16] H.-P. Nilsson. Porting GCC for Dunces, 2000. `ftp://ftp.axis.se/pub/users/hp/pgccfd/pgccfd.pdf`.

[17] J. Singer. GCC .NET—a feasibility study. In *Proceedings of the First International Workshop on C# and .NET Technologies*, Feb 2003. To appear. `http://www.cl.cam.ac.uk/~jds31/research/gccnet`.

[18] R. M. Stallman. *GNU Compiler Collection Internals*. Free Software Foundation, 2002. `http://gcc.gnu.org/onlinedocs/gccint`.

[19] R. M. Stallman. GNU Compiler Collection, 2003. `http://gcc.gnu.org`.

[20] The Java HotSpot Virtual Machine, v1.4.1, 2002. `http://java.sun.com/products/hotspot/`.

[21] R. Tolksdorf. Programming Languages for the Java Virtual Machine, 2003. `http://grunge.cs.tu-berlin.de/~tolk/vmlanguages.html`.

[22] T. Waddington. egcs-jvm, 1999. `http://sourceforge.net/projects/egcs-jvm/`.