

.NET to Java Comparison

F. Plavec

Electrical and Computer Engineering
Department

University of Toronto

10 King's College Road

Toronto, Ontario, M5S 3G4, Canada

franjo.plavec@utoronto.ca

ABSTRACT

Microsoft's .NET Framework, and Sun's Java HotSpot Virtual Machine are latest achievements in platforms independent of underlying system. Both platforms provide runtime environment that is independent of the underlying operating system and computer architecture, and/or programming language used in application development. .NET Framework provides the cross-language operability, but can currently run only on Windows, while Java platform is currently supported on many of the most popular platforms, but its main focus is on Java programming language. Both, .NET and Java, use machine-independent intermediate representation of the program to provide target system independence. The intermediate code is translated into machine specific code at run time and then executed. This approach enables managed code execution, and opens opportunities for runtime optimizations.

In this paper we compare the performance of these two platforms using publicly available Java Grande Benchmark Suite, and its counterpart written in C# programming language. Our measurements show that .NET outperforms Java HotSpot Virtual Machine in most cases. On the average, .NET performs 16 % faster for large-scale applications, and the 75 % faster for kernels. We also evaluate other aspects that might decide the ultimate winner between these platforms.

Categories and Subject Descriptors

D.2.8 [Software Engineering]: Metrics – *Performance measures*

General Terms

Measurement, Performance, C# and Java Languages.

Keywords

.NET, C#, Java, Java Grande, performance, benchmark.

1. INTRODUCTION

Application development process traditionally includes compilation, which is usually done before deploying the application to end users. If multiple platforms are to be supported, multiple versions of executable code need to be produced and tested. Most programs use platform-specific APIs, like operating system calls and graphic libraries. Therefore, besides compiling it for multiple platforms, parts of the application need to be re-written for each of the platforms. All this produces significant overhead for software development companies.

Emergence of the Internet and its availability on all platforms imposes the need for machine-independent platform that would

enable the same code to be run on various computers, regardless of the target operating system and underlying computer architecture. This trend becomes even more popular with the emergence of various mobile and hand-held devices, whose capabilities are significantly different from those of desktop systems.

Efforts to ease and simplify application deployment process resulted in emergence of Sun's Java, and Microsoft's .NET platform. The basic idea behind these platforms is to compile the code to the machine-independent intermediate representation (Byte Code for Java, and Microsoft Intermediate Language - MSIL for .NET). Intermediate code is shipped to the end-user, where it is interpreted or compiled at run time, using Virtual Machine (VM), or Just In Time Compiler (JIT).

In this paper we compare the performance of .NET and Java platform using Java Grande Benchmark suite. The benchmark suite has been re-written to C# programming language in order to run it on .NET platform.

The remainder of the paper is structured as follows. In Section 2 we introduce Java platform and Java HotSpot Virtual Machine. Section 3 gives an overview of .NET platform and its components. Section 4 describes the Java Grande Benchmark suite. In Section 5 we present the results of the measurements performed. We discuss related work in Section 6, and conclude in Section 7. Complete results of all benchmark programs are provided in the Appendix.

2. JAVA PLATFORM

Java was the first platform to enable running the same program on different computer systems, regardless of underlying operating system and architecture. Although some programming languages before Java were supported under more operating systems, they still relied on the system calls, graphic libraries, and other target system dependent features. All parts of the program that used these features needed to be re-designed for each platform to be supported.

Java platform introduces new term: Java Virtual Machine (JVM), or more generally, Virtual Machine (VM). The Java Virtual Machine is an abstract computing machine. Like a real computing machine, it has an instruction set and manipulates various memory areas at the run time [6].

Source program, written in Java, is first compiled to machine-independent code, called bytecode. Virtual machine stands between the program and the target system. At the run time, virtual machine reads the bytecode, and executes it by translating

it to target machine instructions, or operating system calls. Depending on the dynamic of the bytecode execution, the code is said to be interpreted, or just-in-time compiled.

2.1 Code Interpretation

Virtual machines typically interpret the input bytecode. Code interpretation is a way of executing the code by executing each instruction of input code separately. Although the virtual machine, is not limited to interpretation, that is the most common usage of the term virtual machine. Any computer architecture – operating system combination that has Java Virtual Machine implemented is said to support Java platform.

2.2 Just-In-Time Compilation

Just-in-time compilation is done by virtual machines called Just-in-time compilers (JITs). In stead of translating and executing the program instruction by instruction, like interpreters do, JIT compilers compile the code method by method. When the method is called for the first time, its complete code is translated to machine code, and then executed. Generated code is stored into a code cache, so it can be re-used on subsequent calls to the method. Just-in-time compiling takes more time to compile on first call to the method, but this pays out if the method is executed many times.

2.3 Java HotSpot Virtual Machine

The Java HotSpot Virtual Machine is Sun Microsystems' virtual machine for the Java 2 Platform Standard Edition since version 1.3. Java HotSpot VM consists of two basic components: the runtime and the compiler. Runtime includes a bytecode interpreter, memory management and garbage collection functionality, and machinery for handling thread synchronization and other low level tasks. Unlike javac compiler, which translates java code to bytecode, Java HotSpot VM compiler translates bytecode into native code [20].

Java HotSpot Virtual machine takes advantage of so-called 90/10 rule, i.e., that programs spend 90 percent of the time executing 10 percent of the code. Rather than compiling program method by method, like just-in-time compilers, Java HotSpot VM compiles only parts of the code that are executed frequently. Program execution starts by interpretation. During the program execution, the code is analyzed, and critical parts of code that are executed frequently are detected. By avoiding compilation of infrequently executed code (most of the program), the Java HotSpot compiler can devote much more attention to compilation and optimization of the performance-critical parts of the program, without necessarily increasing the overall compilation time. Code analysis is continued dynamically as the program runs, so virtual machine can adapt to changes in the program execution and environment.

There exist two versions of Java HotSpot Virtual Machine: Server and Client version. The Client VM and Server VM are very similar, and share a lot of code. The only part of the system that is different is the compiler (see Figure 1). Compiler that will be used can be chosen by providing appropriate switch to Java HotSpot Virtual Machine. If no switch is provided, client compiler will be used by default

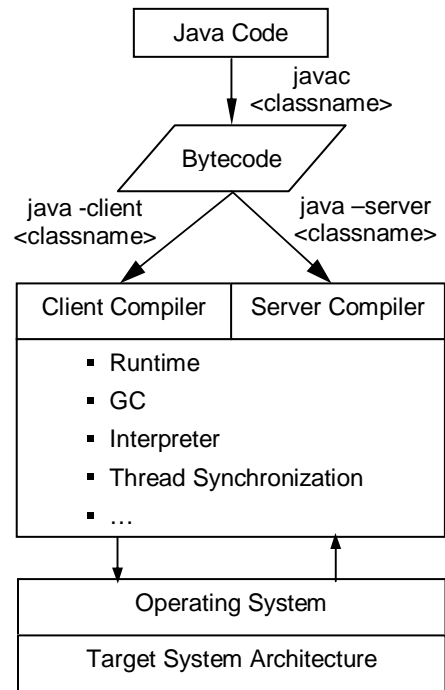


Figure 1. Overview of Java HotSpot Virtual Machine

2.3.1 Java HotSpot Client Compiler

The client compiler is tuned for the performance of profile of typical client applications. Compilation is performed in two phases. In the first phase, a platform-independent front end constructs an intermediate representation (IR) from the bytecodes. In the second phase, the platform-specific background generates machine code from the IR. During code generation, client compiler performs only the simplest optimizations, thus requiring less time to analyze and compile the code. This means that the client VM starts up faster, and requires a smaller memory footprint [19].

2.3.2 Java HotSpot Server Compiler

The server compiler is tuned for the performance profile of typical server applications. The server compiler is advanced adaptive compiler that supports many of the traditional optimizations, including dead code elimination, loop invariant hoisting, common subexpression elimination, and constant propagation. The server compiler also performs some optimizations that are more specific to Java, such as null-check and range-check elimination, aggressive method inlining, and, if necessary, dynamic deoptimization, and possibly reoptimization.

Method inlining is important optimization in any object oriented programming language. Method inlining reduces the number of method invocations and associated performance overhead. Method inlining also opens more space for other optimizations, by producing larger blocks of continuous code.

Dynamic deoptimization is necessary in order to support aggressive inlining in Java. The Java language allows classes to be loaded during runtime, and such dynamically loaded classes

can change the structure of a program significantly, thus making any inlining that was done prior to loading the class invalid.

The Java HotSpot Server compiler is highly portable, relying on a machine description file to describe all aspects of the target architecture [6, 19].

3. .NET PLATFORM

.NET Framework is a platform that provides secure environment for running programs possibly written in multiple source languages. .NET framework consists of two key components: .NET Framework Class Library, and Common Language Runtime (CLR).

.NET Framework Class library is a comprehensive, object-oriented collection of reusable types that can be used to develop applications. These include support for simple data types, I/O functionalities, database support, Graphical User Interfaces (GUIs), and others.

When the source code is compiled, it is translated to Microsoft Intermediate Language (MSIL), machine-independent intermediate language. MSIL includes instructions for loading, storing, initializing, and calling methods on objects, as well as instructions for arithmetic and logical operations, control flow, direct memory access, exception handling, and other low-level operations.

Common Language Runtime provides support for MSIL code execution, by providing features such as code checking and compilation, memory management, thread management, cross-language integration, cross-language exception handling, enhanced security, debugging and profiling services, and others. Basic building blocks of .NET applications are called assemblies. Assemblies were designed to simplify application deployment and to solve versioning problems that can occur with component-based applications. Assemblies contain not only the code that will be executed at run time, but also contain other types of information such as type, security and version information [9].

3.1 .NET Common Language Runtime

.NET CLR provides support for MSIL code execution on .NET Framework platform by providing runtime services, and JIT compiler that converts MSIL to native code. When the type is first loaded, the loader creates and attaches a stub to each of the types methods. On the initial call to the method, the stub passes control to the JIT compiler, which converts the MSIL code for that method into native code and modifies the stub to direct execution to the location of the native code. Generated code is stored in code cache, so subsequent calls to the method proceed directly to the saved native code.

During code compilation JIT compiler performs simple optimizations that do not incur much overhead. These optimizations include constant folding, constant and copy propagation, method inlining, code hoisting, loop unrolling, common subexpression elimination, register allocation (for up to 64 local variables), and some peephole optimizations. JIT compiler also performs some optimizations that can be applied only at run time, such as aggressive inlining, optimizations across assemblies, optimizing away levels of indirection, and processor specific optimizations [10].

The runtime supplies another mode of compilation, called install-time code generation. The install-time code generation mode converts the entire assembly to native code, and stores it to native code cache. The resulting file loads and starts more quickly than it would have if it were being converted to native code by the standard JIT option. The drawback of this approach is that during the install-time code generation optimizations performed by the JIT compiler on the run time cannot be applied, so overall performance might decrease. The problem can be solved by install-time generating only the parts of the code (i.e. assemblies) that are executed at the program start-up, such as form and data initialization. That way the application will start faster, while the most of the code will still be JIT compiled, and take advantage of runtime optimizations.

3.2 C# Programming Language

C# is a new programming language, designed specifically for .NET Framework platform. C# is based on C++ programming language. Since Java language was also initially based on C++, C# and Java have many things in common. However, there are some significant differences in data types, and operations that must be taken into account.

C# source code is prior to execution translated into MSIL code using csc compiler. Compiler provides the command line switch (/optimize+) that enables optimizations while generating the MSIL code. Optimizations include unused local variable and unreachable code removal, and branch optimization. Also, try-catch blocks with an empty try blocks are eliminated, while try-finally blocks with an empty try or finally blocks are converted to normal code [4].

4. JAVA GRANDE BENCHMARK SUITE

The Java Grande Benchmark Suite is a benchmark suite for performance comparison of alternative Java execution environments, in ways which are important to so-called Grande applications. A Grande application is an application which has large requirements for either memory, bandwidth, processing power, or all. Grande applications include computational science and engineering codes, as well as large scale database applications and business and financial models [1, 3, 8].

Java Grande Forum provides different types of benchmarks suitable for various executing environments. Java Grande Sequential Benchmark Suite Version 2.0, which was used in the work described in this paper, is most suitable for execution on single processor systems.

Java Grande Sequential Benchmark Suite is divided into three sections. Section 1 contains benchmark programs that measure the performance of low level operations, such as arithmetic and maths library operations. Section 2 contains benchmark programs consisting of short codes, often called kernels, which carry out specific operations frequently used in Grande applications. Section 3 of sequential benchmark suite contains real Grande codes, suitably modified for inclusion in the benchmark suite by removing any I/O and graphical components. The detailed description of Java Grande Sequential Benchmark Suite is provided in the following sections.

4.1 Section 1

The Section 1 benchmarks are designed to test the performance of low-level operations that will ultimately determine the

performance of real applications. These benchmarks are designed to run for a fixed period of time: the number of operations executed in that time is recorded, and the performance is reported as operations/second. The Section 1 contains following benchmarks: Arith, Assign, Cast, Create, Exception, Loop, Math, Method, and Serial.

The Arith benchmark measures the performance of arithmetic operations (add, multiply and divide) on the primitive data types (int, long, float and double). Performance units are adds, multiplies or divides per second.

The Assign benchmark measures the cost of assigning to different types of variables. The variables may be scalars or array elements, and may be local variables, instance variables or class (static) variables. In the cases of instance and class variables, they may belong to the same class or to a different one. Performance units are assignments per second.

The Cast benchmark tests the performance of casting between different primitive types. The types tested are int, long, float, and double. Performance units are casts per second.

The Create benchmark tests the performance of creating objects and arrays. Arrays of different sizes are created for ints, longs, floats and objects. Complex and simple objects are created, with and without constructors. Performance units are arrays or objects per second.

The Exception benchmark measures the performance of exception handling. The cost of creating, throwing and catching exceptions is measured. Performance units are exceptions per second.

The Loop benchmark measures loop overheads, for a simple 'for' loop, a reverse 'for' loop and a 'while' loop. All loops have empty loop bodies. Performance units are iterations per second.

The Math benchmark measures the performance of all the methods in the java.lang.Math class. Performance units are operations per second. Few of the methods include the cost of an arithmetic operation (add or multiply) into total cost. This is necessary in order to produce a stable iteration, which will not overflow and cannot be optimized away. If necessary, the performance can be corrected by using relevant results from the Arith benchmark.

The Method benchmark determines the cost of a method call. The methods can be instance, final instance or class methods, and may be called from an instance of the same class, or a different one. Performance units are calls per seconds.

The Serial Benchmark measures the performance of serialization, both writing and reading of objects to and from a file. The types of object tested are arrays, vectors, linked lists and binary trees. Performance units are bytes per second.

4.2 Section 2

The Section 2 benchmarks are chosen to be short codes containing the type of computation likely to be found in Grande applications. For each benchmark, a small (size A), medium (size B), and large (size C) version is supplied. Sizes represent the sizing of the problem being solved. The Section 2 contains following benchmarks: Series, LUFact, SOR, HeapSort, Crypt, FFT, and Sparse.

The Series benchmark computes the first N fourier coefficients of the function $f(x) = (x+1)^x$ on the interval (0, 2), where N is 10^4 , 10^5 , and 10^6 , for sizes A, B, and C. Performance units are

coefficients per second. This benchmark heavily exercises trigonometric functions.

The LUFact benchmark solves an N x N linear system using LU factorization followed by a triangular solve, where N is 500, 1000, and 2000, for sizes A, B, and C. Performance units are MFlops per second. This benchmark is memory and floating point intensive.

The SOR benchmark performs 100 iterations of Jacobi Successive Over-relaxation on a NxN grid, where N is 1000, 1500, and 2000, for sizes A, B, and C. SOR exercises typical access patterns in finite difference applications, for example, solving Laplace's equation in 2D with Dirichlet boundary conditions. The performance reported is in iterations per second.

The HeapSort benchmark sorts an array of N integers using a heap sort algorithm, where N is 10^6 , $5*10^6$, and $25*10^6$, for sizes A, B, and C. Performance is reported in number of items sorted per second. This benchmark is memory and integer operation intensive.

The Crypt benchmark performs IDEA (International Data Encryption Algorithm) encryption and decryption on an array of N bytes, where N is $3*10^6$, $2*10^7$, and $5*10^7$, for sizes A, B, and C. Performance units are bytes per second. The Crypt benchmark is bit/byte logical operation intensive.

The FFT benchmark performs a one-dimensional forward transformation of N complex numbers, where N is 2097152, 8388608, and 16777216, for sizes A, B, and C. This kernel exercises complex arithmetic, shuffling, non-constant memory references and trigonometric functions.

The Sparse benchmark uses an unstructured sparse matrix stored in compressed-row format with a prescribed sparsity structure. This kernel exercises indirect addressing and non-regular memory references. An N x N sparse matrix is used for 200 iterations, where N is $5*10^4$, 10^5 , and $5*10^5$, for sizes A, B, and C.

4.3 Section 3

The benchmarks in Section 3 are intended to be representative of Grande applications, suitably modified for inclusion in the benchmark suite by removing any I/O and graphical components. For each benchmark, a small (size A) and large (size B) version is supplied. For each benchmark the execution time is reported. The Section 3 contains following benchmarks: Search, Euler, Moldyn, Monte Carlo, and Raytracer.

The Search benchmark solves a game of connect-4 on a 6 x 7 board using an alpha-beta pruning technique. The problem size is determined by the initial position from which the game is analyzed. This benchmark is memory and integer operation intensive.

The Euler benchmark solves the time-dependent Euler equations for fluid flow in a channel with a "bump" on one of the walls. A structured, irregular, N x 4N mesh is employed, and the solution method is a finite volume scheme using a fourth order Runge-Kutta method with both second and fourth order damping. The solution is iterated for 200 time steps. For benchmark size A, N is 64, while N is 96 for benchmark size B.

The Moldyn benchmark is an N-body code modeling particles interacting under a Lennard-Jones potential in a cubic spatial volume with periodic boundary conditions. The number of particles is given by N (2048 and 8788 for sizes A and B).

The Monte Carlo benchmark is a financial simulation, using Monte Carlo techniques to price products derived from the price of an underlying asset. The code generates N sample time series with the same mean and fluctuation as a series of historical data, where N is $2 \cdot 10^3$, and $6 \cdot 10^4$ for sizes A and B.

The Raytracer benchmark measures the performance of a 3D raytracer. The scene rendered contains 64 spheres, and is rendered at a resolution of NxN pixels, where N is 150, and 500 for sizes A and B.

4.4 C# Grande Benchmark Suite

In order to perform measurements required to compare performance of Java and .NET platforms, Java Grande Benchmark Suite needed to be re-written to one of the programming languages that can be compiled to MSIL. C# programming language was selected, because of the similarities with Java language.

Most of the benchmarks in Java Grande Benchmark Suite contain validation code that enables checking if the program produces expected results. The validation code was used for validation of the code re-written to C#. Many programs in the benchmark suite use random input values. Validation code for these programs relies on the fact that the random number generator with the same seed always produces the same numbers. However, .NET Framework Class Library provides random number generator that produces different numbers than the Java platform for the same seed. In order to successfully use validation code, user class MyRandom was defined, which implements the same random number generator that Java platform uses [17].

Most of the benchmarks were successfully re-written to C# programming language, with two exceptions.

C# version of the Math benchmark in Section 1 does not contain methods Round that take float and double, since these, or equivalent methods do not exist in System.Math namespace of .NET Framework Class Library. .NET Framework Class Library does contain the method Round that takes double as an argument, but that method is equivalent to java.lang.Math rint method.

Validation fails for the Section 3 Moldyn benchmark Size A. The difference between the result produced by Java, and the result produced by .NET platform is $2 \cdot 10^{-12}$. The difference occurs because equivalent methods in java.lang.Math class, and System.Math class sometimes produce results that differ in precision. This difference accumulates in Moldyn benchmark program, since it calculates many values of the low order of magnitude. Size B of the same benchmark program does not fail validation, since the input value set is different.

5. EXPERIMENTAL RESULTS

In this section we present the results of experiments performed. All the experiments described in this section were performed on the system with Intel Pentium 4, 2 GHz processor, 1 GB of physical memory, running Windows 2000 Professional (SP 3) operating system. Java programs were compiled and run on Java 2 SDK 1.4.1 platform, while .NET Framework SDK v1.0.3705 was used for C# programs.

Java programs were run using Java HotSpot Client and Server VMs. .NET programs were run with and without install-time code generation. All C# programs were compiled with options optimize+, and debug-. Complete results of all experiments are

given in the Appendix. This section presents only the averages of the results that are most interesting for comparison. The results are divided into sections in accordance with sections of Java Grande Benchmark Suite.

5.1 Section 1

Figure 2 shows the average performance of Section 1 benchmark programs.

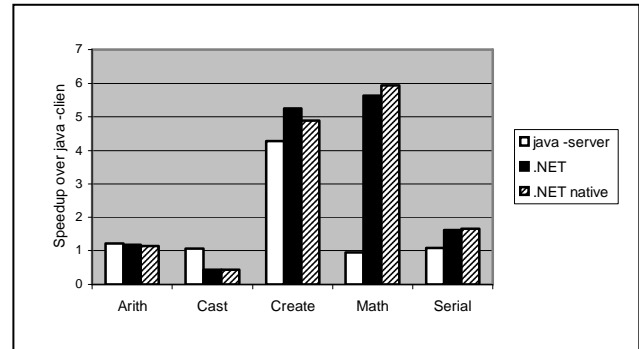


Figure 2. Section 1 programs average performance

First column in Figure 2 represents performance of Java HotSpot Server VM, while the second and the third column represent performance of .NET platform using JIT compilation, and running native, install-time compiled code. All values are relative to the performance of Java HotSpot Client VM.

The results shown in Figure 2 show that .NET is on the average faster than Java for all operations except casts. The reason for poor performance on casts is the internal representation of simple types in .NET platform. Simple types are internally represented as structures, which allows them to be treated as any other objects, but that approach introduces some overhead.

Creating objects in .NET is on the average more than 5 times faster than in Java. However, more detailed examination (see Table 1 in Appendix) shows that .NET creates base object (class Object) slower than Java. This means that the internal organization of base Object class is more complicated in .NET, but the overhead pays out when it comes to creating user objects.

Math bench shows that .NET has better implementation of Math library. Surprisingly, .NET gains this advantage on simple functions, such as Abs, Max, Min, and Round.

Part of the results for Section 1 benchmarks is not shown in the Figure 2, because of disproportion of their values. Exception and Loop benchmarks are completely optimized away by Java HotSpot Server Compiler, thus producing infinite performance for parts of these benchmarks. Method and Assign benchmarks are also highly optimized by this compiler, because of extensive inlining, and common subexpression elimination, thus producing significant performance improvement over Java Client VM.

On the average, .NET turns out to be 2.3 times faster than Java HotSpot Client VM for Section 1 benchmark programs for both, JIT-ed, and native code. Java Hotspot Server VM is on the average 105 times faster than Client VM for Section 1 benchmark programs, excluding the benchmarks which are completely

optimized away. This is mostly due to advanced optimizations that Server Compiler performs.

5.2 Section 2

The average performance of Section 2 benchmark programs is presented in Figure 3.

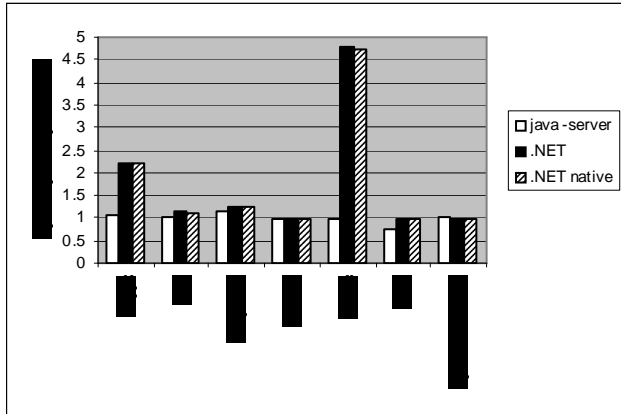


Figure 3. Section 2 programs average performance

All values in Figure 3 are averages of all three benchmark sizes (A, B, and C), and are relative to the performance of Java HotSpot Client VM.

Unlike the Section 1 benchmarks, Section 2 benchmark programs show more equality in performance. In most cases performance is approximately equal for all configurations, with two exceptions: Series and Crypt.

Series benchmark heavily exercises trigonometric functions, which have shown to be faster in Section 1 benchmarks, and that is the reason for .NET’s advantage in this benchmark program. Crypt benchmark extensively uses logical operations. Unfortunately, Section 1 of the benchmark suite does not contain the program that would test the performance of these operations, so it is not possible to say if this is really the reason for .NET’s advantage in Crypt benchmark.

On the average, both configurations of .NET platform exercise the speedups of 75 % over Java HotSpot Client VM, while Server VM on the average suffers from slowdown of 2 % for Section 2 benchmarks.

5.3 Section 3

Figure 4 shows the average performance of Section 3 benchmark programs.

Values shown in Figure 4 are averages of benchmark sizes A and B, and are relative to the performance of Java HotSpot Client VM.

Benchmark programs in Section 3 are real Grande applications. Speedups that .NET platform exercises running these programs can be explained by the speedups in more elemental operations already mentioned for Section 1 and 2 benchmarks.

The only benchmark program that suffers from slowdown on .NET platform is Euler. This can be explained by extensive usage of arrays and array assignments, which are slower on .NET platform than Java (see Table 1 in Appendix).

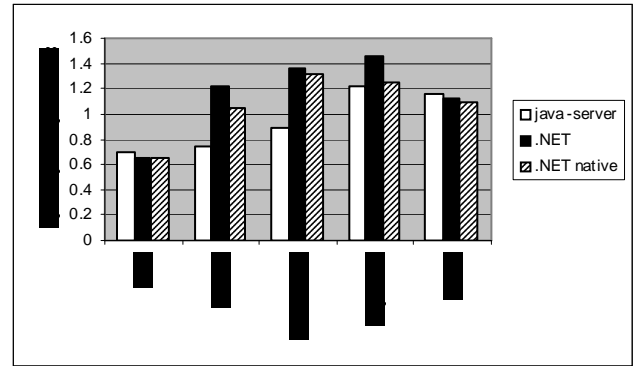


Figure 4. Section 3 programs average performance

On the average, .NET JIT-ed code is 16 % faster than the Java HotSpot Client VM for Section 3 benchmark programs. However, .NET install-time compiled code is only 7 % faster than Client VM. This is because large scale applications profit from runtime optimizations which are not performed when the code is pre-compiled [10, 11, 12].

Java HotSpot Server VM suffers the average slowdown of 7 % over Client VM when running Section 3 benchmark programs. This was unexpected, since the Server VM should benefit from advanced optimization on large-scale applications. The programs in Section 3 are still not “large enough” to benefit from the optimizations performed by Server VM. The confirmation for this conclusion is the fact that the average slowdown for size A of Section 3 of the benchmark suite is 9 %, while it is only 4 % for size B, which means that the performance improves with problem size. It is reasonable to expect that this slowdown would turn into speedup when the sizing of the problem increases sufficiently.

6. RELATED WORK

The Java HotSpot Virtual Machine architecture is the culmination of many years of research at Sun Microsystems [19]. Improvements in Java Virtual Machine performance were expected, and necessary, since the basic virtual machines based on interpretation, and simple JIT compilers showed to be too slow for real word applications.

Since the .NET is fairly new platform, there has been much less research conducted in the area of performance improvement targeting specifically this platform. Moreover, part of the information on current research activities is kept secret by Microsoft, and unavailable to public. There is an ongoing project at the Indian Institute of Science on profile-guided optimizations for a .NET JIT compiler, sponsored by Microsoft Research. The work should result in implementation and evaluation of applicability of profile guided dynamic recompilation [5].

Currently .NET platform is supported only on Windows. There is an ongoing project to port the .NET Common Language Runtime to Linux, supported by Microsoft [13]. There is no data available on any other platforms to be supported in the future.

There are many benchmarks available for performance evaluation of various Java platforms [2, 3, 7, 14, 15]. However, there is no publicly available benchmark suite for .NET platform performance evaluation. This is mainly due to the fact that the .NET platform is fairly new, but the situation is expected to

change when the new version of .NET platform becomes available.

7. CONCLUSION

Java and .NET Framework provide means of executing programs independently of the target system, and/or the programming language used. This is particularly useful for Internet applications that need to execute on any target computer.

One of the important aspects that needs to be considered when choosing between these two platforms is performance. The most common way of evaluating performance is benchmarking. In this paper we presented Java Grande Benchmark Suite, and its counterpart written in C# programming language as a means of evaluating performance of Java and .NET platforms.

The results obtained by measurement show that .NET provides better performance opportunities for program execution. On the average, .NET performs 16 % faster for large-scale applications, and the 75 % faster for kernels than Java HotSpot Client VM.

There are other aspects besides the performance that might also affect the wide usage of one or the other platform. While .NET Framework can currently run only on Windows, Java is true multi-platform environment, supported on all major platforms in use today. Although .NET Framework was also designed for multi-platform support, it yet needs to develop support for platforms other than Windows.

On the other hand, .NET Framework supports more of the most popular programming languages. Although both platforms can support virtually any programming language, Java currently does not support many popular languages. The emphasis of Java platform is on the usage of Java programming language.

It is reasonable to expect that both .NET Framework, and Java will play significant role in the world of independent platforms in the future. The advantages and disadvantages of these platforms might determine the ultimate winner, or they might end-up in everlasting battle, bringing more and more performance and features with each version.

8. REFERENCES

- [1] Bull J. M., Smith L. A., Westhead M. D., Henty D. S., and Davey R. A. A Methodology for Benchmarking Java Grande Applications. <http://citeseer.nj.nec.com/bull99methodology.html>, 1999.
- [2] Dongarra J., Wade R, and McMahan P. Linpack Benchmark, <http://www.netlib.org/benchmark/linpackjava>, 2000.
- [3] EPCC. The Java Grande Forum Benchmark Suite, http://www.epcc.ed.ac.uk/javagrande/index_1.html
- [4] Gunnerson E. A Programmer's Introduction to C#, Apress, 2000.
- [5] Indian Institute of Science. Profile-guided optimizations for a .NET JIT compiler, <http://purana.csa.iisc.ernet.in/~kapil/project.htm>, November 2002.
- [6] Lindholm T., and Yellin F. The Java (TM) Virtual Machine Specification, Second Edition, <http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html>, July 2002.
- [7] Marshall S., Gangelen M., and Sy A. The Plasma Benchmark, <http://rsb.info.nih.gov/plasma>
- [8] Mathew J. A., Coddington P. D., and Hawick K. A. Analysis and Development of Java Grande Benchmarks. In Proc. of the ACM 1999 Java Grande Conference, San Francisco, April 1999.
- [9] Microsoft Corporation. .NET Framework SDK, MSDN Library, <http://msdn.microsoft.com/library/default.asp?url=/nhp/Default.asp?contentid=28000451>
- [10] Microsoft Corporation. Performance Considerations for Runtime Technologies in the .NET Framework, MSDN Library. <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/dotnetperftechs.asp>, August 2001.
- [11] Microsoft Corporation. Performance Optimization in Visual Basic .NET, MSDN Library, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dv_vstechart/html/vbtchperfopt.asp, September 2002.
- [12] Microsoft Corporation. Performance Tips and Tricks in .NET Applications, MSDN Library, <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/dotnetperftips.asp>, August 2001.
- [13] MONO: Open source .NET implementation for Linux. <http://www.go-mono.com>, December 2002.
- [14] Pozo R., and Miller B. SciMark 2.0, <http://math.nist.gov/scimark2>
- [15] SPEC. SPEC JVM98 Benchmarks, <http://open.spec.org/osg/jvm98>, 2001.
- [16] Sun Microsystems, Inc. Frequently asked questions about the Java (TM) HotSpot Virtual Machine, <http://java.sun.com/docs/hotspot/PerformanceFAQ.html>
- [17] Sun Microsystems, Inc. Java.util.Random class. <http://java.sun.com/j2se/1.3/docs/api/java/util/Random.html>
- [18] Sun Microsystems, Inc. The Java HotSpot (TM) Performance Engine Architecture. White Paper, <http://java.sun.com/products/hotspot/whitepaper.html>, April 1999.
- [19] Sun Microsystems, Inc. The Java HotSpot (TM) Virtual Machine. Technical White Paper, http://java.sun.com/products/hotspot/docs/whitepaper/Java_HotSpot_WP_Final_4_30_01.pdf, May 2001.
- [20] Wilson S., and Kesselman J. Java (TM) Platform Performance, Strategies and Tactics. http://java.sun.com/docs/books/performance/1st_edition/html/JPTitle.fm.html, June 2000.

APPENDIX: Experimental Results

Table 1. Section 1 benchmark results

	Benchmark (Units)	Java – client	Java - server	.NET	.NET install-time generated code
Arith (adds/s, multiplies/s, divides/s)	Add:Int	3.60E+008	1.39E+009	9.00E+008	9.01E+008
	Add:Long	2.25E+008	1.25E+008	1.34E+008	1.33E+008
	Add:Float	1956251.8	1979891.8	1982959	1982959
	Add:Double	1950476.2	1976834	1982959	1985843
	Mult:Int	1.37E+008	1.40E+008	1.28E+008	1.28E+008
	Mult:Long	7.31E+007	4.50E+007	1.24E+008	1.00E+008
	Mult:Float	1968095.4	1998048.8	1989122	1991830
	Mult:Double	1894191.6	1905116.2	1916347	1919041
	Div:Int	3.48E+007	3.48E+007	3.48E+007	3.48E+007
	Div:Long	1.40E+007	1.95E+007	1.77E+007	1.77E+007
	Div:Float	1800439.5	1867080	1817859	1820445
	Div:Double	1833154.4	1902461.6	1835619	1846209
Assign (assignments/s)	Same:Scalar:Local	9.33E+008	1.34E+011	2.22E+009	2.21E+009
	Same:Scalar:Instance	1.04E+009	3.20E+010	1.04E+009	1.04E+009
	Same:Scalar:Class	9.19E+008	3.73E+010	1.06E+009	1.05E+009
	Same:Array:Local	4.80E+008	1.10E+009	3.83E+008	2.80E+008
	Same:Array:Instance	2.44E+008	5.57E+008	1.83E+008	1.81E+008
	Same:Array:Class	4.79E+008	1.09E+009	3.87E+008	3.87E+008
	Other:Scalar:Instance	3.50E+008	3.20E+010	1.03E+009	1.04E+009
	Other:Scalar:Class	3.56E+008	3.73E+010	1.06E+009	1.06E+009
	Other:Array:Instance	2.04E+008	1.10E+009	5.66E+008	6.55E+008
	Other:Array:Class	3.24E+008	1.10E+009	8.66E+008	4.79E+008
Cast (casts/s)	IntFloat	5.84E+007	5.04E+007	1.38E+007	1.38E+007
	IntDouble	5.95E+007	4.34E+007	1.70E+007	1.71E+007
	LongFloat	2.01E+007	2.98E+007	1.16E+007	1.23E+007
	LongDouble	2.01E+007	2.39E+007	1.29E+007	1.31E+007
Create (arrays/s)	Array:Int:1	3.32E+007	2.08E+007	4.25E+007	3.71E+007
	Array:Int:2	2.27E+007	1.89E+007	3.63E+007	3.47E+007
	Array:Int:4	1.71E+007	1.58E+007	3.00E+007	2.73E+007
	Array:Int:8	1.14E+007	1.15E+007	2.67E+007	2.34E+007
	Array:Int:16	6907833.5	7094483.5	1.69E+007	1.68E+007
	Array:Int:32	3826964.5	3996097.5	1.12E+007	1.03E+007
	Array:Int:64	1950476.2	3200750.2	5904144	6124860
	Array:Int:128	1016074.6	1717832.6	3666309	3256480
	Array:Long:1	2.27E+007	1.89E+007	3.75E+007	3.51E+007
	Array:Long:2	1.72E+007	1.58E+007	3.40E+007	3.02E+007
	Array:Long:4	1.15E+007	1.15E+007	2.63E+007	2.31E+007
	Array:Long:8	6916582	7114199	1.76E+007	1.63E+007
	Array:Long:16	3838081	3977857.8	1.05E+007	1.05E+007

Benchmark (Units)	Java – client	Java - server	.NET	.NET install-time generated code	
Create (arrays/s, objects/s)	Array:Long:32	1965073.9	3224435.2	6472819	5864834
	Array:Long:64	1019209.7	1726958.4	3322518	3117200
	Array:Long:128	530212.8	855543.5	1830696	1657092
	Array:Float:1	3.30E+007	2.08E+007	4.14E+007	3.54E+007
	Array:Float:2	2.26E+007	1.90E+007	3.88E+007	3.35E+007
	Array:Float:4	1.70E+007	1.59E+007	3.24E+007	3.03E+007
	Array:Float:8	1.14E+007	1.16E+007	2.62E+007	2.39E+007
	Array:Float:16	6853509.5	7152711	1.67E+007	1.44E+007
	Array:Float:32	3804923.2	4008220	1.02E+007	1.03E+007
	Array:Float:64	1938843.1	3189037.8	6354821	6110697
	Array:Float:128	1011358	1720141.1	3462969	2788671
	Array:Object:1	3.28E+007	2.08E+007	4.29E+007	3.71E+007
	Array:Object:2	2.25E+007	1.89E+007	4.02E+007	3.58E+007
	Array:Object:4	1.69E+007	1.58E+007	3.40E+007	3.03E+007
	Array:Object:8	1.13E+007	1.16E+007	2.46E+007	2.32E+007
	Array:Object:16	6809076.5	7152711	1.41E+007	1.47E+007
	Array:Object:32	3777204	4008612.2	1.02E+007	9190038
	Array:Object:64	1924631.1	3208522.8	5767796	5662151
	Array:Object:128	1002055	1722311	2555050	3037674
	Object:Base	6.17E+007	6.48E+007	5.36E+007	4.76E+007
	Object:Simple	2664065	6.49E+007	4.74E+007	4.94E+007
	Object:Simple:Constructor	2658704.5	6.47E+007	5.30E+007	4.59E+007
	Object:Simple:1Field	2480019.5	3.51E+007	5.00E+007	4.80E+007
	Object:Simple:2Field	2416091.5	3.51E+007	4.24E+007	4.06E+007
	Object:Simple:4Field	2422665.2	2.38E+007	3.07E+007	3.00E+007
	Object:Simple:4fField	2420517.8	2.37E+007	3.01E+007	2.98E+007
	Object:Simple:4LField	2357410	1.43E+007	1.98E+007	1.91E+007
	Object:Subclass	2655946	6.45E+007	5.15E+007	4.50E+007
	Object:Complex	2315825.2	2.26E+007	2.12E+007	1.92E+007
	Object:Complex:Constructor	2289419.2	2.25E+007	2.46E+007	1.81E+007
Exception (exceptions/s)	Throw	3547549	Infinity	66276.6	60144.72
	New	240578.89	275135.7	64350.71	56732.56
	Method	228510.22	206085.97	58018.31	49527.94
Loop (iterations/s)	For	4.24E+008	Infinity	1.31E+009	1.35E+009
	ReverseFor	9.98E+008	Infinity	1.31E+009	1.27E+009
	While	5.00E+008	4.37E+008	9.99E+008	9.75E+008
Math	AbsInt	3.77E+007	4.14E+007	9.27E+008	9.27E+008
	AbsLong	4.39E+007	4.16E+007	2.38E+008	2.37E+008
	AbsFloat	3.61E+007	3.44E+007	2.71E+008	2.71E+008

	Benchmark (Units)	Java – client	Java - server	.NET	.NET install-time generated code
Math (operations/s)	AbsDouble	3.74E+007	3.38E+007	2.71E+008	2.71E+008
	MaxInt	3.73E+007	4.23E+007	8.87E+008	1.04E+009
	MaxLong	3.98E+007	3.26E+007	2.13E+008	2.22E+008
	MaxFloat	2.83E+007	3.03E+007	4.49E+007	4.43E+007
	MaxDouble	3.48E+007	3.46E+007	1.10E+007	1.08E+007
	MinInt	3.73E+007	4.23E+007	8.87E+008	1.04E+009
	MinLong	3.95E+007	3.24E+007	2.14E+008	2.19E+008
	MinFloat	2.88E+007	3.00E+007	4.57E+007	4.58E+007
	MinDouble	3.47E+007	3.45E+007	1.13E+007	1.11E+007
	SinDouble	9920678	8532000	1.14E+007	1.14E+007
	CosDouble	8532000	7363595.5	9718251	9718251
	TanDouble	2846223.2	2580157.5	5806223	5917365
	AsinDouble	650778.5	631397.2	1761720	1776082
	AcosDouble	519296.12	505679	1717977	1722311
	AtanDouble	7521807	6971915	6611250	6569894
	Atan2Double	4955538	3799276.5	7721018	7676162
	FloorDouble	6907833.5	6594751	2.01E+007	1.96E+007
	CeilDouble	6907833.5	6586797.5	1.64E+007	1.63E+007
	SqrtDouble	5.22E+007	4.21E+007	5.14E+007	5.22E+007
	ExpDouble	2085539.8	1979891.8	3109627	3095058
	LogDouble	3177657	2999194.5	6472819	6553600
	PowDouble	863479.2	892841.56	2992621	3002713
	RintDouble	6594751	6386030.5	1.30E+008	1.30E+008
Random	2903111.5	3037674.2	1.25E+007	1.14E+007	
IEEERemainderDouble	527237.1	540283.9	385978.1	385978.1	
Method (calls/s)	Same:Instance	1.63E+008	8.39E+010	2.14E+008	2.15E+008
	Same:SynchronizedInstance	3.68E+007	5.05E+007	1.63E+007	1.52E+007
	Same:FinalInstance	1.66E+008	7.44E+010	2.01E+008	2.02E+008
	Same:Class	1.52E+008	7.44E+010	2.01E+008	2.01E+008
	Same:SynchronizedClass	4.31E+007	5.75E+007	1.63E+007	1.49E+007
	Other:Instance	3.56E+007	6.72E+010	2.01E+008	2.01E+008
	Other:InstanceOfAbstract	3.56E+007	7.44E+010	1.53E+008	1.53E+008
	Other:Class	4.41E+007	7.49E+010	2.01E+008	2.02E+008
Serial (bytes/s)	Writing:Linklist	3.14E+005	3.44E+005	5.10E+005	4.53E+005
	Reading:Linklist	3.34E+005	3.51E+005	5.26E+005	5.35E+005
	Writing:Binarytree	3.00E+005	3.45E+005	3.82E+005	3.71E+005
	Reading:Binarytree	2.85E+005	3.16E+005	5.06E+005	4.98E+005
	Writing:Vector	3.26E+005	3.52E+005	4.01E+005	5.62E+005
	Reading:Vector	311340.84	3.40E+005	5.82E+005	6.15E+005
	Writing:Array	337786.56	3.61E+005	5.99E+005	5.93E+005
	Reading:Array	351407	3.84E+005	6.42E+005	6.42E+005

Table 2. Section 2 benchmark results

Benchmark (Units)	Size	Java -client	Java -server	.NET	.NET install-time generated code
Crypt (Kbyte/s)	A	2.59E+03	2.71E+03	6.09E+03	6.00E+03
	B	2.63E+03	2.78E+03	5.70E+03	5.73E+03
	C	2.63E+03	2.79E+03	5.72E+03	5.72E+03
FFT (Samples/s)	A	1.22E+05	1.22E+05	1.39E+05	1.39E+05
	B	1.07E+05	1.06E+05	1.23E+05	1.18E+05
	C	1.01E+05	1.02E+05	1.12E+05	1.12E+05
HeapSort (items/s)	A	1.12E+06	1.33E+06	1.49E+06	1.49E+06
	B	7.14E+05	8.21E+05	8.86E+05	8.84E+05
	C	5.26E+05	5.79E+05	6.05E+05	6.07E+05
LUFact (Mflops/s)	A	1.37E+02	1.31E+02	1.34E+02	1.34E+02
	B	1.39E+02	1.33E+02	1.33E+02	1.35E+02
	C	1.45E+02	1.45E+02	1.42E+02	1.43E+02
Series (coefficients/s)	A	1.34E+02	6.47E+02	2.62E+03	2.61E+03
	B	1.35E+02	6.23E+02	2.63E+03	2.61E+03
	C	1.43E+02	4.10E+02	2.62E+03	2.61E+03
SOR (Iterations/s)	A	4.00E+01	3.00E+01	3.93E+01	3.93E+01
	B	1.78E+01	1.34E+01	1.75E+01	1.75E+01
	C	1.00E+01	7.55E+00	9.79E+00	9.79E+00
SparseMatmult (Iterations/s)	A	2.84E+01	2.95E+01	2.90E+01	2.90E+01
	B	9.30E+00	9.55E+00	9.13E+00	9.12E+00
	C	1.49E+00	1.46E+00	1.41E+00	1.41E+00

Table 3. Section 3 benchmark results. Reported performance is execution time in seconds

Benchmark	Size	Phase	Java –client	Java -server	.NET	.NET install-time generated code
Euler	A	Init	0.281	0.875	0.453	0.297
		Run	9.813	15.812	15.297	15.297
		Total	10.109	16.703	15.765	15.594
	B	Init	0.469	0.985	0.156	0.156
		Run	21.968	27.984	33.5	34.391
		Total	22.469	28.985	33.672	34.562
Moldyn	A	Run	5.75	7.375	4.562	5.484
		Total	5.906	7.438	4.562	5.484
	B	Run	112.687	163.469	99.032	110.64
		Total	112.781	163.891	99.047	110.656
Monte Carlo	A	Run	15.156	17.672	10.672	10.984
		Total	15.438	18.266	10.922	11.219
	B	Run	89.672	96.734	68.61	70.687
		Total	90.641	97.953	69.516	71.609
Raytracer	A	Init	0.016	0.016	0.015	0
		Run	13.593	11.672	9.532	10.985
		Total	13.609	11.688	9.562	10.985
	B	Init	0	0	0	0
		Run	159.61	126.047	106.985	126.156
		Total	159.625	126.063	107	126.156
Search	A	Run	9.5	8.407	8.422	8.516
	B	Run	39.922	34.14	35.593	37.109