HPC.NET - are CLI-based Virtual Machines Suitable for High Performance Computing?

Werner Vogels

Dept. of Computer Science, Cornell University
http://www.cs.cornell.edu/vogels
vogels@cs.cornell.edu

ABSTRACT

The Common Language Infrastructure is a new, standardized virtual machine that is likely to become popular on several platforms. In this paper we review whether this technology has any future in the high-performance computing community, for example by targeting the same application space as the Java-Grande Forum. We review the technology by benchmarking three implementations of the CLI and compare those with the results on Java virtual machines.

1. INTRODUCTION

The Common Language Infrastructure (CLI) is probably better known as the Common Language Runtime (CLR) of the Microsoft .NET Framework. The CLR, however, is only one of the implementations of the ECMA-335 standard; others such as SSCLI [15] and Mono [13] have been released or are in beta development phases.

The ECMA standard document [6] described the technology as follows:

> *ECMA-335 defines the Common Language Infrastructure (CLI) in which applications written in multiple high level languages may be executed in different system environments without the need to rewrite the application to take into consideration the unique characteristics of those environments.*

Many of us are familiar with the latter of these goals as 'write-once, run-everywhere', which is one of the major goals of the Java virtual machine [8,11]. Java has been investigated in the past as a viable platform for high performance computing, especially by the Java Grande Forum [10]. Under the leadership of Forum, a number of benchmarks [1,2,3,4,12,14] have been published that are used to compare JVM implementations, and their suitability for high-performance computing.

Our research goal is to investigate whether CLI virtual machine implementations can be useful for the high-performance computing community. The ability to use multiple programming languages, including FORTRAN, on a single virtual machine is very attractive from a software development perspective. Developers targeting a CLI can mix languages without any problem, making it simpler to transition legacy libraries and applications into modern portable runtime environments. The portability of type-safe runtimes becomes more and more important with increased platform heterogeneity and the widespread deployment of grid-computing environments.

In this research we take the approach that the community has already accepted the premise that the Java virtual machine has some application in HPC, and that if the performance of CLI-based virtual machines is comparable or better than the performance of the best Java virtual machines there is a possible future for CLI-based high-performance computing. If this is true, then we believe that there may be an even bigger future for CLI-based virtual machines, given the significant software engineering benefits.

We have taken a similar approach to the Java Grande community by bringing a number of the popular HPC benchmarks to the CLI. In this paper we compare the results the benchmark on different CLI-based virtual machines and compare the CLI results with the results of the JVM on the same platform. The benchmarks are available for public download so the results can be independently verified.

The paper is organized as follows: In section 2 of this paper we give an overview of the CLI and the design considerations behind the technology. In section 3 we describe the benchmarks that are the subject of this paper, the porting process and the verification of the benchmarks. Section 4 presents the results of the benchmarks on

SC'03, November 15-21, 2003, Phoenix, Arizona, USA.

different CLI and Java virtual machines. In section 5 we investigate some of the performance issues that were noted on the different architectures. We close of the paper with a brief conclusion and statement of future work.

2. CLI VIRTUAL MACHINE TECHNOLOGY

The objective of the CLI is to make it easier to write components and applications from any language. It does this by defining a standard set of types, making all components fully self-describing and providing a high performance common execution environment. This ensures that all CLI-compliant system services and components will be accessible to all CLI-aware languages and tools. In addition, this simplifies deployment of components and applications that use them, all in a way that allows compilers and other tools to leverage the execution environment. The Common Type System covers, at a high level, the concepts and interactions that make all of this possible.

Behind the CLI specification and execution model are a core set of concepts. These key ideas were folded into the design of the CLI both as abstractions and as concrete techniques that enable developers to organize and partition their code. One way to think of them is as a set of design rules [15]:

- Expose all programmatic entities using a unified type system
- Package types into completely self-describing, portable units
- Load types in a way that they can be isolated from each other at runtime, yet share resources
- Resolve intertype dependencies at runtime using a flexible binding mechanism that can take version, culture-specific differences (such as calendars or character encodings), and administrative policy into account
- Represent type behavior in a way that can be verified as type safe, but do not require all programs to be type-safe
- Perform processor-specific tasks, such as layout and compilation, in a way that can be deferred until the last moment, but do not penalize tools that do these tasks earlier
- Execute code under the control of a privileged execution engine that can provide accountability and enforcement of runtime policy
- Design runtime services to be driven by extensible metadata formats so that they will gracefully accommodate new inventions and future changes

There are basically 4 main areas in the CLI specifications where these design decisions come together:

- *Common Type System* –The Common Type System (CTS) provides a rich type system that supports the types and operations found in many programming languages. The CTS is intended to support the complete implementation of a wide range of programming languages
- *Metadata* – The CLI uses metadata to describe and reference the types defined by the Common Type System. Metadata is stored ("persisted") in a way that is independent of any particular programming language. Thus, metadata provides a common interchange mechanism for use between tools that manipulate programs (compilers, debuggers, etc.) as well as between these tools and the Virtual Execution System
- *Common Language Specification* – The Common Language Specification is an agreement between language designers and framework (class-library) designers. It specifies a subset of the CTS Type System and a set of usage conventions. To provide users with the greatest ability to access frameworks, languages by implement at least those parts of the CTS that are part of the CLS. Similarly, frameworks will be most widely used if their publicly exposed aspects (classes, interfaces, methods, fields, etc.) use only types that are part of the CLS and adhere to the CLS conventions
- *Virtual Execution System* – The Virtual Execution System (VES) implements and enforces the CTS model. The VES is responsible for loading and running programs written for the CLI. It provides the services needed to execute managed code and data, using the metadata to connect separately generated modules together at runtime (late binding.)

2.1. The CLI Execution Engine

The execution engine (VES or Virtual machine) is where all the concepts come together; it provides the environment for executing managed code. It provides direct support for a set of built-in data types, defines a hypothetical machine with an associated machine model and state, a set of control flow constructs, and an exception-handling model It manages the component and isolation model, as well as runtime services such as exception handling, automatic heap and stack management, threading and execution domains. JIT compilation, memory management, assembly and class loading, type resolution, metadata parsing, stack walking, and other fundamental mechanisms are implemented here.

2.2. Available implementations

There are multiple implementations of the ECMA standard available; most prominently the two commercial implementations from Microsoft: the .NET Framework and

the .NET Compact Framework, which are both in production use. Next to these are the Shared Source CLI (SSCLI aka 'rotor') and Mono, which are semi-community projects and are available in source and binary forms.

The only CLI-targeted programming language standardized within ECMA is C# [6], but the different CLI implementations may come with additional compilers. There is a large community of language developers who are building CLI-targeted compilers for languages such as Eiffel, FORTRAN, COBOL, ML, Python, Oberon, etc.

- *.NET Common Language Runtime*. This is the commercial CLR developed by Microsoft for all x86-based workstation and server platforms. The current version is 1.1 and is packaged with all new Windows operating systems and is integrated into the visual studio development tools. It comes out-of-the-box with compilers for C++, Visual Basic, C#, JavaScript and Java, which all can produce 'managed code' to be executed by the CLR virtual machine.
- *.NET Compact Framework*. This framework (CF) targets small devices with processors such as StrongARM, XScale, SH4 and MIPS. There are notable differences between the CLR and the CF, but mainly in the support services, such as limited runtime code generation, no install time code generation, no remoting, limited unsafe code interoperability, no generational GC, and there are two CIL instructions missing that are related to indirect function calls. Most importantly, the CF has a different Just-In-Time compile engine that optimizes for a small memory model and reduced power consumption.
- *Shared Source CLI*. This is a public implementation of the ECMA specification, also known under the codename Rotor by a team at Microsoft. The code is as close to the production CLR as possible but is focused on portability instead of performance optimization. It has a different JIT and a generic portability layer that shields the core form platform specifics. It currently targets the x86 and PowerPC architectures and the Window, FreeBSD and MacOSX operating systems. An earlier version was also ported to Linux.
- *Mono*.The Mono project, sponsored by Ximian, is an effort to create an open source, portable implementation of the ECMA specifications. Mono includes a compiler for the C# language, a CLI virtual machine and a set of base class libraries. The system currently runs x86 (JIT), StrongARM, S390, PowerPC and Sparc (interpreter, JIT under development), supported operating systems are Windows and Linux. *Mano* is a version of Mono that is targeted toward PalmOS5.

The .NET Compact Framework will not be examined in this paper, as its goals are not performance oriented.

3. BENCHMARKS

The project focuses on porting and developing benchmarks, as well as executing them on a variety of platforms. The initial goal is to establish a public repository of benchmarks that can be used to compare different CLI implementations; these benchmarks are able to examine low-level functionality as well as library and application level modules. All the benchmarks investigate and compare the CLIs in the light of high-performance computing. In this sense, the project has similar goals to the Java-Grande benchmark project.

The common language available for all implementations is C#, so it is necessary to port the benchmarks to this language or develop new benchmarks that deal with CLI-specific issues in C#. We would have liked to use the Java versions of the most common benchmarks directly, but the only compiler available at this moment is the .NET J# compiler, which emits code that uses some .NET CLR-specific libraries for support. Therefore, it is not possible to use the intermediate code from this compiler directly under other runtimes as we do not want to mix runtime-specific libraries.

The benchmarks are set up in such a way that we use a single compiler (the CLR 1.1 C# compiler) to generate the intermediate code, and this code is then executed on each of the different runtimes.

The second goal of the project is to compare the CLI-based results with the results of the benchmarks on different JVM implementations. For this it is necessary to keep the C# code as close as possible in structure and layout to the Java version of the benchmarks. Support code such as timers and random number generators are kept identical between the C# and Java versions, even though more efficient implementation could have been made.

The third goal of the project is to investigate the performance characteristics in terms of memory consumption, data/instruction cache hits and misses, IO intensity, etc. For this we use a set of external commercial grade profilers to control the execution as well as to inspect the environment. These results are outside of the scope of this paper.

3.1. The porting process

The process of porting Java to C# is relatively simple. We have used some of the automated translation tools as well as manual translation, but with either approach the process is rather straightforward. The parts that need the most work are those that use platform-specific services such as serialization and object persistence. Some of the code from specific Java libraries is not available in CLI-based libraries such as a Gaussian method on the Random number generator or specific input tokenizers.

Name	Description
Arith	Measures the performance of arithmetic operations.
Assign	Measures the cost of assigning to different types of variable.
Cast	Tests the performance of casting between different primitive types.
Create	Tests the performance of creating objects and arrays.
Exception	Measures the cost of creating, throwing and catching exceptions, both in the current method and further down the call tree.
Loop	Measures loop overheads.
Serial	Tests the performance of serialization, both writing and reading of objects to and from a file.
Math	Measures the performance of all the methods in the Math library.
Method	Determines the cost of method calls.

Table 1: Micro-benchmarks from section 1 of the Java Grande 2.0 benchmark.

3.2. The Micro-benchmarks

The micro-benchmark suite tests low-level operations such as arithmetic, loop overhead, exception handling, etc.(see table 1) Most of the tests come from section 1 of the Java Grande v2.0 benchmark[3]. From section 1 of the multi-threaded Java Grande v1.0 benchmark [14] we have taken the benchmarks for barrier synchronization, fork-join tests and method-object synchronization (table 2). There were a number of specific cases that we wanted to investigate and that were not covered by the Java Grande tests, namely true multi-dimensional matrix versus jagged array, boxing and un-boxing of value types, and additional threading and lock tests (table 3). Some of these tests were based section 1a of the DHPC Java Grande benchmark [12].

3.3. The Macro-Benchmarks

In this suite we include most of the kernels and applications from the SciMark benchmark, sections 2 and 3 of the serial Java Grande v2.0, and Section 2a of the DHPC Java Grande benchmark. The complete list of benchmarks can be found in table 4.

Table 4: Macro benchmarks derived form the Scimark, section 2 an 3 of the Java Grande suit and section 2a of the DHPC Java Grande suite

Name	Description
Barrier	This measures the performance of barrier synchronization. Two types of barriers have been implemented: the Simple Barrier uses a shared counter, while the Tournament Barrier uses a lock-free 4-ary tree algorithm.
ForkJoin	This benchmark measures the performance of creating and joining threads.
Synchronization	This benchmark measures the performance of synchronized methods and synchronized blocks under contention.

Table 2: Micro-benchmarks from section 1 of the multi-threaded Java Grande 1.0 benchmark

Name	Description
Matrix	This measures assignments of different styles of matrices, such as jagged versus true multidimensional.
Boxing	This tests the explicit and implicit boxing and unboxing of value types.
Thread	Measures the startup costs of using additional threads.
Lock	Tests the use of locking primitives under different contention scenarios.

Table 3: Micro-benchmarks developed to test specific CLI technology

3.4. The Current State of the Benchmark Porting

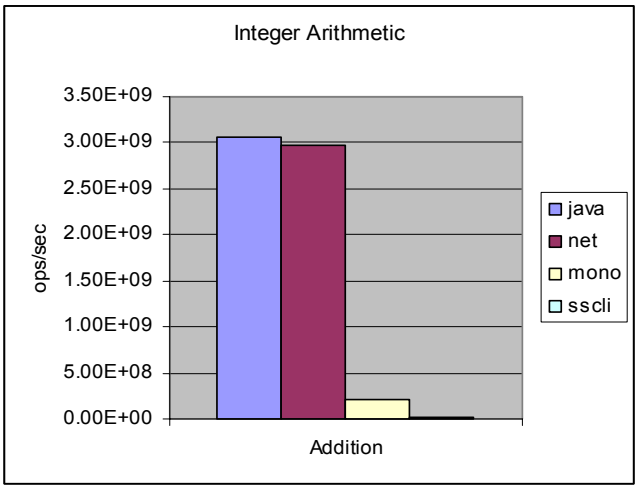
All the serial versions of the benchmarks have been ported to C# and compile to CIL code. The port of the parallel versions, for shared memory and for message based parallelization, is planned to begin once we have sufficient experience with the current benchmarks implementations.

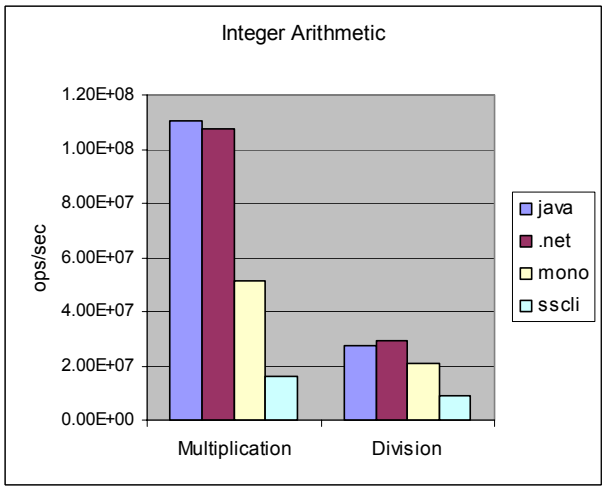
The focus of our current effort is on the validation of the results of the computations by the different kernels. We have completed this validation for the benchmarks in the SciMark suite (FFT, SOR, LINPACK, SparseMatMul and Monte Carlo), and these are currently available for download. Also completed and available for download are the micro-benchmarks. We expect to have the complete suite of benchmarks validated and available for download by the time of the conference.

Name	Description
Fast Fourier Transform	FFT performs a one-dimensional forward transform of 4K complex numbers. This kernel exercises complex arithmetic, shuffling, non-constant memory references and trigonometric functions. The first section performs the bit-reversal portion (no flops) and the second performs the actual Nlog(N) computational steps.
Fibonacci	Calculates the 40th Fibonacci number. It measures the cost of many recursive method calls, which are required to generate the sequence.
Sieve	Calculates prime numbers using the Sieve of Eratosthenes. It uses integer arithmetic with a lot of array overhead.
Hanoi	Solves the 25-disk Tower of Hanoi problem, which is a sorting puzzle.
HeapSort	Sorts an array of N integers using a heap sort algorithm.
Crypt	Performs IDEA (International Data Encryption Algorithm encryption and decryption on an array of N bytes.
Linpack	Computes the LU factorization of a dense NxN matrix using partial pivoting. Exercises linear algebra kernels (BLAS) and dense matrix operations. The algorithm is the right-looking version of LU with rank-1 updates.
Sparse matrix multiply	This benchmark uses an unstructured sparse matrix stored in compressed-row format with a prescribed sparsity structure. This kernel exercises indirection addressing and non-regular memory references.
SOR.	Jacobi Successive Over-relaxation on a NxN grid exercises typical access patterns in finite difference applications, for example, solving Laplace's equation in 2D with Drichlet boundary conditions. The algorithm exercises basic "grid averaging" memory patterns, where each A(i,j) is assigned an average weighting of its four nearest neighbors.
Monte Carlo.	A financial simulation, using Monte Carlo techniques to price products derived from the price of an underlying asset. The integration approximates the value of Pi by computing the integral of the quarter circle y = sqrt(1 - x^2) on [0,1]. The algorithm exercises random-number generators, synchronized function calls, and function inlining.
MolDyn	This benchmark is an N-body code modeling argon atoms interacting under a Lennard-Jones potential in a cubic spatial volume with periodic boundary conditions. The computationally intense component of the benchmark is the force calculation, which calculates the force on a particle in a pair-wise manner. This involves an outer loop over all particles in the system and an inner loop ranging from the current particle number to the total number of particles.
Euler	Solves the time-dependent Euler equations for flow in a channel with a “bump”on one of the walls. It uses a structured, irregular Nx4N mesh.
Search	Solves a game of connect-4 on a 6 _ 7 board using a alpha-beta pruned search technique. The benchmark is memory and integer intensive.
RayTracer	This benchmark measures the performance of a 3D ray tracer. The scene rendered contains 64 spheres, and is rendered at a resolution of NxN pixels.

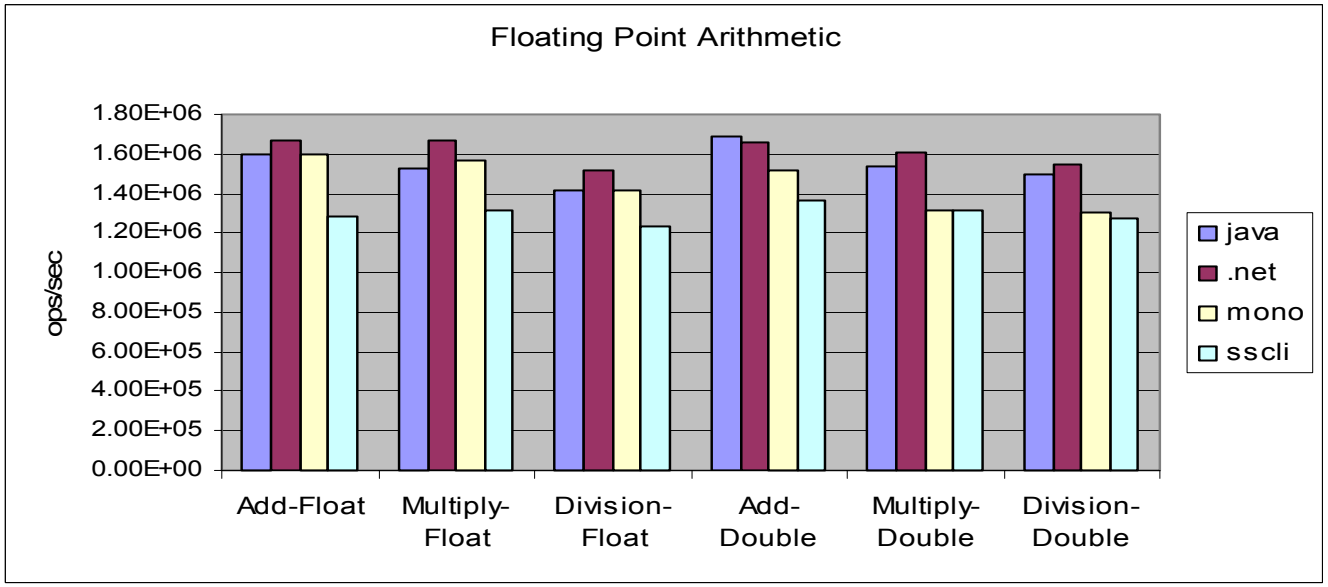
Table 4: Macro benchmarks derived form the Scimark, section 2 an 3 of the Java Grande suit and section 2a of the DHPC Java Grande suite

.

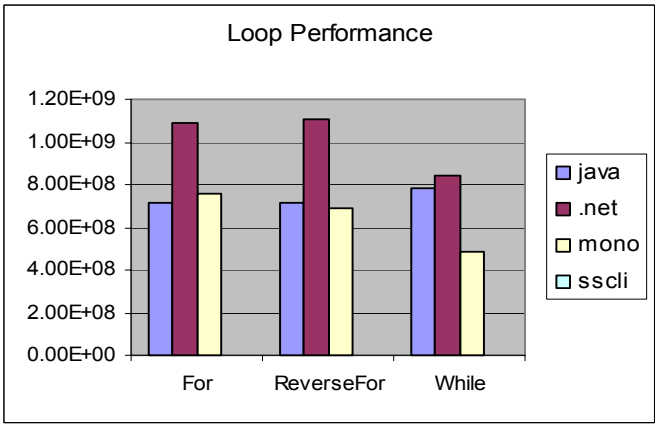


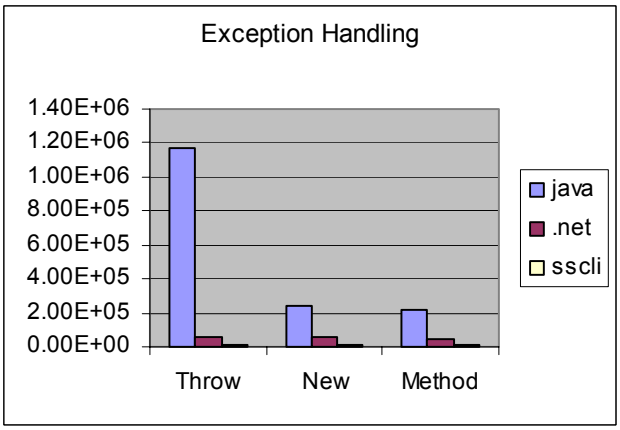


Graph 1-2: Integer Arithmetic

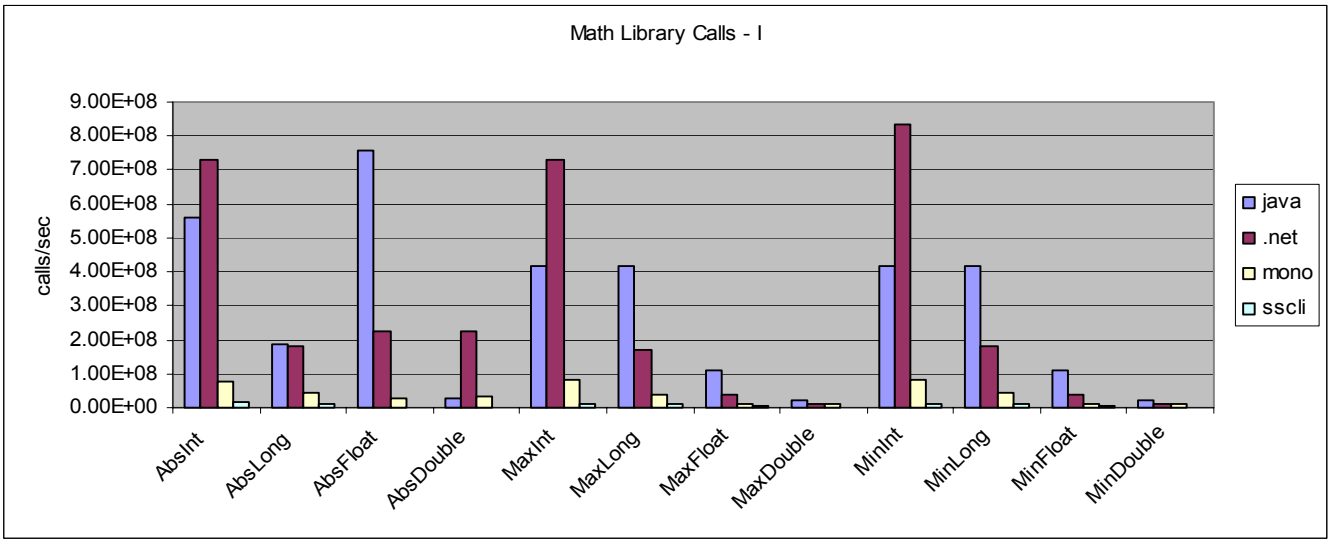


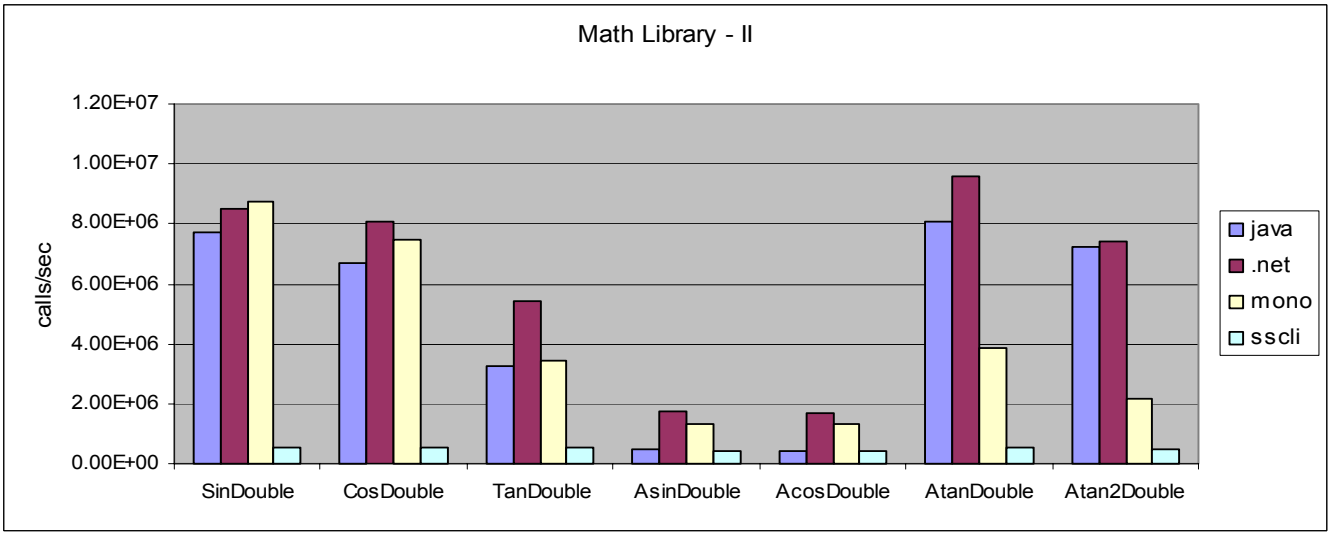
Graph 3: Floating Point arithmetic

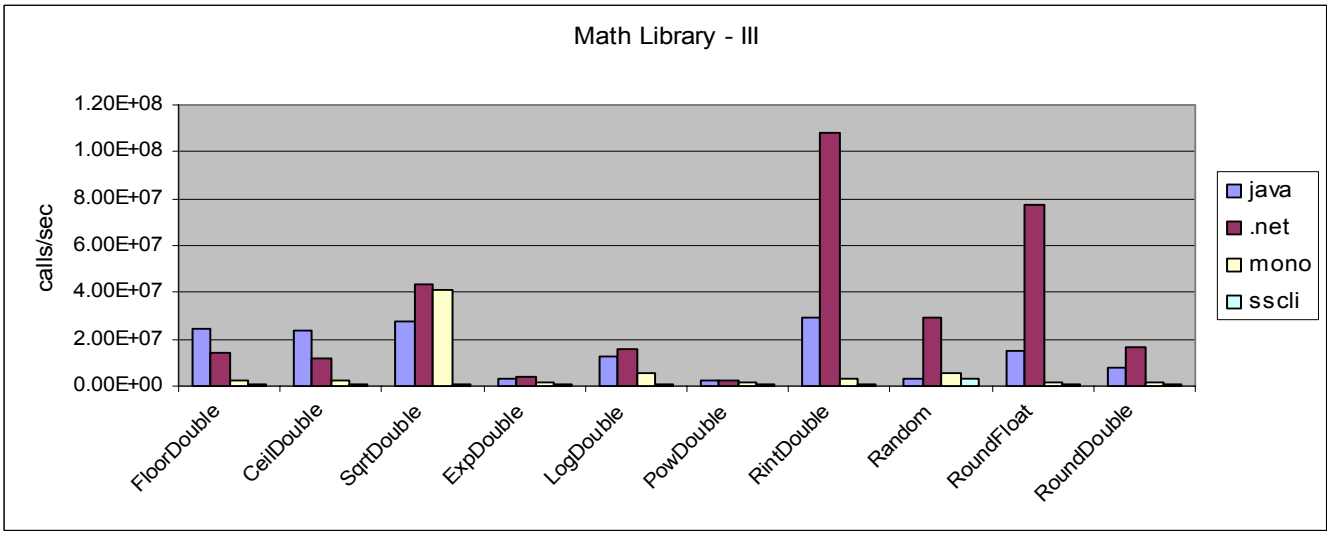




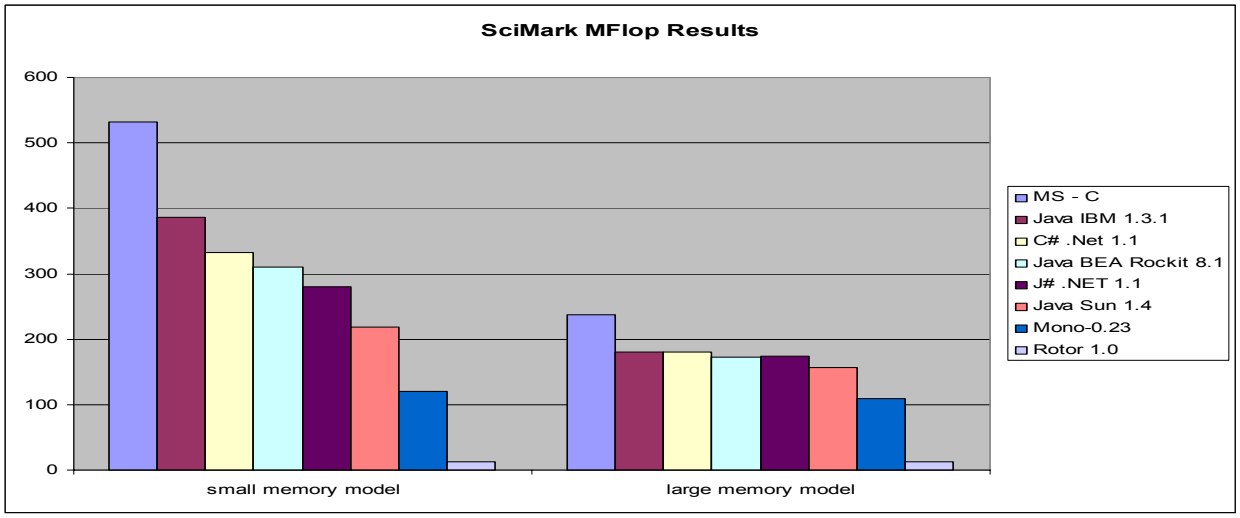
Graph 4-5: Performance of Loops and Exception handling



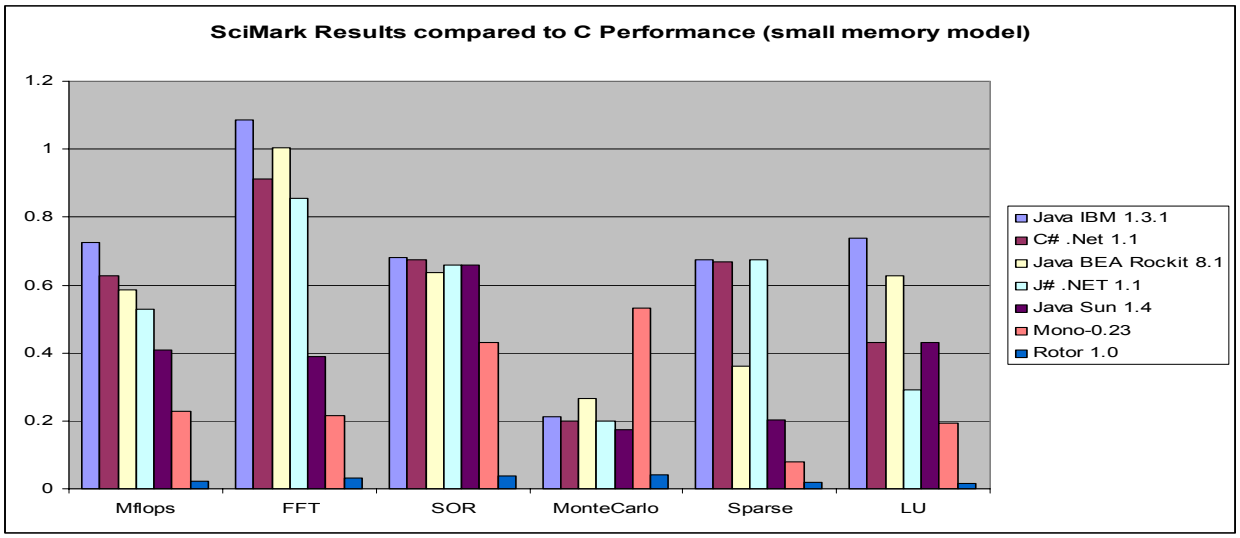




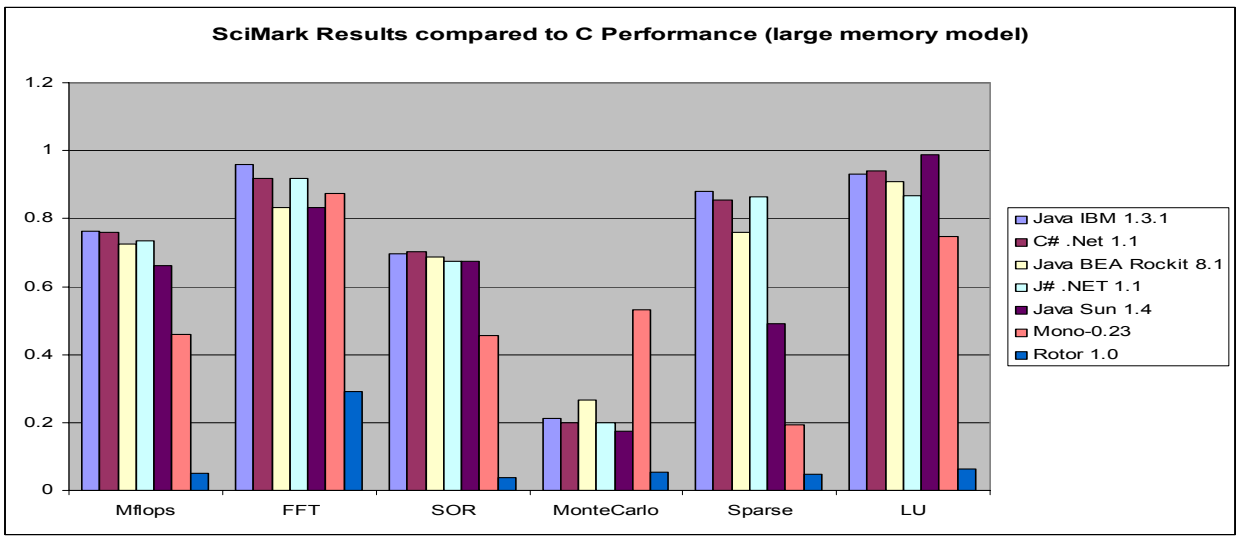
Graph 6-8: Performance of the various Math Library routines (higher numbers is better)



Graph 9: SciMark results in MFlops for both memory models.



Graph 10: SciMark results of the individual benchmark kernels in the small memory model



Graph 11: SciMark results for the large memory problem size.

4. BENCHMARK RESULTS

In this section we will present some of the micro- and macro-benchmark results. The micro-benchmark tests reported here were run on an Intel Pentium P4 XEON 2.8 GHz, with 2 GB of memory, with Windows Server 2003as the operating system. The benchmarks were also run on other architectures, and the results presented for this machine are considered to be representative for a large class of Intel Pentium III and 4 machines.

For the micro-benchmarks we compared IBM JVM 1.3.1, Microsoft .NET CLR 1.1, Mono 0.23 and SSCLI 1.0. The benchmarks were executed 100 times, and the results were inspected for outliers, of which there were none. The numbers presented in the graph are representative of a single run. Graphs 1-5 are a small subset of the available results: they present some integer and floating point arithmetic, loop performance and exception handling. Graphs 6-8 present the performance of the different Math library routines.

For the macro-benchmarks we present here the SciMark numbers on a dual processor Intel Pentium P4 XEON 2.2 GHz, running Windows Server 2003. Added to the list of virtual machines to test are the BEA JRocket 8.1 Server JVM, the Sun 1.4.1 HotSpot JVM and the Microsoft J# compiler for .NET 1.1. Graphs 9-11 show main MFlops results as well as the result of the individual kernels.

5. EVALUATION

When we examine at the graphs presented in the previous section we see that there are clear differences between the 3 CLI implementations and the JVM that we have tested. From 30,000 miles there appears to be a pattern where often Mono runs at half the performance of CLR 1.1 and SSCLI is 5 to 10 times as slow, and the performance of the JVM is close to the CLR 1.1 performance.

But when examining the individual benchmark results we see that the performance variations in the different benchmarks are too erratic to support this conclusion. For example, if we examine the arithmetic micro-benchmarks, we see that some integer operations in the CLR will perform (addition and division) slower but others (e.g. multiplication) will run faster, when compared to the JVM. If the performance difference in these benchmarks would be similar across the board, we could possibly draw conclusions that these were structural issues in the systems. The variations in the microbenchmarks results suggest that there are forces other than fundamental architectural ones that influence the outcome of the benchmarks. We will need to have a closer look at the origin of the differences in performance.

If we take the integer division test as an example, we see that the code (see table 6) repeatedly divides the result of the previous division by a constant. The resulting CIL code (also table 6) is rather simple: push the two variables on the stack, execute the division, and pop the result into the first variable.

When we look at the x86 machine code that is generated and executed we see major differences between the output of the different virtual machines. The machine language generated by Mono is very close to the actual CIL code, it uses two memory locations for each of the variables, loads those and stores the result again. The Rotor code more or less does the same but makes a mess of it by emulating the *cdq* instruction with loads and shifts. The CLR 1.1 code is clearly more optimized: it realizes that a constant is used and that the result of the division can be left in a register. It does something weird by temporarily storing the constant in a variable, which appears to be an unnecessary operation. The Java code does not make this mistake and uses registers and constants throughout the loop. The relevant piece of code for CLR and JVM can be found in table 7 and for Mono and Rotor in table 8.

If we repeat this exercise for the integer addition benchmarks, we see similar differences in generated machine code. In this benchmark 4 integers are added in circular fashion repeatedly. The CLR 1.1 uses only registers, as does the IBM JVM, Mono uses one register and three memory locations, and SSCLI four memory locations.

This leads us to conclude that there is a significant variety in the level of optimization of the code emitted by the different Just-In-Time components. Some JITs are capable of highly optimizing a few operations, but different JITs seem to optimize different operations, and not all optimization have the same level of success.

```
int size = INITSIZE;
int i1 = Int32.MaxValue;
int i2 = 3;
…
        for (int i = 0; i
< size; i++)
        {
                i1 /= i2;
                i1 /= i2;
                i1 /= i2;
                i1 /= i2;
                i1 /= i2;
                i1 /= i2;
                i1 /= i2;
                i1 /= i2;
                i1 /= i2;
                i1 /= i2;
                i1 /= i2;
                i1 /= i2;
                i1 /= i2;
                i1 /= i2;
                i1 /= i2;
                i1 /= i2;
        }
```

The C# code for the integer division arithmetic benchmark

```
IL_0000:  ldc.i4
0x2710
IL_0005:  stloc.0
IL_0006:  ldc.i4
0x7fffffff
IL_000b:  stloc.1
IL_000c:  ldc.i4.3
IL_000d:  stloc.
…
IL_0038:  ldloc.1
IL_0039:  ldloc.2
IL_003a:  div
IL_003b:  stloc.1
```

The resulting relevant piece of CIL code

Table 5: Extract from the C# benchmark code and the resulting CIL

mov esi,7FFFFFFFh
....
mov esi,eax
mov eax,3
mov dword ptr [esp+10h],eax
mov eax,esi
cdq
mov ecx,dword ptr [esp+10h]
idiv eax,ecx
	call 19B01BFF
	mov ecx,dword ptr [esi]
	mov edx,eax
	mov eax,esi
	....
	cdq
	mov ebp,3
	idiv eax,ebp

The relevant pieces of x86 machine code generated by the .NET CLR 1.1 JIT	The code generated by the IBM 1.3.1 JVM JIT

Table 6: The x86 machine code generated by the two commercial virtual machine JIT engines

mov dword ptr [ebp-30h],7FFFFFFFh
mov dword ptr [ebp-34h],3
....
mov eax,dword ptr [ebp-30h]
mov ecx,dword ptr [ebp-34h]
cdq
idiv eax,ecx
mov ebx,eax
mov dword ptr [ebp-30h],ebx
	mov eax,0x7fffffff
	mov [ebp-0x14],eax
	....
	mov eax,[ebp-0x14]
	push eax
	mov eax,[ebp-0x18]
	mov ecx,eax
	pop eax
	mov edx,eax
	sar edx,0x1f
	idiv ecx
	mov [ebp-0x14],eax

The code generated by the Mono-0.23 JIT	The code generated by the SSCLI 1.0 JIT

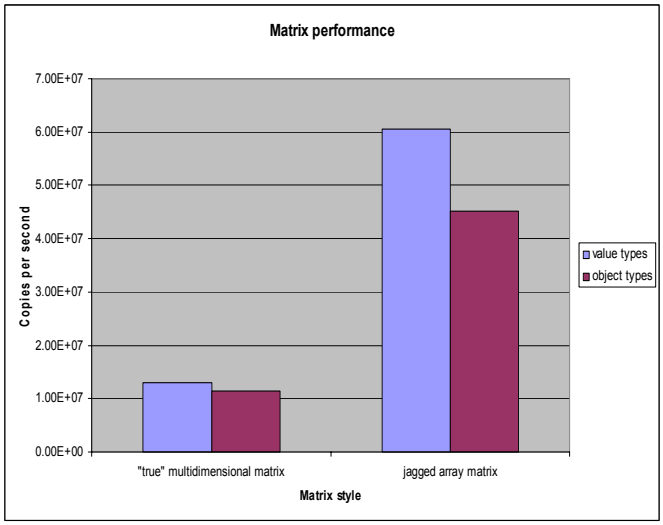
Table 7. The machine code generated by the two open source virtual machine JIT engines

There is a variety of factors that determine the machine code produced, for example the CLR 1.1 JIT, will produce better optimized code for smaller code segments; for example if the integer addition benchmark is isolated into a separate code segment, the JIT always success into finding the optimal code, if the benchmark shares the code segment with the other arithmetic benchmark, the JIT is at times less successful. This is related to the fact the CLR 1.0 and 1.1 JITs only consider a maximum of 64 local variables for enregistration (tracking local variables for storage in registers), and all the remaining variable will be located in the stack frame.

The level of optimizations produced by the JIT engines appears to be the dominating factor in the resulting performance of the low-level compute benchmarks. Given that JIT engine technology is under constant improvement it is likely that we will see continuous performance improvements in the coming years. Many of the optimizations depend on how much knowledge the JIT engine has built-up about the state of the program, and what shortcuts can be taken under which conditions. One of these shortcuts is for example the elimination of in-loop array bound checks when the array index has a known relationship to the loop counter. In this case a good JIT engine will perform a single bounds check before the loop, improving the loop performance dramatically. In CLR 1.1, we can easily force this optimization by using the array.length property as the bounds in the loop; if we introduce this for example in the sparse matrix multiply kernel of the SciMark benchmark instead of using a separate variable, we see an instant performance improvement of 15% or more.

If we look at the outcome of the micro-benchmarks, we can see that there are a there are a few areas of consistent differences: The CLR 1.1 version of the Math library appears to perform better than the Java version (see graphs 6-8), the loop overhead in CLR 1.1 is lower (graph 4), while exception-handling in all implementations of the CLI is significantly more costly than in the JVM (graph 5).

It is ironic to see that one of the major bottlenecks identified by the Java Grande Forum, the lack of true multidimensional arrays, does not appear under the CLR. Copy assignments in true multidimensional matrices run at 25 percent of the performance of jagged arrays, again due to optimized JIT optimizations (graph 12).



Graph 12: Comparing true multi-dimensional with jagged Matrix performance in .NET CLR 1.1

From the SciMark benchmarks results we see that the two major commercial platforms are close in performance. There where the benchmarks mainly focus on integer arithmetic, the JVM has the advantage because of better optimized JIT code emission. There where matrix manipulation plays an important role we see that the CLR gains the advantage. Running the benchmarks at larger problem sizes reduces the slight advantage the JVM has due to the better optimized array management in the CLR.

One particular point with respect to the SciMark benchmark must be noted. The Monte Carlo simulation in

the benchmark is mainly a test of the access to synchronized methods. The C++ version of the benchmarks does not have any of these locking primitives and as such the comparison does not yield a valid result. The excellent performance the mono version achieves makes us believe that its synchronization implementation does not make use of the native OS synchronization techniques, as it outperforms local locking mechanisms. We are investigating these architectural issues in current phase of our project.

6. CONCLUSIONS

The question we wanted to investigate in this paper was: Are CLI implementations useful for high-performance computing community? Assuming that the community has accepted Java as a possible target for HPC, we see that the benchmarks indicate that the commercial implementation the CLI, the .NET CLR 1.1, performs as good as the top-of-the-line Java Virtual Machine, the IBM 1.3.1 JVM, and significantly better than the BEA and Sun implementations of the JVM.

The main performance difference in the virtual machines can be brought back to the optimizations the individual developers of the Just-In-Time code emission engines have found. As such, it is likely that over time all the virtual machines will have comparable performance as the JITs start to emit more and more optimized codes.

As for the differences among the three CLI implementations, we have seen the performance of Mono steadily improving with each new release and we expect that it will mirror the performance of the CLR in the future. From the performance numbers of the SSCLI we can only conclude that it needs a new JIT if it wants to play a role in any environment that takes performance seriously.

The source for the benchmarks used in this paper and the future benchmarks can be downloaded from the CLI-Grande website: http://cli-grande.sscli.net

7. ACKNOWLEDGEMENTS

The author was supported, in part, by DARPA under AFRL grant RADC F30602-99-1-0532 and by AFOSR under MURI grant F49620-02-1-0233. Additional Support was provided through a grant by the Microsoft Corporation.

Dan Fay provided the motivation to write the paper, Peter Drayton helped out with debugging the JIT output and Cindy Ahrens provided valuable editorial assistance.

8. REFERENCES

[1] Bull, J.M., Smith, L.A., Pottage, L., Freeman, R. "Benchmarking Java against C and Fortran for Scientific Applications", in Proceedings of ACM Java Grande/ISCOPE Conference, June 2001.

[2] Bull, J.M., Smith, L.A., Westhead, M.D., Henty, D.S. and Davey, R.A., "Benchmarking Java Grande Applications", in Proceedings of the Second International Conference on The Practical Applications of Java, Manchester, U.K., April 2000, pp. 63-73

[3] Bull, J.M., Smith, L.A.,Westhead, M.D., Henty, D.S. and Davey, R.A., "A Benchmark Suite for High Performance Java", Concurrency, Practice and Experience, vol. 12, pp. 375-388.

[4] Bull, J.M., Smith, L.A., Westhead, M.D., Henty, D.S. and Davey, R.A. "A Methodology for Benchmarking Java Grande Applications", in Proceedings of ACM 1999 Java Grande Conference, June 1999, ACM Press, pp. 81-88.

[5] ECMA, Standard ECMA-334 C# Language Specification 2nd edition, December 2002, http://www.ecma-international.org/publications/files/ecma-st/Ecma-334.pdf

[6] ECMA, Standard ECMA-335 Common Language Infrastructure (CLI) 2nd edition, December 2002, http://www.ecma-international.org/publications/files/ecma-st/Ecma-335.pdf

[7] Erik Meijer and John Gough, "Technical Overview of the Common Language Runtime", http://research.microsoft.com/~emeijer/Papers/CLR.pdf

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

[9] Gough, J., "Stacking them up: A Comparison of Virtual Machines", in Proceedings ACSAC-2001, 2001.

[10] Java Grande Forum Panel, "Java Grande Forum Report: Making Java Work for High-End Computing", Supercomputing 1998, Nov. 13, 1998, http://www.javagrande.org/reports.htm

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

[12] Mathew, J.A., and P. D. Coddington and K. A. Hawick, "Analysis and Development of Java Grande Benchmarks", in the Proceedings. of the ACM 1999 Java Grande Conference, San Francisco, 1999",

[13] Mono Project, http://www.go-mono.org

[14] Smith, L.A., and J. M. Bull, "A Multithreaded Java Grande Benchmark Suite", in Proceedings of the Third Workshop on Java for High Performance Computing, Sorrento, Italy, June 2001

[15] Stutz, D., Neward, T., Shilling, G., "Shared Source CLI", O'Reilly & Associates, March 2003