The Scientific Byte Code Virtual Machine

Rasmus Andersen
University of Copenhagen
eScience Centre
2100 Copenhagen, Denmark
Email: rasmus@diku.dk

Brian Vinter
University of Copenhagen
eScience Centre
2100 Copenhagen, Denmark
Email: vinter@diku.dk

***Abstract*—Virtual machines constitute an appealing technology for Grid Computing and have proved a promising mechanism that greatly simplifies and enforces the employment of grid computer resources.**

While existing sandbox technologies to some extent provide secure execution environments for applications deployed in a heterogeneous platform as the Grid, they suffer from a number of problems including performance drawbacks and specific hardware requirements.

This project introduces a virtual machine capable of executing platform-independent byte codes specifically designed for scientific applications. Native libraries for the most prevalent applications domains mitigate the performance penalty. As such, grid users can view this machine as a basic grid computing element and thereby abstract away the diversity of the underlying real compute elements.

Regarding security, which is of great concern to resource owners, important aspects include stack isolation by using a harvard memory architecture, and no support for neither I/O nor system calls to the host system.

Keywords: Grid Computing, virtual machines, scientific applications.

I. Introduction

Although virtualization was first introduced several decades ago, the concept is now more popular than ever and has revived in a multitude of computer system aspects that benefit from properties such as platform independence and increased security. One of those applications is grid Computing[5] which seeks to combine and utilize distributed, heterogeneous resources as one big virtual supercomputer. Regarding utilization of the public's computer resources for grid computing, virtualization, in the sense of virtual machines, is a necessity for fully leveraging the true potential of grid computing. Without virtual machines, experience shows that people are, with good reason, reluctant to put their resources on a grid where they have to not only install and manage a software code base, but also allow native execution of unknown and untrusted programs. All these issues can be eliminated by introducing virtual machines.

As mentioned, virtualization is by no means a new concept. Many virtual machines exist and many of them have been combined with grid computing. However, most of these were designed for other purposes and suffer from a few problems when it comes to running high performance scientific applications on a heterogeneous computing platform. Grid computing is tightly bonded to eScience, and while standard jobs may run perfectly and satisfactory in existing virtual machines, 'gridified' eScience jobs are better suited for a dedicated virtual machine in terms of performance.

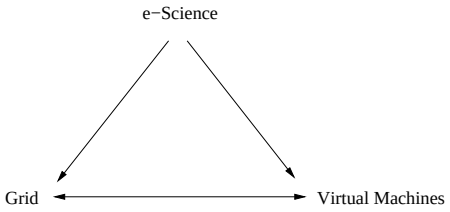

Fig. 1. Relatationship between VMs, the Grid, and eScience

Hence, our approach addresses these problems by developing a portable virtual machine specifically designed for scientific applications: The Scientific Byte Code Virtual Machine (SciBy VM).

The machine implements a virtual CPU capable of executing platform independent byte codes corresponding to a very large instruction set. An important feature to achieve performance is the use of optimized native libraries for the most prevalent algorithms in scientific applications. Security is obviously very important for resource owners. To this end, virtualization provides the necessary isolation from the host system, and several aspects that have made other virtual machines vulnerable have been left out. For instance, the SciBy VM supports neither system calls nor I/O.

The following section (II) motivates the usage of virtual machines in a grid computing context and why they are beneficial for scientific applications. Next, we describe the architecture of the SciBy VM in Section III, the compiler in Section IV, related work in Section VI and conclusions in Section VII.

II. Motivation

The main building blocks in this project arise from properties from virtual machines, eScience, and a grid environment in a combined effort, as shown in figure 1.

The individual interactions impose several effects from the viewpoint of each end, described next.

A. eScience in a Grid Computing Context

eScience, modelling computationally intensive scientific problems using distributed computer networks, has driven the development of grid technology and as the simulations get

more and more accurate, the amount of data and needed compute power increase equivalently. Many research projects have already made the transition to grid platforms to accommodate the immense requirements for data and computational processing. Using this technology, researchers gain access to many networked computers at the cost of a highly heterogeneous computing platform. Obviously, maintaining application versions for each resource type is tedious and troublesome, and results in a deploy-port-redeploy cycle. Further, different hardware and software setups on computational resources complicate the application development drastically. One never knows to which resource a job is submitted in a grid, and while it is possible to assist each job with a detailed list of hardware and software requirements, researchers are better left off with a virtual workspace environment that abstracts a real execution environment.

Hence, a virtual execution environment spanning the heterogeneous resource platform is essential in order to fully leverage the grid potential. From the view of applications, this would render a resource access uniform and thus the much easier ”compile once run anywhere” strategy; researchers can write their applications, compile them for the virtual machine and have them executed anywhere in the Grid.

B. Virtual Machines in a Grid Computing Context

Due to the renewed popularity of virtualization over the last few years, virtual machines are being developed for numerous purposes and therefore exist in many designs, each of them in many variants with individual characteristics. Despite the variety of designs, the underlying technology encompasses a number of properties beneficial for Grid Computing [4]:

1) Platform Independence: In a grid context, where it is inherently intrinsic to move around application code as freely as application data, it is highly profitable to enable applications to be executed anywhere in the grid. Virtual machines bridge the architectural boundaries of computational elements in a grid by raising the level of abstraction of a computer system, thus providing a uniform way for applications to interact with the system. Given a common virtual workspace environment, grid users are provided with a compile-once-run-anywhere solution.

Furthermore, a running virtual machine is not tied to a specific physical resource; it can be suspended, migrated to another resource and resumed from where it was suspended.

2) Host Security: To fully leverage the computational power of a grid platform, security is just as important as application portability. Today, most grid systems enforce security by means of user and resource authentication, a secure communication channel between them, and authorization in various forms. However, once access and authorization is granted, securing the host system from the application is left to the operating system.

Ideally, rather than handling the problems after system damage has occurred, harmful - intentional or not - grid applications should not be able to compromise a grid resource in the first place.

Virtual machines provide stronger security mechanisms than conventional operating systems, in that a malicious process running in an instance of a virtual machine is only capable of destroying the environment in which it runs, i.e. the virtual machine.

3) Application Security: Conversely to disallowing host system damage, other processes, local or running in other virtualized environments, should not be able to compromise the integrity of the processes in the virtual machine.

System resources, for instance the CPU and memory, of a virtual machine are always mapped to underlying physical resources by the virtualization software. The real resources are then multiplexed between any number of virtualized systems, giving the impression to each of the systems that they have exclusive access to a dedicated physical resource. Thus, grid jobs running in a virtual machine are isolated from other grid jobs running simultaneously in other virtual machines on the same host as well as possible local users of the resources.

4) Resource Management and Control: Virtual machines enable increased flexibility for resource management and control in terms of resource usage and site administration. First of all, the middleware code necessary for interacting with the Grid can be incorporated in the virtual machine, thus relieving the resource owner from installing and managing the grid software. Secondly, usage of physical resources like memory, disk, and CPU usage of a process is easily controlled with a virtual machine.

5) Performance: As a virtual machine architecture interposes a software layer between the traditional hardware and software layers, in which a possibly different instruction set is implemented and translated to the underlying native instruction set, performance is typically lost during the translation phase. Despite of recent advances in new virtualization and translation techniques, and the introduction of hardware-assisted capabilities, virtual machines usually introduce performance overhead and the goal remains achieving near-native performance only. The impact depends on system characteristics and the applications intended to run in the machine.

To summarize, virtual machines are an appealing technology for Grid Computing because they solve the conflict between the grid users at the one end of the system and resource providers at the other end. Grid users want exclusive access to as many resources as possible, as much control as possible, secure execution of their applications, and they want to use certain software and hardware setups.

At the other end, introducing virtual machines on resources enables resource owners to service several users at once, to isolate each application execution from other users of the system and from the host system itself, to provide a uniform execution environment, and managed code is easily incorporated in the virtual machine.

C. A Scientific Virtual Machine for Grid Computing

Virtualization can occur at many levels of a computer system and take numerous forms. Generally, as shown in Figure 2, virtual machines are divided in two main categories:

System virtual machines and process virtual machines, each branched in finer division based on whether the host and guest instruction sets are the same or different. Virtual machines with the same instruction set as the hardware they virtualize do exist in multiple grid projects as mentioned in Section VI. However, since full cross-platform portability is of major importance, we only consider *emulating* virtual machines, i.e. machines that execute another instruction set than the one executed by the underlying hardware.

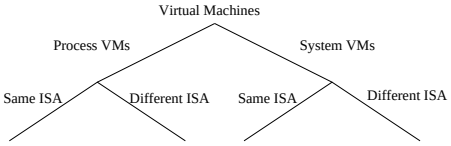

Fig. 2. Virtual machine taxonomy. Similar to Figure 13 in [9]

System virtual machines allow a host hardware platform to support multiple complete guest operating systems, all controlled by a virtual machine monitor and thus acting as a layer between the hardware and the operating systems. Process virtual machines operate at a higher level in that they virtualize a given platform for user applications. A detailed description of virtual machines can be found in [9].

The overall problem with system virtual machines that emulate the hardware for an entire system, including applications as well as an operating system, is the performance loss incurred by converting all guest system operations to equivalent host system operations, and the implementation complexity in developing a machine for every platform type, each capable of emulating an entire hardware environment for essentially all types of software.

Since the application domain in focus is scientific applications only, there is really no need for full-featured operating systems. As shown in Figure 3, process level virtual machines are simpler because they only execute individual processes, each interfaced to the hardware resources through a virtual instruction set and an Application Binary Interface.

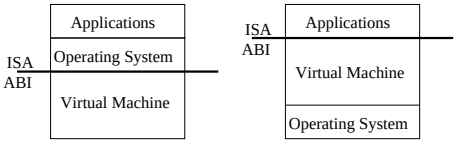

Fig. 3. System VMs (left) and Process VMs (right)

Using the process level virtual machine approach, the virtual machine is designed in accordance with a software development framework. Developing a virtual machine for which there is no corresponding underlying real machine may sound counterintuitive, but this approach has proved successful in several cases, best demonstrated by the power and usefulness of the Java Virtual Machine. Tailored to the Java programming language, it has provided a platform independent computing environment for many application domains, yet there is no commonly used real Java machine [1].

Similar to Java, applications for the SciBy VM are compiled into a platform independent byte code which can be executed on any device equipped with the virtual machine. However, applications are not tied to a specific programming language. As noted earlier, researchers should not be forced to rewrite their applications in order to use the virtual machine. Hence, we produce a compiler based on a standard ansi C compiler.

D. Enabling Limitations

While the outlined work at hand may seem comprehensive, especially the implementation burden with virtual machines for different architectures, there are a some important limitations that greatly simplify the project. Firstly, the implementation burden is lessened drastically by only giving support for running a single sequential application. Giving support for entire operating systems is much more complex in that it must support multiple users in a multi-process environment, and hardware resources such as networking, I/O, the graphics processor, and 'multimedia' components of currently used standard CPUs are also typically virtualized.

Secondly, a virtual machine allows fine-grained control over the actions taken by the code running in the machine. As mentioned in Section VI, many projects use sandbox mechanisms in which they by various means check all system instructions. The much simpler approach taken in this project is to simply disallow system calls. The rationale for this decision is that:

- scientific applications perform basic calculations only
- using a remote file access library, only files from the grid can be accessed
- all other kinds of I/O are not necessary for scientific applications and thus prohibited
- indispensable systems calls must be routed to the grid

III. Architectural Overview

The SciBy Virtual Machine is an abstract machine executing platform independent byte codes on a virtual CPU, either by translation to native machine code or by interpretation. However, in many aspects it is designed similarly to conventional architectures; it includes an Application Binary Interface, an Instruction Set Architecture, and is able to manipulate memory components. The only thing missing in defining the architecture is the hardware. As the VM is supposed to be run on a variety of grid resources, it must be designed to be as portable as possible, thereby supporting many different physical hardware architectures.

Based on the previous sections, the SciBy VM is designed to have 3 fundamental properties:

- Security
- Portability
- Performance

That said, all architectural decisions presented in the following sections rely solely on providing portability. Security is

[1]The Java VM has been implemented in hardware in the Sun PicoJava chips

obtained by isolation through virtualization, and performance is solely obtained by the use of optimized native libraries for the intended applications and taking advantage of the fact that scientific applications spend most of their time in these libraries. The byte code is as such not designed for performance. Therefore, the architectural decisions do not necessarily seek to minimize code density, minimize code size, reduce memory traffic, increase the average number of clock cycles per instruction, or other architectural evaluation measurements, but more for simplicity and portability.

A. Application Binary Interface

The SciBy VM ABI defines how compiled applications interface with the virtual machine, thus enabling platform independent byte codes to be executed without modification on the virtual CPU.

At the lowest level, the architecture defines the following machine types arranged in big endian order:

- 8-bit byte
- 16-, 32-, or 64-bit halfword
- 32-, 64-, or 128-bit word
- 64-, 128-, or 256-bit doubleword

In order to support many different architectures, the machine exists in multiple variations with different word sizes. Currently, most desktop computers are either 32- or 64-bit architectures, and it probably won't be long before we see desktop computers with 128-bit architectures. By letting the word size be user-defined, we capture most existing and near-future computers.

Fundamental primitive data types include, all in signed two's complement representation:

- 8-bit character
- integers (1 word)
- single-precision floating point (1 word)
- double-precision floating point (2 words)
- pointer (1 word)

The machine contains a register file of 16384 registers, all 1 word long. This number only serves as a value for having a potentially unlimited amount of registers. The reasons for this are twofold. First of all due to forward compatibility, since the virtual register usage has to be translated to native register usage, in which one cannot tell the upper limit on register numbers. So basically, in a virtual CPU, one should be sure to have more registers than the host system CPU. Currently, 16384 registers should be more than enough, but new architectures tend to have more and more registers. Secondly, for the intended applications, the authors believe that a register-based architecture will outperform a stack-based one[8]. Generally, registers have proved more successful than other types of internal storage and virtually every architecture designed in the last few decades uses a register architecture.

Register computers exist in 3 classes depending on where ALU instructions can access their operands, register-register architectures, register-memory architectures and memory-memory architectures. The majority of the computers shipped nowadays implement one of those classes in a 2- or 3-operand format. In order to capture as many computers as possible, the SciBy VM supports all of these variants in a 3-operand format, thereby including 2-operand format architectures in that the destination address is the same as one of the sources.

B. Instruction Set Architecture

One key element that separates the SciBy VM from conventional machines is the memory model: The machine defines a Harvard memory architecture with separate memory banks for data and instructions. The majority of conventional modern computers use a von Neumann architecture with a single memory segment for both instructions and data. These machines are generally more vulnerable to the well-known buffer overflow exploits and similar exploits derived from 'illegal' pointer arithmetic to executable memory segments. Furthermore, the machine will support hardware setups that have separate memory pathways, thus enabling simultaneous data and instruction fetches. All instructions are fetched from the instruction memory bank which is inaccessible for applications: All memory accesses from applications are directed to the data segment. The data memory segment is partitioned in a global memory section, a heap section for dynamically allocated structures, and a stack for storing local variables and function parameters.

1) Instruction Format: The instruction format is based on *byte codes* to simplify the instruction stream. The format is as follows: Each instruction starts with a one-byte *operation code* (opcode) followed by possibly more opcodes and ends with zero or more operands, see Figure 4. In this sense, the machine is a multi-opcode multi-address machine. Having only a single one-byte opcode limits the instruction set to only 256 different instructions, whereas multiple opcodes allows for nested instructions, thus increasing the number of instructions exponentially. A multi-address design is chosen to support more types of hardware.

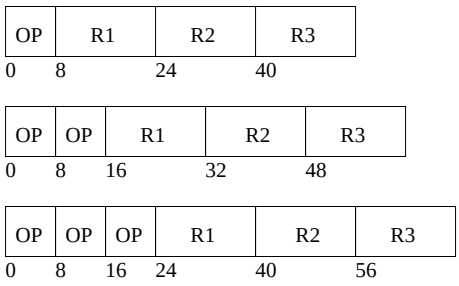

Fig. 4. Examples of various instruction formats on register operands.

2) Addressing Modes: Based on the popularity of addressing modes found in recent computers, we have selected 4 addressing modes for the SciBy VM, all listed below.

- Immediate addressing: The operand is an immediate, for instance MOV R1 4 which moves the number 4 to register 1.
- Displacement addressing: The operand is an offset and a register pointing to a base address, for instance ADD

R1 R1 4(R2) which adds to R1 the value found 4 words from the address pointed out by R2.

- Register addressing: Operand is a register, for instance MOV R1 R2
- Register indirect addressing: Address part is a register containing the address of an operand, for instance ADD R1, R1, (R2), which adds to R1 the value found at the address pointed out by R2.

3) Instruction Types: Since the machine defines a Harvard architecture, it is important to note that data movement is carried out by `LOAD` and `STORE` operations which operate on words in the data memory bank. `PUSH` and `POP` operations are available for accessing the stack.

Table I summarizes the most basic instructions available in the SciBy VM. Almost all operations are simple 3-address operations with operands, and they are chosen to be simple enough to be directly matched by native hardware operations.

Instruction group	Mnemonic
Moves	load, store
Stack	push, pop
Arithmetic	add, sub, mul, div, mod
Boolean	and, or, xor, not
Bitwise	and, or, shl, shr, ror, rol
Compare	tst, cmp
Control	halt, nop, jmp, jsr, ret, br, be_eq, br_lt, etc

TABLE I
BASIC INSTRUCTION SET OF THE SCIBY VM

While these instructions are found in virtually every computer, they exist in many different variations using various addressing modes for each operand. To accommodate this and assist the compiler as much as possible, the SciBy VM provides regularity by making the instruction set orthogonal on both operations, data types, and the addressing modes. For instance the 'add' operation exists in all 16 combinations of the 4 addressing modes on the two source registers for both integers and floating points. Thus, the encoding of an 'add' instruction on two immediate source operands takes up 1 byte for choosing arithmetic, 1 byte to select the 'add' on two immediates, 2 bytes to address one of the 16384 registers as destination register and then 16 bytes for each of the immediates, yielding a total instruction length of 36 bytes.

C. Libraries

In addition to the basic instruction set, the machine implements a number of basic libraries for standard operations like floating-point arithmetic and string manipulation. These are extensions to the virtual machine and are provided on an per-architecture basis as statically linked native libraries optimized for specific hardware.

As explained above, virtual machines introduce a performance overhead in the translation phase from virtual machine object code to the native hardware instructions of the underlying real machine. The all-important observation here is that scientific applications spend most of their running time executing 'scientific instructions' such as string operations, linear algebra, fast fourier transformations, or other library functions. Hence, by providing optimized native libraries, we can take advantage of the synergy between algorithms, the compiler translating them, and the hardware executing them.

Equipping the machine with native libraries for the most prevalent scientific algorithms and enabling future support for new libraries increases the number of potential instructions drastically. To address this problem, multiple opcodes allows for nested instructions as shown in Figure 5. The basic instructions are accessible using only one opcode, whereas a floating point operation is accessed using two opcodes, i.e. *FP_lib FP_sub R1 R2 R3*, and finally, if one wishes to use the `WFTA` instruction from the `FFT_2` library, 3 opcodes are necessary: *FFT_lib FFT_2 WFTA args*.

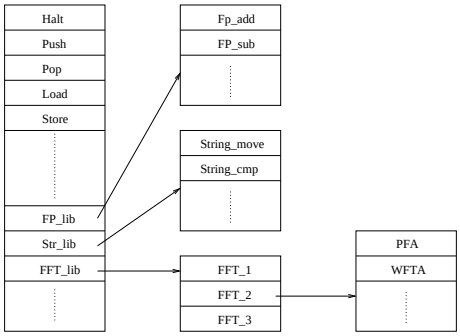

Fig. 5. Native libraries as extension to the instruction set

A special library is provided to enable file access. While most grid middlewares use a staging strategy that downloads all input files prior to the job execution and uploads output files afterwards, the MiG-RFA [1] library accesses files directly on the file server on an on-demand basis. Using this strategy, an application can start immediately, and only the needed fragments of the files it accesses are transferred.

Ending the discussion of the architecture, it is important to re-emphasize that all focus in this part of the machine is on portability. For instance, when evaluating the architecture, one might find that:

- Having a 3-operand instruction format may give unnecessarily large code size in some circumstances
- Studies may show that the displacement addressing mode is typically used to nearby addresses, thereby suggesting that these instructions only need a few bits for the operand
- Using register-register instructions may give unnecessarily high instruction count in some circumstances
- Using byte codes increases the code density
- Variable instruction encoding decreases performance

Designing an architecture includes a lot of trade-offs, and even though many of these issues are zeroed by the interpreter or translator, the proposed byte code is far from optimal by normal architecture metrics. However, the key point is that we target only a special type of applications on a very broad hardware platform.

IV. Compilation and Translation

While researchers do not need to rewrite their scientific applications for the SciBy VM, they do need to compile their application using a SciBy VM compiler that can translate the high level language code to the SciBy VM code. While developing a new compiler from scratch of course is a possibility, it is also a significant amount of work which may prove unprofitable since many compilers designed to be retargetable for new architectures already exist.

Generally, retargetable compilers are constructed using the same classical modular structure: A front end that parses the source file, and builds an intermediate representation, typically in the shape of a parse tree, used for machine-independent optimizations, and a back end that translates this parse tree to assembly code of the target machine.

When choosing between open source retargetable compilers, the set of possibilities quickly narrows down to only a few candidates: GCC and LCC. Despite the pros of being the most popular and widely used compiler with many supported source languages in the front end, GCC was primarily designed for 32-bit architectures, which greatly complicates the retargeting effort. LCC however, is a light-weight compiler, specifically designed to be easily retargetable to a new architecture.

Once compiled, a byte code file containing assembly instruction mnemonics is ready for execution in the virtual machine, either by interpretation or by translation, where instructions are mapped to the instruction set of the host machine using either a load-time or run-time translation strategy. Results remain to be seen, yet the authors believe that in case a translator is preferable to an interpreter, the best solution would be load-time translator, based on observations from scientific applications:

- their total running time is fairly long which means that the load-time penalty is easily amortized
- they contain a large number of tight loops where run-time translation is guaranteed to be inferior to load-time translation

V. Experiments

To test the proposed ideas, a prototype of the virtual machine has been developed, in the first stage as a simple interpreter implemented in C. There is no compiler yet, so all sample programs are hand-written in assembly code with the only goal of giving preliminary results that will show whether development of the complete machine can be justified.

The first test is a typical example of the scientific applications the machine targets: A Fast Fourier Transform (FFT). The program first computes 10 transforms on a vector of varying sizes, then checksums the transformed vector to verify the result. In order to test the performance of the virtual machine, the program is also implemented in C to get the native base line performance, and in Java to compare the results of the SciBy VM with an existing widely used virtual machine.

The C and SciBy VM programs make use of the fftw library[6], while the Java version uses an FFT algorithm from the SciMark suite[7]. Obviously, this test is highly unfair in disfavor of the Java version for several reasons. Firstly, the fftw library is well-known to give the best performance, and comparing hand-coded assembly with compiler-generated high-level language performance is a common pitfall. However, even though Java-wrappers for the fftw library exist, it is essential to put these comparisons in a grid context. If the grid resources were to run the scientific applications in Java Virtual Machine, the programmers - the grid users - would not be able to take advantage of the native libraries, since allowing external library calls breaks the security of the JVM. Thereby, the isolation level between the executing grid job and the host system is lost[2]. In the proposed virtual machine, these libraries are an integrated part of the machine, and using them is perfectly safe.

Vector size	Native	SciBy VM	Java
524288	1.535	1.483	7.444
1048576	3.284	3.273	19.174
2097152	6.561	6.656	41.757
4194304	14.249	14.398	93.960
8388608	29.209	29.309	204.589

TABLE II
Comparison of the performance of an FFT application on a 1.86 GHz Intel Pentium M processor, 2MB cache, 512 MB RAM

Vector size	Native	SciBy VM	Java
524288	0.879	0.874	4.867
1048576	1.857	1.884	10.739
2097152	3.307	3.253	23.520
4194304	6.318	6.354	50.751
8388608	13.045	12.837	110.323

TABLE III
Comparison of the performance of an FFT application on a dual core 2.2 GHz AMD Athlon 4200 64-bit, 512 kB cache per core, 4GB RAM

As shown in Table II the FFT application is run on the 3 machines using different vector size, $2^{19}, ..., 2^{23}$. The results show that the SciBy VM is on-par with native execution, and that the Java version is clearly outperformed.

Since the fftw library is multithreaded, we repeat the experiment on a dual core machine and on a quad dual-core machine. The results are shown in Table III and Table IV.

From these results it is clear that for this application there

[2]In fact there is a US Patent (#6862683) on a method to protect native libraries

Vector size	Native	SciBy VM	Java
524288	0.650	0.640	4.955
1048576	1.106	1.118	12.099
2097152	1.917	1.944	27.878
4194304	3.989	3.963	61.423
8388608	7.796	7.799	134.399

TABLE IV
Comparison of the performance of an FFT application on a quad dual-core Intel Xeon CPU, 1.60 GHz, 4MB cache per core, 8GB RAM

is no overhead in running it in the virtual machine. It has immediate support for multi-threaded libraries, and therefore the single-threaded Java version is even further outperformed on multi-core architectures.

VI. Related Work

GridBox [3] aims at providing a secure execution environment for grid applications by means of a sandbox environment and Access Control Lists. The execution environment is restricted by the *chroot* command which isolates each application in a separate file system space. In this space, all system calls are intercepted and checked against pre-defined Access Control Lists which specify a set of allowed and disallowed actions. In order to intercept all system calls transparently, the system is implemented as a shared library that gets preloaded into memory before the application executes. The drawback of the GridBox library is the requirement of a UNIX host system and application and it does not work with statically linked applications. Further, this kind of isolation can be opened if an intruder gains system privileges leaving the host system unprotected.

Secure Virtual Grid (SVGrid) [10] isolates each grid applications in its own instance of a Xen virtual machine whose file system and network access requests are forced to go through the privileged virtual machine monitor where the restrictions are checked. Since each grid virtual machine is securely isolated from the virtual machine monitor from which it is controlled, many levels of security has to be opened in order to compromise the host system, and the system has proved its effectiveness against several malicious software tests. The performance of the system is also above acceptable with a very low overhead. The only drawback is that while the model can be applied to other operating systems than Linux, it still makes use of platform-dependent virtualization software.

The MiG-SSS system [2] seeks to combine Public Resource Computing and Grid Computing by using sandbox technology in the form of a virtual machine. The project uses a generic Linux image customized to act as a grid resource and a screen saver that can start any type of virtual machine capable of booting an ISO image, for instance VMware player and VirtualBox. The virtual machine then boots the linux image which in turn retrieves a job from the Grid and executes the job in the isolated sandbox environment.

Java and the Microsoft Common Language Infrastructure are similar solutions trying to enable applications written in the Java programming language or the Microsoft .Net framework, respectively, to be used on different computer architectures without being rewritten. They both introduce an intermediate platform independent code format (Java byte code and the Common Intermediate Language respectively) executable by hardware-specific execution environments (the Java Virtual Machine and the Virtual Execution System respectively). While these solution have proved suitable for many application domains, performance problems and their requirement of a specific programming language class rarely used for scientific applications disqualifies the use of these virtual machines for this project.

VII. Conclusions and Future Work

Virtual machines can solve many problems related to using desktop computers for Grid Computing. Most importantly, for resource owners, they enforce security by means of isolation, and for researchers using the Grid, they provide a level of homogeneity that greatly simplifies application deployment in an extremely heterogeneous execution environment.

This paper presented the basic ideas behind the Scientific Byte Code Virtual Machine and proposed a virtual machine architecture specifically designed for executing scientific applications on any type of real hardware architecture. To this end, efficient native libraries for the most prevalent scientific software packages are an important issue that the authors believe will greatly minimize the performance penalty normally incurred by virtual machines.

An interpreter has been developed to give preliminary results that have shown to justify the ideas of the machine. The machine is on-par with native execution, and on the intended application types, it outperforms the Java virtual machines deployed in grid context.

After the proposed initial virtual machine has been implemented, several extension to the machine are planned, including threading support, debugging, profiling, an advanced library for a distributed shared memory model, and support for remote memory swapping.

References

[1] Rasmus Andersen and Brian Vinter, *Transparent remote file access in the minimum intrusion grid*, WETICE '05: Proceedings of the 14th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprise (Washington, DC, USA), IEEE Computer Society, 2005, pp. 311–318.

[2] ———, *Harvesting idle windows cpu cycles for grid computing*, GCA (Hamid R. Arabnia, ed.), CSREA Press, 2006, pp. 121–126.

[3] Evgueni Dodonov, Joelle Quaini Sousa, and Hélio Crestana Guardia, *Gridbox: securing hosts from malicious and greedy applications*, MGC '04: Proceedings of the 2nd workshop on Middleware for grid computing (New York, NY, USA), ACM Press, 2004, pp. 17–22.

[4] Renato J. Figueiredo, Peter A. Dinda, and José A. B. Fortes, *A case for grid computing on virtual machines*, ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing Systems (Washington, DC, USA), IEEE Computer Society, 2003.

[5] Ian Foster, *The grid: A new infrastructure for 21st century science*, Physics Today **55(2)** (2002), 42–47.

[6] Matteo Frigo and Steven G. Johnson, *The design and implementation of FFTW3*, Proceedings of the IEEE **93** (2005), no. 2, 216–231, special issue on "Program Generation, Optimization, and Platform Adaptation".

[7] Roldan Pozo and Bruce Miller, *Scimark 2.0*, http://math.nist.gov/scimark2/.

[8] Yunhe Shi, David Gregg, Andrew Beatty, and M. Anton Ertl, *Virtual machine showdown: stack versus registers*, VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments (New York, NY, USA), ACM, 2005, pp. 153–163.

[9] J.E. Smith and R. Nair, *Virtual machines: Versatile platforms for systems and processes*, Morgan Kaufmann, 2005.

[10] Xin Zhao, Kevin Borders, and Atul Prakash, *Svgrid: a secure virtual environment for untrusted grid applications*, MGC '05: Proceedings of the 3rd international workshop on Middleware for grid computing (New York, NY, USA), ACM Press, 2005, pp. 1–6.