Performance and Portability of the SciBy Virtual Machine

Rasmus Andersen and Brian Vinter
eScience Centre
University of Copenhagen
Copenhagen, Denmark
{rasmus,vinter}@diku.dk

***Abstract*—The Scientific Bytecode Virtual Machine is a virtual machine designed specifically for performance, security, and portability of scientific applications deployed in a Grid environment. The performance overhead normally incurred by virtual machines is mitigated using native optimized scientific libraries, security is obtained by sandboxing techniques. Lastly, by executing platform-independent bytecodes, the machine is highly portable.**

To evaluate the machine, we demonstrate several use-case scenarios from some of the intended application domains. Further, we show the ease of porting the machine and distributing its jobs to a variety of predominant architectures and compare the results with native execution.

***Keywords*-Grid Computing; Volunteer Computing; Virtual Machine; Scientific Applications**

I. Introduction

In this project, we try to combine strengths from Volunteer Computing and Grid Computing [1] to form a very powerful computing platform. To harness the compute power from this platform, and put in on tap for researchers with immediate needs for massive amounts of CPU-cycles, we have introduced the Scientific Bytecode Virtual Machine (SciBy VM) [2].

Designed specifically for scientific applications, we take advantage of some well-known facts about scientific applications to achieve close to optimal performance on a variety of hardware platforms. Secondly, by executing a platform-independent bytecode, the machine is higly portable, analogously to the Java Virtual Machine [3]. While JVM is not appropriate for high performance computing, it has proved extremely successful for a number of application domains due to the ease of portability. Finally, as security cannot be underestimated when allowing arbitrary code to execute on donated resources, the machine implements several layers of security to ensure not only the host machine, but also the integrity of the application.

In the previous paper [2], we detailed the architecture of the machine, and preliminary results justified further development. In this paper, we evaluate the performance of the machine on a series of sample applications from the intended domain, and demonstrate its portability. Further, we show how to take advantage of the embedded remote file access module, and how to deploy the machine in a stable and widely used Grid system.

II. Motivation

As a computing platform, Volunteer Computing has shown its potential with many successful projects utilizing the idle time from the donated resources. For instance, at the time of writing, the very popular Folding@Home project [4] operates at whapping 4.3 petaflop/s. In comparison, the Roadrunner supercomputer tops the supercomputer top 500 list with a peak performance of 1.1 petaflop/s.

A natural extension to the Volunteer Computing scheme is to 'gridify' it. In their current form, primarily based on the BOINC framework [5], these public resource computing projects are merely one-way systems; one can only donate resources. A resource has a pre-installed program that continuously fetches new *data* to execute, as opposed to a Grid, where resources continuously fetch new *programs*. The entry cost for a researcher to actually submit new programs to these Volunteer Computing systems is high enough to render them useless for smaller research projects in need of compute power [1].

In our perception, an important aspect of Grid Computing is to allow seamless access for researchers to the huge amount of connected resources, thus creating the illusion of unified access to one big supercomputer. Gaining widespread acceptance as an important computing platform depends on the ability of making the technology accessible to general exploitation, not only for computer specialists, but also for researchers from other areas, since they are the real end-users of the systems.

The need for compute power increases rapidly, primarily driven by eScience, i.e. computationally intensive scientific simulations with enormous amounts of data. As the simulations get more and more accurate, the requirements to the compute platform increase equivalently.

'Gridifying' a Volunteer Computing system, i.e. not only making it easy for the public to donate their idle resources, but also enabling researchers to utilize these public resources, gives a whole new dimension to a Grid. However, the two systems differ significantly on two aspects: Security and portability. Firstly, a resource in a Volunteer Computing system only hosts one single application that keeps getting new data, while a resource in Grid system hosts arbitrary untrusted

[1]According to the BOINC website, http://boinc.berkeley.edu/trac/wiki/BoincIntro, it takes 3 man-months to port an existing application to the BOINC framework and about $5000 for hardware used for project management

applications. Hence, there is a substantial difference in the security level to ensure the host system. Secondly, while a 'standard' Grid is typically composed of Linux/Unix desktop computers, clusters, and supercomputers from different trusted compute centres, a volunteer computing grid drastically increases the heterogeneity of the system. For instance, the client statistics from the Folding@Home project reveals that the biggest contributors are NVIDIA GPUs and PlayStation 3's, markedly trailing ATI GPUs and Windows machines. Linux and Mac machines are barely noticeable. Obviously, porting a single Volunteer Computing application to these architectures is considerably easier than enabling arbitrary programs to run on any architecture.

With the abundance of new powerful architectures in personal computers and devices, we now face many radically different CPU architectures, GPU architectures, operating systems, software environments, and specific limitations to memory, network, and disk usage. Therefore, enabling execution of arbitrary code on connected resources requires a lot of work in specifying an execution environment. Ultimately, before non-computer specialists can adopt this computing platform, it is necessary to hide these complexities of the Grid system, and free the applications from all architectural details of the computational resources connected to the Grid.

Specifying or standardizing an execution environment - an executable code format and runtime environment - and ensuring the safety of the host machine, is by no means an easy task. However, as outlined in [2], virtual machines can help bridge the architectural boundaries by raising the abstraction level from the underlying hardware, thus enabling moving around application code as freely as application data. The Java Virtual Machine Specification [6] and the Common Intermediate Language [7] from MicroSoft are two examples that have proved very useful with many different virtual machine implementations. Once compiled into the byte code of the respective standards, an application can run on any of these machines. The drawback of these systems is the requirement of using a specialized programming language rarely used in scientific applications, and the performance drawback from executing byte codes instead of native machine code.

The approach taken in this project is to use a hybrid model, in which a completely platform-independent - and thus highly portable - bytecode is augmented with the ability of calling native libraries to improve performance. With respect to security, virtualization reduces the problem of requiring donators to trust arbitrary programs to only trust the virtual machine that executes the programs. In effect, virtualization increases the application integrity and eases the resource management and control.

III. Related Work

During the last decade, virtual machines [8] have been revived and used for many different purposes. VMware [9], Xen [10], and VirtualBox [11] are some of the most popular *system* virtual machines. With the ability of hosting multiple complete guest operating systems, applications can be isolated safely, and recent progress - even with hardware support - has provided near-native speed. However, they are typically tied to a limited set of platform architectures and difficult to port.

Application virtual machines, for instance JVM and .Net VM, only support running a single application. Applications are expressed as portable intermediate language representations [6], [7] and executed by abstract virtual machines [3], [12]. As opposed to system virtual machines, these machines are highly portable, yet their performance drawback hinders them from being used for high performance computing.

Virtual machines are just one way of providing a `sandbox`, i.e. a confined environment in which a host system allows untrusted 3rd-party code to execute without compromising the host. Native code sandboxes allows a native application to execute under a well-defined set of constraints, all expressed in native code themselves. Since the introduction by Wahbe [13], the literature contains many variants for different architectures. The vx32 [14] would be of primary interest for a Volunteer System since it has minimal intrusion on the host system, thus lowering the workload for the people trying to donate their resources. Where many similar systems require modifications to the host system, for instance kernel modifications, special privileges and permissions, vx32 runs on unmodified host systems. Applications just need to be linked to a user-level library, which then sandboxes the application in a secure execution environment. vx32 is limited to x86 architectures.

An upcoming interesting sandbox is Google's Native Client [15]. Aimed at browser-based applications, the system tries to run compute-intensive applications in the address space of the browser at native speed. When an application is sent to a browser, a Native Client plugin loads the Native Client container, which is a sandbox containing native libraries enabling the application to execute at native speed. Like vx32, Native Client is limited to x86 architectures, and since the workload required to express all types of security restrictions in native machine code is quite substantial, porting to other architectures is unlikely to happen within foreseeable future.

IV. The SciBy Virtual Machine

In the Sciby VM, we combine intermediate bytecode in a virtual machine with native libraries to achieve near-native performance on many different architectures. To our knowledge, this is the first research project that introduces this type of hybrid machine.

From a user perspective, the SciBy Virtual Machine [2] is an abstract virtual machine capable of executing platform-independent bytecodes corresponding to a single sequential application. Users will need to use a designated compiler to translate their source code into the bytecode, which is then sent to and executed by an interpreter on any available resource. The compiler is based on `gcc` [16], thus taking benefit of the many source languages accepted by the front end of the compiler; only the back end has been modified to target the SciBy architecture.

For deployment as a virtual machine for scientific applications in a merged Grid and Volunteer Computing system, the machine is designed solely for portability, host system security, and performance.

A. Security

Host system security is ensured by a virtual machine to isolate untrusted code in a sandbox. Typically, this type of software based fault isolation [13] focuses on disallowing unsafe instructions access to memory outside the sandbox, illegal instructions, privileged instructions, etc. In SciBy VM, we made the deliberate choice of disallowing system calls, including all types of I/O, altogether. Thus, we only allow instructions that perform transformations on data, control flow instructions, and data movement instructions. Dedicated for scientific applications, there is really no need for system calls, and the only type of I/O necessary, is access to input files and output files; this is achieved using the Remote File Access library, presented in Section IV-D. Indispensable system calls will be routed back to the Grid for execution.

A typical sandbox feature, which the SciBy VM also implements, is a Harvard memory model with separate segments for data and code to ensure correct access to data, and preventing jumps to instructions outside the code area. Using this model, the machine is less vulnerable to typical exploits derived from 'illegal' pointer arithmetic to other executable memory segments.

B. Portability

Portability is obtained by designing a completely platform-independent bytecode and by virtualizing a very broad hardware platform. The instruction set includes all typical instructions for data transformation, control flow, and data movement. To capture most of the physical computer platforms, it is designed as an orthogonal multi-opcode multi-address set of instructions in a 3-operand format, thereby permitting easy translation to 2-operand architectures. Addressing modes include immediate addressing, displacement addressing, register addressing, and indirect addressing.

Virtualization occurs at many levels in various computer subsystems. It typically provides an illusion of hardware configurations that are not physically available, for instance virtual memory which gives each process the illusion of having exclusive access to the entire address space of the machine. With the SciBy VM, much focus is placed on being forward compatible by virtualizing hardware setups of the future. Most notably, as the tendency goes towards more and more registers, the machine provides a virtually unlimited number of registers. Just to provide a number for the compiler, it is set to 16384 128-bit registers. Further, as there is now a shift from 32 bit towards 64 bit architectures, the next step probably being 128 bit, the SciBy VM supports all these word sizes.

C. Performance

Performance is obtained by augmenting the instruction categories with one essential category: Instructions that can call external native library functions. Executing bytecodes in a virtual machine will incur performance overhead, but the key point here is to utilize the fact that scientific applications spend most of their time in these libraries. The characteristics of these applications, for instance bioinformatics, high-energy physics, or image processing analysis, is a small but very time-consuming code size. Typically, these application set up input and output files, and then enter some dense loop structures in which time-consuming calls to an external library are continuously made. In the SciBy VM, the surrounding code is replaced by the portable bytecode, but with support for calling native optimized libraries in order to achieve near native performance, this is illustrated in Figure 1.

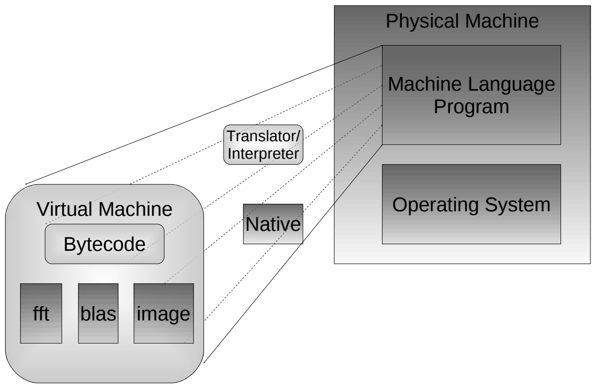

Fig. 1. An untrusted application converted to bytecode, running in VM with embedded scientific native libraries.

Obviously, native libraries are not portable and a version of the machine must be equipped with statically linked libraries for every architecture, and made available for people willing to participate. However, with the success of a library follows ports to other architectures, and it is a simple and small task to embed a library in the machine.

D. Remote File Access

As mentioned above, the SciBy VM does not have access to a file system on the host machine. Instead of transferring input and output files to the resource executing a job, the files remain in the home directory of the job owner at a file server, and are then accessed remotely using the Remote File Access Library [17] (RFA) in the Minimum intrusion Grid, described below. Imposed as a user-level library between an application and the host operating system, it intercepts all file access routines and directs them to the server on which the file in question resides.

This type of on-demand remote file access enables the resource to limit its file retrieval to a set of file fragments holding the needed data, thus only downloading needed data and only uploading modified data. Especially for scientific applications, this technique can be a big advantage compared to the traditional staging technique. On initialization, a job can start right away; it does not need to wait for potentially huge files to be transferred from a file server. And often, only the first parts, or scattered fragments of a file are really needed. For instance, high-energy physics applications typically find their data in huge database input files, but the amount of blocks read from the file is relatively small. Similarly, output files are written remotely on demand.

Using prefetching to hide the latency, and a strategy for dynamically adjusting block sizes in the data transfer that adapts to the application's data access pattern, and mitigates

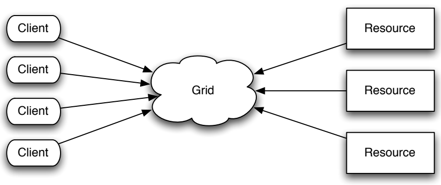

Fig. 2. The abstract MiG model

fluctuating network conditions, the file access library has proved to have very good performance. Compared to staging techniques, tests have shown that speedups of above one thousand are well within the realm of possibilities. In optimal scenarios, it can even be superior to local file access, for instance with CPU-bound applications, where the block transfer times, and more importantly, the system calls for the file access, can be completely hidden by computations.

V. Deployment

While both the SciBy virtual machine and the remote file access library were independently developed and applicable in many systems, they are targeted at, and deployed in, the Minimum intrusion Grid.

A. Minimum intrusion Grid

MiG [18], [19] is a stand-alone approach to Grid that does not depend on any existing systems, i.e. it is a completely separate platform for Grid computing. The philosophy behind the MiG is to provide a Grid infrastructure that imposes as few requirements on users and resources as possible.

Driven by the aim of minimum intrusion on participating parties, the idea is to ensure that users only need a signed X.509 certificate, trusted by Grid, and a web browser that supports HTTP and HTTPS. A resource only needs to create an MiG user on the system and to support inbound ssh and outbound HTTPS. Initially, the resource must register to the MiG system using a certificate.

By keeping the Grid system disjoint from both users and resources, as shown in Figure 2, this model allows the Grid system to appear as a centralized black-box to both users and resources, and all upgrades and trouble shooting can be performed locally within the Grid without intervention from neither users nor resource administrators. Thus, all functionality is placed in a physical Grid system.

The basic functionality in MiG starts with users submitting jobs to MiG and resources sending requests for jobs. A resource then receives an appropriate job from MiG, executes the job, and sends the result to MiG that can inform the user of the job completion. Thus, MiG provides full anonymity; users and resources interact only with MiG, never with each other.

B. Porting SciBy VM and Distributing jobs

The process of porting the SciBy VM to other architectures is straightforward. The interpreter is written in strict ANSI-C making it highly portable. In all experiments presented in the next section, the source files were copied to the test machine and compiled. The applications were compiled into bytecode files on a single machine and then transferred for immediate execution. As regards the libraries, they were all available for the tested platforms. So once built on the particular platform, the library is embedded into the virtual machine for the platform in question.

People willing to donate can then download a version of the machine matching their platform from the website, and just like the native code sandbox [20] provided by the MiG, screen saver software will contact the Grid for a job on activation. During download, a resource description file is generated stating resource features such as which libraries are available. It is then up to the scheduler to match the resource description with job description files, as shown below.

Researchers in need of compute power compile their source code into the SciBy VM bytecode, for instance *fourier.bin*, which needs to be uploaded to one's home directory in MiG. Next, submitting the job to the MiG can be done either through the website using an X.509 certificate or by downloading scripts to automate the process. The job description, in MiG known as a mRSL file, used for submitting the job to the MiG is:

```
::EXECUTE::
scibyvm fourier.bin
::NOTIFY::
jabber: my_id@jabber.org
::INPUTFILES::
fourier.bin
::OUTPUTFILES::
outputimage.pgm
::CPUTIME::
900
::ARCHITECTURE::
SciBy-VM
::RUNTIMEENVIRONMENT::
libfourier
```

The MiG has already run the native code sandbox volunteer computing project, based on existing virtual machines, for a long time, and has proved flexible enough to deal with resources behind NAT routers, and with such dynamic and transient compute elements as screen saver resources.

VI. Evaluation

Generally, virtualization incurs an extra performance overhead. The survey by Domingues et al. [21] shows that for the popular system level virtual machines (VMWare Player, VirtualBox, Qemu, VirtualPC), the performance impact is about 10-15% for CPU-bound applications, and up to 35% for I/O-bound applications. The performance goal for all virtual machines remains achieving near-native speed, so for evaluation, we perform a series of sample applications from the intended application domain, and compare the results to native speed.

The Fast Fast Fourier Transform (FFT) is an obvious choice for evaluating the SciBy VM, since it is a fundamental kernel in so many scientific applications, for instance data compression, fluid dynamics, seismic imaging, image processing, computer tomography, data filtering, spectral analysis,

Vector size	2^{19}	2^{20}	2^{21}	2^{22}	2^{23}
Pentium M					
Native	1.535	3.284	6.561	14.249	29.209
SciBy VM	1.483	3.273	6.656	14.398	29.309
AMD Athlon					
Native	0.879	1.857	3.307	6.318	13.045
SciBy VM	0.874	1.884	3.253	6.354	12.837
Intel Xeon					
Native	0.650	1.106	1.917	3.989	7.796
SciBy VM	0.640	1.118	1.944	3.963	7.799

TABLE I
COMPARISON OF SCIBY VM AND NATIVE PERFORMANCE ON AN FFT APPLICATION ON 3 DIFFERENT ARCHITECTURES. ALL RESULTS ARE IN SECONDS

and digital signal processing. The core of these applications is the necessity of computing Fourier transforms, and the performance of this type of application relies heavily on the routines available for performing the transforms. The results in all experiments are the average of 3 consective runs, all measured in seconds using the Linux `time` command.

In this test, our application uses the `fftw` [22] library to perform 10 transforms on a vector of varying sizes, $2^{19}, ..., 2^{23}$, and then checksums the result vector to verify the result. The application is compared to a native C-version on 3 different machines:

- A 1.86 GHz Intel Pentium M, 2 MB cache, 512 MB RAM
- A dual core 2.2 GHz AMD Athlon 4200 64-bit, 512 kB cache per core, 4 GB RAM
- A dual quad core Intel Xeon, 1.60 GHz, 4 MB cache per core, 8 GB RAM

As shown in Table I, the performance of the SciBy VM is on par with native execution, and when taking advantage of a multi-threaded library, it can utilize multi-core architectures.

A. Image Processing

Image processing is widely used in many scientific applications, such as medical imaging, computer graphics rendering, sensing and detection systems, and in general, over the last few years it is becoming a topic of interest for a broad scientific community. Using `fourier` [23], a portable image processing and analysis library, we write a sample application that applies a series of transformations, for instance `gaussian adaptive smoothing` filter, on a raw 2592x1944 pixel `pgm` image. We then run the application in 4 different setups:

- Native: Run natively, with the image in the local file system
- SciBy VM: Run inside the SciBy VM with the image in the local file system (suspending the local file access restriction)
- Native + RFA: Run natively, using the Remote File Access library to access the file 200 km away (standard 5 Mbps ADSL-line).
- SciBy VM + RFA: Run inside the SciBy VM using the RFA library as above.

These tests are performed on 4 architectures:

- An Intel Core 2 1.86 GHz processor, 2GB memory, running 32 bit Ubuntu linux,
- An AMD Athlon 64-bit processor 3000+, 1GB memory, running Debian-amd64
- A 3.2 GHz PowerPC 64-bit processor, 2 hardware threads, from the Cell Broadband Engine, running 32-bit Yellow Dog Linux[2]
- an Intel Core 2 2.4 GHz processor, 4GB memory, running Mac OSX.

TABLE II
RESULTS FROM THE FOURIER APPLICATION ON A SINGLE-CORE HOST MACHINE. THE TRANSFER TIME OF THE IMAGE DATA FROM THE GRID, 46 SECONDS, IS NOT INCLUDED IN THE NON-RFA RESULTS

Arch.	Native	SciBy	Native+RFA	SciBy+RFA
Core 2	96.36	96.41	142.87	142.90
AMD	78.93	78.89	124.75	124.01
PPC	166.18	164.25	213.76	212.21
Mac	58.72	59.05	105.55	106.01

TABLE III
RESULTS FROM THE `DIPLIB` APPLICATION ON AN 8-CORE HOST MACHINE. THE IMAGE TRANSFER TIME, 46 SECONDS, IS NOT INCLUDED IN THE NON-RFA RESULTS

Cores	Native	SciBy	Native+RFA	SciBy+RFA
1	101.53	101.34	147.47	147.80
2	51.36	51.94	97.76	97.60
4	27.74	27.63	73.23	73.54
8	15.14	15.23	61.87	61.90

Table II shows that there is no overhead when running inside the virtual machine in any of the setups. Since the entire file is used in an unbalanced fashion, there is no gain from using the RFA library. The time to transfer the image using `curl` and `lighttpd` was 46.293 seconds, which is exactly the difference between the runs with and without RFA. Thus, a staging technique would be equal to accessing the file remotely.

Next, to illustrate utilization of a multi-core architecture, we use the `diplib` [24] image processing library, which is multi-threaded. In this test, we apply the very compute-intensive second order derivative Laplace filter on the image, and execute on the 8-core Intel Xeon 1.60 GHz with 8 GB memory. Using the `taskset` command, the experiment is carried out using 1,2,4, and 8 cores.

From the results in Table III, we can once again conclude that there is no overhead from using the virtual machine. And, there is immediate support for multi-core utilization. Again, since the `diplib` image reads the entire image before it processes it, there is no gain from the remote file access library.

To wrap up image processing performance tests, a final example uses the `ffmpeg` [25] library to decode frames from a video file. Each decoded frame is then transformed using the `Imlib2` [26] image library. In this case, we just a apply a simple blurring effect to every frame. Since the `ffmpeg` balances I/O and processing, i.e. it consecutively reads data for a single frame, and then decodes it, the RFA library can take advantage of the prefetching, thus having every frame available in the instant it is needed. Therefore, as shown in

[2]The `fourier` image library does not utilize the SPEs in the Cell BE.

TABLE IV
RESULTS FROM THE VIDEO PROCESSING, INCLUDING TRANSFER TIME OF THE IMAGE FROM THE GRID

Arch.	Native	SciBy	Native+RFA	SciBy+RFA
Core 2	376.58	377.11	261.81	262.01
AMD	260.24	260.13	192.89	193.11

TABLE V
RESULTS FROM THE BLAS BENCHMARK

Arch.	Native	SciBy VM
Core 2	38.466	38.211
AMD	46.340	46.870
PPC	43.513	43.662
Mac	36.344	36.492

IV, using the RFA library is faster than using local file access. The test is performed on the Intel and AMD computers only.

B. Basic Linear Algebra Subroutines

`BLAS` (Basic Linear Algebra Subroutines)is widely used for high-performance computing and benchmarking. `BLAS` is a set of efficient routines for most of the basic vector and matrix operations. They are widely used as the basis for other high quality linear algebra software packages, for instance `lapack` and `linpack`. In this test, we perform a series of operations from the ATLAS [27] and GotoBLAS [28] libraries on a 500 by 500 matrix. It is all computed in memory, so there is no file access. The results displayed in V once again show that the SciBy VM achieves near native speed.

VII. CONCLUSIONS AND FUTURE WORK

Merging Volunteer Computing and Grid Computing systems is a big step towards realizing one of the initial ideas of Grid Computing, namely harnessing idle CPU power from all types of computer resources and putting them on tap for world-wide sharing. To harness the compute power from an ever-increasing farm of architectures and use it for scientific research, 3 fundamental obstacles have been identified and dealt with: portability, security, and performance. To this end, we have presented and evaluated the SciBy Virtual Machine.

Security is an all-important topic when utilizing other people's computers. The virtual machine ensures host system integrity by isolating the application in a secure execution environment.

Bytecodes and a virtual machine executing them provides platform-independence at the cost of performance. In this paper we have introduced a hybrid model, where the machine executes mobile bytecodes and uses native optimized libraries to mitigate the performance drawback. Thus, the model applies best to applications that spend most of their time in the libraries, which is the case for scientific applications.

Using typical scientific libraries, we have evaluated the machine's performance and found it to be on par with native execution. For other types of applications, the machine will currently perform poorly, but lessons learned from similar machines will surely alleviate this problem.

REFERENCES

[1] I. Foster, "The grid: A new infrastructure for 21st century science," *Physics Today*, vol. 55(2), pp. 42–47, 2002.

[2] R. Andersen and B. Vinter, "The scientific byte code virtual machine," in *GCA*, H. R. Arabnia, Ed. CSREA Press, 2008, pp. 175–181.

[3] T. Lindholm and F. Yellin, *The Java(TM) Virtual Machine Specification (2nd Edition)*. Prentice Hall PTR, April 1999. [Online]. Available: http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20\&amp;path=ASIN/0201432943

[4] Folding@home, "Folding@home distributed computing," http://folding.stanford.edu/.

[5] D. P. Anderson, "Boinc: A system for public-resource computing and storage," in *GRID '04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 4–10.

[6] J. Gosling, B. Joy, G. Steele, and G. Bracha, *Java Language Specification*, 3rd ed. Addison-Wesley Professional, July 2005. [Online]. Available: http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20\&amp;path=ASIN/0321246780

[7] E. International, *Standard ECMA-335 - Common Language Infrastructure (CLI)*, 4th ed., June 2006. [Online]. Available: http://www.ecma-international.org/publications/standards/Ecma-335.htm

[8] W. Mcewan, "Virtual machine technologies and their application," in *Accessed 14 March 2003 www.ddj.com/documents/s=882/ddj0008f/0008f.htm*, 2002, pp. 55–62.

[9] "Vmware," http://www.vmware.com/.

[10] I. Pratt, K. Fraser, S. Hand, C. Limpach, A. Warfield, D. Magenheimer, J. Nakajima, and A. Mallick, "Xen 3.0 and the art of virtualization," in *Proceedings of Linux Symposium 2005*, July 2005.

[11] J. Watson, "Virtualbox: bits and bytes masquerading as machines," *Linux J.*, vol. 2008, no. 166, p. 1, 2008.

[12] E. Meijer, R. Wa, and J. Gough, "Microsoft clr overview."

[13] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *In Proceedings of the 14th ACM Symposium on Operating Systems Principles*, 1993, pp. 203–216.

[14] B. Ford and R. Cox, "Vx32: lightweight user-level sandboxing on the x86," in *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2008, pp. 293–306.

[15] B. Yee and D. Sehr, "Native client: A sandbox for portable, untrusted x86 native code," Tech. Rep., 2001.

[16] "The gnu compiler collection," http://gcc.gnu.org.

[17] R. Andersen and B. Vinter, "Direct application access to grid storage: Research articles," *Concurr. Comput. : Pract. Exper.*, vol. 19, no. 9, pp. 1287–1298, 2007.

[18] B. Vinter, "The Architecture of the Minimum intrusion Grid (MiG)," in *Communicating Process Architectures 2005*, sep 2005, pp. –.

[19] H. H. Karlsen and B. Vinter, "Minimum intrusion grid - the simple model," in *WETICE '05: Proceedings of the 14th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprise*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 305–310.

[20] R. Andersen and B. Vinter, "Harvesting idle windows cpu cycles for grid computing," in *GCA*, H. R. Arabnia, Ed. CSREA Press, 2006, pp. 121–126.

[21] P. Domingues, F. Araujo, and L. Silva, "Evaluating the performance and intrusiveness of virtual machines for desktop grid computing," in *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–8.

[22] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005, special issue on "Program Generation, Optimization, and Platform Adaptation".

[23] M. E. Celebi, "fourier," http://mac.softpedia.com/get/Development/Libraries/Fourier.shtml.

[24] M. van Ginkel, "diplib," http://www.diplib.org.

[25] F. Bellard, "ffmpeg," http://www.ffmpeg.org/.

[26] C. Haitzler, "Imlib2," http://docs.enlightenment.org/api/imlib2/html/.

[27] R. C. Whaley and A. Petitet, "Minimizing development and maintenance costs in supporting persistently optimized BLAS," *Software: Practice and Experience*, vol. 35, no. 2, pp. 101–121, February 2005.

[28] "Gotoblas," http://www.tacc.utexas.edu/resources/software/.