Virtualization technologies: An overview

Fernando Terroso Sáenz, Ricardo Fernández-Pascual y José M. García[1]

Abstract— **Virtualization spans through many aspects of computer architecture. Over the years, virtual machines have been researched and built by operating system developers, compiler developers, language designers, and hardware designers. And recently, it has become again a hot topic in the computer research panorama.**

In this paper we show the complexity and importance of this field in the current computer research world. We consider that, to have a wide vision of the general computer research world, it is crucial understand and learn some aspects of how virtualization works, its characteristics and the main solutions for some of its problems.

Therefore, we present a global vision of the state of the art in the computer virtualization field.

Key words— **paravirtualization, full virtualization, virtualization overview, virtual machines**

I. Introduction

BROADLY speaking, the term *virtualization* describes the separation of a resource or request for a service from the underlying physical delivery of that service [1]. So, when a system is virtualized, its interface and all resources visible through that interface are mapped onto the interface and resources of a real system actually implementing them. Formally, virtualization involves the construction of an isomorphism that maps a virtual *guest* system to a real *host* [2]. Figure 1 illustrates the isomorphism that maps the guest state to the host state (function V in the figure). For a sequence of operations, *e*, that modifies the guest's state, there is a corresponding sequence of operations *e'* in the host that performs an equivalent modification to the host's state.

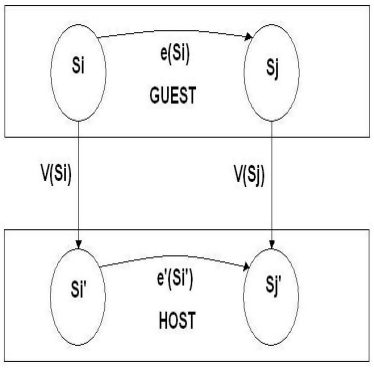

Fig. 1. Isomorphism between a guest and a host system

The above definition may suggest that virtualization is the same as abstraction, but this is not always true. Virtualization differs from abstraction in that virtualization does not necessarily hide details [3]; the level of detail in a virtual system is often the same as that in the underlying real system.

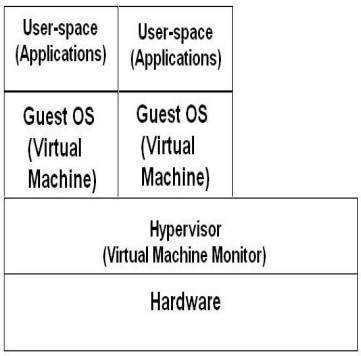

Fig. 2. Overview of a virtual machine environment

When the concept of virtualization is applied to an entire machine (CPU, Memory and I/O devices) then we talk of a *System Virtual Machine* (VM). A virtual machine is implemented by adding a software layer to an execution platform to give it the appearance of a different platform, or for that matter, to give the appearance of multiple platforms. A virtual machine may have an operating system (OS), an instruction set, or both, differing from those implemented on the underlying real hardware [4]. Usually, the software added to give the appearance of a different platform is called *Virtual Machine Monitor* (VMM): a virtual machine is the environment created by the Virtual Machine Monitor [2]. Figure 2 gives a basic idea of a virtual machine environment.

On the other hand, *Process Virtual Machines* have the purpose of providing a virtualized environment to application programs to improve their portability. This is made emulating a guest ISA and I/O features.

But there are other levels in a system architecture where virtualization is useful. This is the case of the so called *High-Level Language Virtual Machines* (HLL VM) which provide an abstract platform which is more convenient for implementing certain computer languages. This platform may provide an abstract ISA, operating system abstractions (eg. via a library), memory management and other services.

In this paper we offer an historical perspective of the virtualization field and the past and current reasons that have made virtualization a hot topic in the research panorama. Furthermore, we explain the most important virtualization problems and the different techniques that resolve them. Each of the techniques are illustrated with real software or hardware products that implement it.

The rest of the paper is organized as follows: In section II we expose an historical perspective of why virtualization is, nowadays, an important research

[1]Departamento de Ingenieria y Tecnologia de Computadores, Facultad de Informatica. Universidad de Murcia, e-mail: {fernando, r.fernandez, jmgarcia}@ditec.um.es

topic. In section III the main virtualization techniques are explained. In section IV an HLL VMs overview is offered and finally in section V we expose some conclusions of this work.

II. Why use virtualization?

A. An historical approach

With the earliest machines, before the 1950's, software and hardware were developed very close each other: software was written specifically for the hardware. This included operating systems, compilers, and application programs. This was not a problem insofar as the computer systems were used in small communities, especially by scientists and engineers, and the basic concepts of the stored program computer were still evolving.

But this state of the affairs changed quickly when machines began to be used as individual tools, and used for more general tasks, because user communities grew, operating systems became more complex, and the number of application programs rapidly expanded; hence rewriting and distributing software for each new computer system became a major burden [4]. In other words, it was necessary that software had a new characteristic: *portability*. It was not until the IBM System/360 family in the early 1960s, that the importance of full software compatibility was fully recognized and articulated. With the recognition that architecture interface and implementation could be separated, common machine architectures were established. The need to separate software and hardware brougth the concept of the Instruction Set Architecture (ISA) [5].

But that was not the only interface standardized. At a higher level of abstraction, operating systems began to both shield application programs from hardware specifications and to protect running applications and user data, supporting controlled sharing.

With the two interfaces mentioned, software experimented a great evolution and could be made much more flexible and portable than would otherwise be possible. However, incompatibilities between ISAs and operating systems limited software portability to the group of platforms that conform to the same or very similar standards as the ones for which the software was developed, specially in the case of binary portability. Software was restricted to a specific operating system (or family of operating systems) and a specific ISA. This was an unpleasant restriction when newer, cheaper and better platforms were available but the software being used required an older platform.

The basic problem was that software was associated with a *physical* machine with specific characteristics. To achieve real software portability, a new layer in the computer system hierarchy was needed. That new layer had the purpose of giving an execution platform the appearance of a different platform, or for that matter, to give the appearance of multiple different platforms, at the same time. Since then, we could talk about *virtual machines* instead of real machines.

Furthermore, virtual machines allowed to improve backward compatibility of new computer systems. Because a virtual machine was capable of hiding the physical characteristics of a system, it was possible to run on a new machine application programs and operating systems that were developed for older systems (with its own physical characteristics and ISA). This was an important advance in the administration of data centers.

B. Current state

Comparing modern computer systems with the ones from a decade ago, we see a spectacular improvement both in performance and storage capacity. Even more, a simple modern laptop has more powerful resources than a data center from four decades ago. The problem is that we risk wasting those hardware resources. This is because, nowadays, it is quite usual to have a system with just a single OS which is only used by a single user. So, to avoid wasting resources one can use the system to do more than one thing at once. And this is the point where virtualization enters the scene of modern data centers, because they have the same utilization problem than we have explained for a laptop, among others. You can improve the resource utilization of a system if you share it among different OSes, by means of virtualization.

Virtualization techniques can be applied to many of the IT infrastructure layers, including networks, storage, laptop or server hardware, operating systems and applications.

Virtualization has allowed easier administration of data centers thanks to the isolation between logical and physical infrastructure. It is also possible to isolate in a safe way different service environments on the same physical machine (each environment runs on a different virtual machine) with the security benefits that this implies. For example, if one of the environments on a machine is hacked and it is necessary to restart it, that action will not affect the other environments.

Virtualization also eases system updates. If different services are running on virtual machines then system administrators can start up a new virtual machine with the new version of the service, test it and when everything has been tested, stop the virtual machine with the old version of the service and set the virtual machine with the new version as the default service. Doing that, users see a much shorter period of downtime.

Virtualization techniques are also useful for multiprocessors. They allow to divide the resources of a multiprocessor over different partitions, each of them running independently of the others.

As system virtual machines can improve portability of OSes, High-Level Language Virtual Machines can also improve portability of application programs. With this kind of virtualization, it is not necessary to have a binary version of an application program

for each OS where we want to run it. The virtual machine is responsible for providing a virtual ISA to the application and hide the details of the host operating system.

III. Virtual machines

Popek and Goldberg's 1974 paper [2] establishes three essential characteristics for system software to be considered a VMM, and be able to provide a correct and practical virtualization environment:

- Fidelity. Software on the VMM executes identically to its execution on hardware.
- Performance. A majority of guest instructions are executed by the hardware without intervention of the VMM.
- Safety. The VMM manages all hardware resources.

A more precise criterion for good VM performance would be that most instructions were executed with similar performance as in a physical machine.

The main challenges to be overcome for achieving the requisites mentioned above are [6]:

- Virtual machines must be isolated from each other.
- Virtual machines must to be capable to share the physical resources of the host machine.
- The performance overhead introduced by virtualization should be small.

In this section we show the different technologies that solve these items and we also describe software and hardware products that implement them.

A. Virtual machine isolation

With correct isolation, each OS and application program which runs on a virtual machine has the same behaviour as it would have if it ran in native mode despite sharing the system with other VMs. Being able to store and recover the state of a VM is crucial when there are several VMs running on the same host machine and it is necessary to carry out context switches between VMs. A way to model the state of a virtual machine is proposed in [3]. In this proposal the state of a virtual machine is modeled as a 4-tuple, S=(E, M, P, R), where E refers to the executable storage, M refers to the model of operation, P is the program counter, and R is a pair denoting the physical location and the size of the virtual memory space.

For example, both *Intel VT-X* [7] and *AMD-V* [8] technologies, the two extensions to the IA-32 architecture for supporting virtualization, have data structures that encapsulate all the information needed to capture the state of a virtual machine or to resume a virtual machine.

In the case of Intel VT-X technology, the name of this structure is *Virtual Machine Control Structure* (VMCS). The VMCS includes a *guest-state area* and a *host-state area*, each of which contains fields corresponding to different components of processor state [9]. The guest-state area of the VMCS is used to contain elements of the state of virtual CPU associated with that VMCS. This includes those IA-32 registers that manage operation of the processor, such as the segment registers, and many others. The guest-state area does not contain fields corresponding to registers that can be saved and loaded by the VMM itself (e.g., the general-purpose registers). The host-state area contains a number of fields that specify the instructions and events that cause VM exits.

In the case of AMD-V technology, an analogous data structure called *Virtual Machine Control Block* (VMCB) is used. The VMCB consists of two areas too. The first area contains control bits including the intercept enable mask. This mask determines which conditions cause a VM exit. The second area maintains the guest state. This save state area preserves the segment registers and most of the virtual memory, but not the general purpose or floating point registers, like with Intel VT-X [10].

The two solutions shown in the last paragraph are hardware solutions, or so-called *VM-assists* because its main purpose is to help to achieve an efficient virtualization environment in an easier way. But these solutions still need a software complement like a software VMM which was introduced in section I.

A.1 Security

Security requires virtual machine isolation. When a VM is active, it should not be able to execute any privileged instructions on the host machine which would let it to either access data of other VMs or gain total control of the machine.

For example, the privileged instruction SPT (Set CPU Timer) of the IBM System/370 [11] replaces the CPU interval timer with the contents of a location in memory. If a malicious program had privileges to execute that instruction, it would be able to replace periodically the CPU interval timer, avoiding the periodic intervention of the VMM, and get the total control of the machine. Hence, the VMM needs to have some control over which instructions can be executed directly by the guest systems. To do this, both guest OS and application programs are executed on a non-privileged mode in the host machine. This way, the possibility that a malicious application program could get the control of the system by means of privileged instructions is avoided.

To do this, IBM Power 5 [12] systems have a privileged state of the processor, called *hypervisor mode*, introduced in the Power4 processors. The processor must be in this state in order to be able to execute privileged instructions or to have write access to some of the processor system registers, such as the register that defines the location and size of the hardware page table associated with the partition. Hypervisor mode is accessed via a hypervisor call function (hcall), which is generated by the operating system kernel running in a partition. Hypervisor mode allows for a mode of operation that is required for various system functions where logical partition

integrity and security are required. When required, the Hypervisor validates that the partition has ownership of the resources it is attempting to access, such as processor, memory, or I/O; and then completes the function on behalf of the guest.

On the other hand, both AMD-V and Intel VT-X introduce two new modes of operation. One, called *VMX root operation* in Intel VT-X and *host mode* in AMD-V, is largely similar to its function in a normal IA-32 processor without virtualization technology, the main difference is the inclusion of a set of new instructions for entering and exiting to this mode of operation. The other mode, called *VMX non-root operation* in Intel VT-X and *guest mode* in AMD-V, is limited in some aspects from the behavior on a normal processor. The limitations ensure that critical shared resources are kept under the control of a VMM running in VMM root operation/host mode.

However, executing the guest OS in a non-privileged mode implies that each privileged instruction that the guest OS wants to execute must to be *trapped-and-emulated* by the VMM. The VMM will intercept the execution of the privileged instruction and perform the appropriate actions to simulate the results of the instruction without affecting other VMs.

There are two possible solutions to handle the execution of privileged instructions and the consistent processing by the VMM: *full virtualization* and *paravirtualization*.

Full virtualization provides total emulation of the underlying physical system and creates a complete virtual platform in which the guest OS can be executed without modifications.

In contrast, paravirtualization presents each VM with an abstraction of the hardware that is similar but not identical to the underlying physical hardware. Paravirtualization requires modifications to the guest OSes that are running on the VMs [13]. An interesting fact in this technology is that the guest machines are aware of the fact that they are running in a virtualized environment. Paravirtualization achieves performance closer to non-virtualized hardware than full virtualization because there is less emulation overhead [14].

There is a number of VMMs that implement full virtualization: VMwares's ESX Server [15], KVM [16] and Virtual Box [17], among others. Xen [18] and the VMMs in IBM iSeries, pSeries and zSeries take the paravirtualization approach.

B. Resource sharing

When different VMs are running on the same physical machine, they need to share the hardware resources while maintaining isolation. In this section we show the different solutions to achieve a efficient resource virtualization for memory and I/O devices.

B.1 Memory virtualization

In a typical virtual environment we distinguish three types of memory [3]:

- Virtual memory: virtual memory of the guest OS.
- Real memory: memory which is *regarded* by the guest OS as the physical memory.
- Physical memory: the physical memory of the host machine.

The addition of another level in the memory hierarchy requires additional mechanisms for memory management because the total size of *real* memory of all the guests can be bigger than the actual physical memory on the system.

Full virtualization solves this problem with architected page tables that are used in most ISAs, including IA-32 and IBM/370, using a structure called *shadow page tables* to keep the virtual-to-physical mapping for each VM. To make this work, the page table pointer register is virtualized.

For instance, the approach explained above is taken by VMwares's ESX Server. Furthermore, ESX Server implements a technique called *transparent page sharing* which stores the pages with identical content in physical locations which are shared by all the VMs with read-only privileges. As soon as any VM tries to modify a shared page, it gets its own private copy. This approach is completely transparent to the guest OS.

Using paravirtualization the solution is easier thanks to the collaboration between the guest OS and the VMM. This way, the guest OS indicates to the VMM whenever it wants to modify the page tables and then the VMM performs the appropriate actions.

For example, Xen registers guest OS page tables directly with the *memory-management unit* (MMU), and restricts guest OSes to read-only access. Page table updates are passed to Xen via a hypercall. This is one of the advantages of paravirtualization, at the cost of having to modify the guest OS.

B.2 I/O virtualization

Virtualizing I/O devices is complicated by the fact that the number of different devices of a given type is often large and continuously growing. The techniques to share I/O devices have been developed since the early days of time sharing with the appearance the IBM VM/370 [11].

Like memory virtualization, I/O virtualization is transparent to the guest system when using full virtualization. This means that all I/O requests executed by the VM are trapped by the VMM and redirected to the specific device after modifying it (or not) to follow the specifications of the particular specific physical device.

One example of the solution explained above is found in VMwares's ESX Server. With this product, the first way to virtualize an I/O device is to emulate the device in the *VMMonitor* which runs natively on the hardware. If the device to be virtualized already has a physical counterpart on the host, the job of emulating is simply one of converting the parameters in some *virtual device interface* (VDI) into parameters

of the actual *hardware device interface* (HDI). If the requested device is not natively supported by VM-Monitor but is supported by the host OS, the request is converted into a host OS call.

Paravirtualization can achieve better performance typically because the VM and the VMM collaborate. The guest OS sends an I/O request directly to the VMM (or to a special VM with access rights to the physical devices) which analyzes it and performs the appropriate actions to satisfy the guest OS demand.

The above solution is implemented by Xen which provides a special VM called *domain 0* which is the unique virtual machine running on the Xen hypervisor that has rights to access physical I/O resources as well as to interact with the other virtual machines running on the system [18]. Everytime a VM wants to access some I/O device, it opens an *event channel* with the domain 0 VM, that allows it to make requests via asynchronous inter-domain interrupts in the Xen hypervisor.

Finally, it is interesting to show how the I/O devices are handled in IBM Power5 system which also provides partitioning. In that platform, the system allows that each peripheral component interconnect (PCI) slot in the system can be individually assigned to a VM. The hypervisor ensures that each logical partition can access only the PCI slots assigned to it. The IBM Power5 uses a component, which runs in its own VM, called *Virtual I/O Server* (VIOS) whose function is to provide virtual devices to be used by other virtual machines. Moreover, to reduce the complexity of the hypervisor, the support for I/O adapters is delegated to the operating system running in each logical partition. This eliminates the need for updates to the hypervisor to support new I/O devices. Hence, a virtual machine can be configured either to use direct access to a device or to access that device though VIOS in a similar way as accessing a device though domain 0 in Xen.

B.3 Multiprocessor Virtualization

In a multiprocessor system resource sharing is more complex than in an uniprocessor one. A multiprocessor can be partitioned so that multiple applications can simultaneously exploit the resources of the system. The I/O virtualization techniques described above essentially perform resources multiplexing in time. Multiprocessor systems provide a new dimension, that of multiplexing resources in space.

There are two basic ways to perform multiprocessor partitioning: *physical partitioning* and *logical partitioning*. In physical partitioning each image uses resources, processors in particular, that are physically distinct from the resources used by the other OS images. With the latter, images share some of the physical resources, usually in a time-multiplexed manner.

Logical partitioning is more flexible and needs additional mechanisms to provide the needed services to share resources in a safe and efficient way.

The IBM Power5 implements the two types of partition, both logical and physical. In this system, each partition (called a *logical partition*, LPAR) running on the system can be either a *dedicated* or a *shared* partition [12] and views processors as virtual processors. The virtual processor of a dedicated processor partition has a physical processor allocated to it, while the virtual processor of a shared processor partition shares the physical processors of a shared processor pool with virtual processors of other shared processor partitions.

C. Performance overhead

Hardware and software virtualization techniques suffer different overheads. While software virtualization requires careful engineering to ensure efficient execution of guest kernel code, hardware virtualization delivers near native speed for anything that avoids certain operations, like starting an I/O operation which requires the intervention of the VMM.

In this sense, paravirtualization improves the execution time of the *modified* guest OS and application programs, because this technique allows an interaction between the guest and the VMM through which the guest can notify the VMM when a privileged operation needs to be executed.

IV. High level languages virtual machines

The primary goal of a High-Level Language Virtual Machine is to provide an abstract ISA which is free of quirks and requirements of any specific hardware platform. Because it is not designed for a real processor, it is a called *virtual* ISA (V-ISA). This virtual ISA can be designed so that it reflects important features of a specific high-level language (HLL) or a class of HLLs. Furthermore, a V-ISA considers data aspects are at least at the same level of importance as the instructions. For example, these ISAs usually have richer type systems than traditional ISAs, sometimes very similar to the type system of the high level language which is being implemented.

HLL VMs usually also provide higher level runtime services to the applications than traditional ISAs. These services include garbage collection, data persistence (serialization), threading, exception handling and others depending on the languages being implemented.

Using a HLL VM, applications can be distributed in a V-ISA form. At execution time on a HLL VM, a VM loader is invoked and converts the program into a form that is dependent on the virtual machine implementation. Finally the VM interprets and/or translates from the V-ISA to the host ISA.

This increases portability of application programs and eases the development of the compiler for that language. Also, using the same HLL-VM to implement several languages enables code sharing between the compilers of each language since all languages will benefit from the same optimizations performed by the virtual machine, usually by means of *just in time* (JIT) compilation [19]. Using a virtual machine to execute a program instead of compiling it

directly to native code also enables new opportunities to improve performance, because a JIT compiler has more precise information about the actual execution environment and the dynamic behavior of the application than an ahead of time compiler. For example, JIT compilers usually gather profile information to decide which methods should be compiled with the highest optimization level and to perform speculative inlining.

Virtual ISAs are also useful even when no virtual machine is used to perform the execution either as a portable binary distribution format [20] or as part of a standard compilation workflow [21].

The *Pascal P-Code Virtual Machine* [22], the *Java Virtual Machine* (JVM) [23] and the *Common Language Runtime* (CLR) [19] are three examples of HLL VMs.

V. Conclusion and future work

Virtualization provides separation between the use of a resource or a service and the underlying physical layer. Virtual machines apply this concept to computing platforms to improve flexibility, backward compatibility, resource utilization and administration of data centers

Historically, virtualization has offered backward compatibility of computer systems and, nowadays, it offers some features that make the work of data center administrators easier. These benefits are the main reason why virtualization is, again, an important hot topic in computer engineering.

The different techniques used to achieve a correct and efficient virtualization can be divided, on the one hand, into hardware and software solutions and, on the other hand, into paravirtualization and full virtualization proposals.

These techniques can be used to achieve strong isolation between VMs, to supply security avoiding access from a VM to the data of another VM and preventing that any guest system from gaining total control of the physical machine. Furthermore, a virtualization environment should allow resource sharing between VMs running on that environment, and this includes memory, I/O devices and processors.

Virtualization techniques can be used to implement HLL VMs, which are useful to ease compiler development, increase application portability and provide a number of runtime services to the applications.

Finally, there are many open topics of interest in vitualization. As future work, we want to mention the following: improvement of resource management for virtual machines, designing hardware to ease the implementation of virtual machines (both system virtual machines and HLL VMs), using virtualization for heterogeneous component abstraction (in heterogeneous CMPs, for example), using virtualization to achieve global system optimization and designing new memory models better suited for virtualization than current ones.

VI. Acknowledgements

This work has been jointly supported by the Fundación Seneca under grant 00001/CS/2007, and the Spanish MEC under grant "Consolider Ingenio-2010 CSD2006-00046", and also by the EU FP6 NoE HiPEAC IST-004408. Fernando Terroso Sáenz is also supported by a research grant from the Fundación Séneca.

References

[1] "Virtualization Overview," Tech. Rep., www.wmware.com, 2006.

[2] G. J. Popek and R. P. Goldberd, "Formal requeriments for Virtualizable Third-Generation Architectures," *Communications of the ACM*, 1974.

[3] J. E. Smith and R. Nair, *Virtual Machines. Versatile plataforms for systems and processes*, Morgan Kaufmann Publishers, 2005.

[4] J. E. Smith and R. Nair, "An Overview of Virtual Machine Architectures," copyright by Elsevier Science, 2003.

[5] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A quantitative Approach*, Morgan Kaufmann Publishers, 4 edition, 2007.

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," *Proc. 19th ACM Symp. on Operating System Principles*, 2003.

[7] Intel, "Intel Vanderpool Technology for IA-32 Processors (VT-x) Preliminary Specification," 2005.

[8] AMD, "AMD64 Architecture Programmer's Manual Volume 2: System Programming," 2007.

[9] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig, "Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization," *Intel Technology Journal*, vol. 10, 2006.

[10] M. Myers and S. Youndt, "An Introduction to Hardware-Assisted Virtual Machine (HVM) Rootkits," http://crucialsecurity.com/, 2007.

[11] P.H. Gum, "System/370 Extended Architecture: Facilities for Virtual Machines," *IBM Journal Res. Development*, vol. 27, no. 6, 1983.

[12] W. J. Armstrong, R. L. Arndt, D. C. Boutcher, R. G. Kovacs, D. Larson, K. A. Lucke, N. Nayar, and R. C. Swanberg, "Advanced virtualization capabilities of POWER 5 systems," *IBM Journal Res. and Dev.*, vol. 49, no. 4/5, 2005.

[13] T. Abels, P. Dhawan, and B. Chandrasekaran, *An overview of Xen virtualization*, pp. 109–111, Dell Power Solutions, 2005.

[14] J. S. Reuben, "A Survey on Virtual Machine Security," *TKK T-110.5290 Seminar on Network Security*, 2007.

[15] S. Devine, E. Bugnion, and M. Rosenblum, "Virtualization system including a virtual machine monitor for a computer with a segmented architecture," *US Patent 63971242*, 1998.

[16] Qumranet Inc, *KVM: Kernel-Based Virtualization Machine. White Paper*, 2006.

[17] *The VirtualBox architecture*, http://www.virtualbox.org/wiki, 2008.

[18] The Xen Project, *Xen Architecture Overview*, 1.2 edition, 2008.

[19] Ecma, "Common Language Infrastructure (CLI)," *Standar Ecma-335*, 2006.

[20] M. Cornero, R. Costa, R. Fernández-Pascual, A. Ornstein, and E. Rohou, "An Experiemental Environment Validating the Suitability of CLI as an Effective Deployment Format for Embedded Systems," *Proc. of the 2008 International Conference on High Performance Embedded Architectures Compilers (HiPEAC-2008)*, 2008.

[21] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," *Proc. of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, 2004.

[22] K.V. Nori, U. Ammann, K. Jensen, H. H. Nageli, and C. Jacobi, "The Pascal P-Compiler: Implementation Notes," Tech. Rep., Institut fur Informatik ETH, Zurich, 1975.

[23] J. Gosling, B. Joy, and G. Steele, *The Java Lenguage Specification*, Addison-Wesley, 1996.