# VAMOS: Virtualization Aware Middleware

Abel Gordon* Muli Ben-Yehuda†* Dennis Filimonov† Maor Dahan†
abelg@il.ibm.com muli@cs.technion.ac.il sdenf@mail.technion.ac.il smaorda@t2.technion.ac.il
*IBM Research—Haifa †Technion—Israel Institute of Technology

## Abstract

Machine virtualization is undoubtedly useful, but does not come cheap. The performance cost of virtualization, for I/O intensive workloads in particular, can be heavy. Common approaches to solving the I/O virtualization overhead focus on the I/O stack, thereby missing optimization opportunities in the overall stack. We propose VAMOS, a novel software architecture for middleware, which runs middleware modules at the hypervisor level. VAMOS reduces I/O virtualization overhead by cutting down on the overall number of guest/hypervisor switches for I/O intensive workloads. Middleware code can be adapted to VAMOS at only a modest cost, by exploiting existing modular design and abstraction layers. Applying VAMOS to a database workload improved its performance by up to 32%.

## 1 Introduction

Machine virtualization provides many benefits: it enables server consolidation, makes it possible to run legacy environments on new platforms, and helps simplify system management. However, it also degrades the performance of common workloads, particularly if they are I/O intensive workloads.

Extensive research has been carried out in order to reduce the virtualization (including I/O virtualization) performance penalty, both at the hardware and software layers. However, these works were mainly focused on improving the interaction between the hardware [12, 15, 22], the hypervisor [4, 16, 17, 18] and the operating system [9, 19, 21], avoiding changes at the application layer.

While virtualization is quickly becoming ubiquitous, applications remain oblivious to the changes in the underlying platform. Software architects and designers still use the same principles and models that used to apply to non-virtualized systems for building their applications. We argue that changes at the application layer are inevitable: the shift to machine virtualization everywhere requires that we adapt our applications to cooperate with the virtualization layer.

Adapting a nearly infinite number of applications is not feasible. However, in practice most server applications depend on one or more middleware layers, such as a database server, a web server, or an application server. By adapting the middleware to virtualized platforms, we can indirectly adapt many applications and regain lost performance. We need to re-think the way we build middleware. Most middleware have a modular design that already provides abstraction layers for operating system services, such as memory management, disk or network I/O, and CPU scheduling. We can exploit this to allow middleware software modules to cooperate with the hypervisor or run at host level, without necessitating a rewrite of the entire middleware.

This novel architecture model raises new opportunities for performance optimizations at the middleware layer, which do not conflict with today's optimization at lower layers. For example, the middleware can interact with the hypervisor I/O sub-systems using interfaces or protocols defined at the application level. Instead of using a para-virtual block [4, 17], network [4, 17], file system [9] or socket [19] interfaces, a database server could use a para-virtual SQL interface while a web server could use a para-virtual HTTP interface. By using high level interfaces we could remove the cycles spent in the hypervisor emulating the virtual hardware and we could also decrease the number of transitions between the hypervisor and the guest, thus improving both I/O throughput and latency [19].

Virtual appliances and Platform as a Service (PaaS) clouds are natural places to apply this new architecture model, primarily because the middleware layer there is under the control of the provider, and the adaptations will be transparent to the customer's applications. We can extend the model for Infrastructure as a Service (IaaS)

clouds, implementing the adaptations at the hypervisor level and exposing them as optional accelerators to the virtual machines. Assuming the customers decide to install required extensions into the virtual machines image, as they might install para-virtual drivers such as VMware-tools or virtio drivers [17], they will benefit from the improvements.

The main contribution in this work is VAMOS, a novel architecture for middleware running in virtual environments. Using VAMOS middleware cooperates with the hypervisor to improve the overall application performance. A proof of concept of VAMOS shows up to 32% I/O performance improvements for database workloads when compared with standard approach to I/O virtualization.

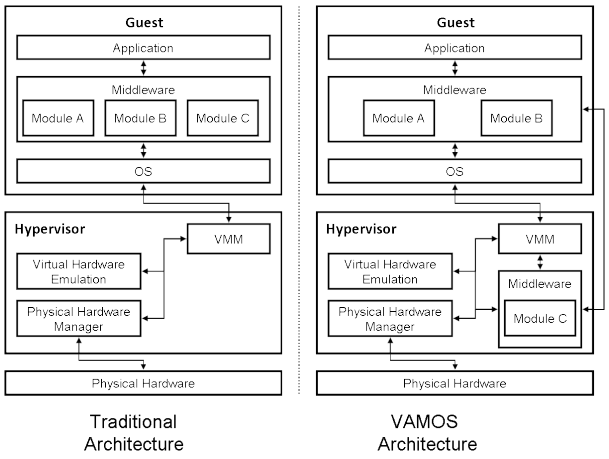


Figure 1: Comparing architecture models

## 2 VAMOS Architecture

In this section we present VAMOS, an architecture model that allows cooperation between the hypervisor and the middleware, improving I/O performance. First, we explain the causes for the virtualization overhead and then we describe the VAMOS architecture and its advantages.

### 2.1 Causes of Virtualization Overhead

It is well known that virtualization has a significant overhead [1, 5]. When we run a workload in a virtual machine, it does not get exclusive access to all the hardware resources, even when we use hardware virtualization support such as Intel VMX or AMD SVM. Usually, the VM runs directly on top of the physical CPU, however, when a sensitive instruction is executed, the CPU transfers the control to the hypervisor for handling. During this handling period, the guest is temporary interrupted and the hypervisor consumes precious cycles handling the sensitive instruction, which might require emulating virtual hardware. In addition, the transitions have a fixed cost caused by the CPU switching between the guest and the hypervisor contexts, and variable cost caused by the hypervisor polluting the guest cache [5, 8].

I/O intensive applications suffer the most [18] because they cause relatively many guest/host transitions. To get rid of the virtualization overhead, we need to reduce the number of transitions and/or reduce the handling cost.

### 2.2 Proposed Architecture

The number of abstraction layers required to access the hardware resources and the absence of cooperation between the hypervisor and the application cause many transitions to the hypervisor context. We can solve these problems running part of the middleware in the hypervisor context. Using this approach, the middleware can cooperate with the hypervisor and obtain direct access to the physical resources, bypassing the virtualization layer and mitigating part of the overhead. Figure 1 compares the traditional software architecture model with the VAMOS architecture. However, for VAMOS to be feasible, we need:

1. An isolated runtime environment at the hypervisor level which is capable of executing middleware code and providing the necessary services to access physical resources, such as network devices or disk drives.

2. A communication channel between the middleware running in the guest, the middleware running in the host and the hypervisor.

3. A methodology for adapting existing middleware code at a reasonable cost.

Today's hypervisors, such as KVM or Xen, or hosted hypervisors, already provide a fully operational operating system to manage the physical hardware. We can take advantage of this fact and run middleware modules as additional processes or runtime libraries in the host. The cooperation between the middleware and hypervisor can be implemented using regular mechanisms to communicate between user-space processes or kernel modules. In addition, the middleware can access physical resources via the host OS services, such as file systems, network interfaces, and virtual memory. For the guest to host communication, we can exploit the channels implemented to run para-virtualized guests [4, 17]. Alternatively, we can use more efficient channels, based on polling and side core invocation techniques [2, 8, 11, 13].

Machine virtualization typically requires isolating the physical resources from untrusted virtual machines. If

we treat the host side middleware modules as trusted entities, we can allow these modules to access physical resources. However, such middleware modules might contain security vulnerabilities. Thus, to minimize the effects of potential client attacks on the host, the hypervisor should restrict the privileges of those modules. This can be achieved using plugin-isolation techniques such as Vx32 [7] or Native Client [23].

While we have the appropriate environment to run part of middleware at the hypervisor level, we still need a methodology to adapt existing code. Modular design [14, 20] and abstraction layers [10] are well known techniques implemented in most if not all middleware products to decouple different components and to abstract OS services. For example, as shown in Figure 2, MySQL abstracts and implements the I/O logic into pluggable storage engines. Another example is Apache Tomcat, which encapsulates the logic responsible for handling the communication with the clients into independent modules, called connectors. JVMs usually abstract OS services to have a single cross platform core implementation and easy to maintain platform dependent modules [3]. We can take advantage of the modular design and run part of the middleware at the hypervisor context. The following criteria characterize the modules which will improve the application performance when moved to the hypervisor:

1. Modules which interact directly with system resources, generating many transitions to the hypervisor context and requiring emulation of virtual hardware.

2. Modules which can be easily re-factored into a client side running in the guest and a server side running in the host.

3. Modules which do not share state with other components, avoiding data sharing and synchronization between the guest and the hypervisor.

4. All the internal state used by the server side can be discarded on live migration, checkpoint or restore operations. In other words, internal state on the server side is only used as a cache.

Other characteristics might also be added to the previous list, such as clients capable of caching or batching requests, to reduce the number of transitions.

## 2.3 Applicability

Middleware commonly uses guest OS services to perform I/O operations, forcing the guest OS I/O stack to access the relevant virtual resources. Each of these accesses causes a transition to the hypervisor context, where the hypervisor spends prestigious cycles emulating the virtual hardware.

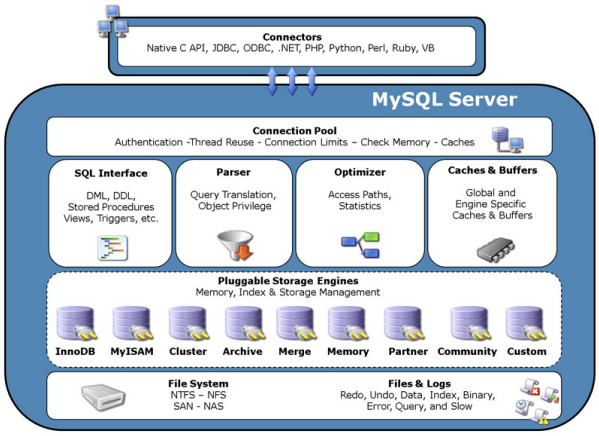


Figure 2: MySQL Architecture

By running the middleware modules responsible for I/O operations at the hypervisor level, we can reduce the number of transitions between the guest and the hypervisor as well as part of the virtual hardware emulation. Instead of switching on every low-level event such as packet sent or disk block written, we can switch when higher-level events (e.g., an SQL query) occur. Since higher-level events typically cause many lower-level events, we end up switching less often, with all of the resulting benefits. Another advantage of the proposed approach is that we end up running less operating system code in guest context (e.g., less file system code or networking code), which also consumes CPU cycles. For example, as we show in Section 3, a database server can run the module responsible for storing and retrieving the persisted data at the hypervisor level and use the host OS file system services to directly access the physical disk. A Web Server or Application Server can run the module responsible for communicating with the clients at the hypervisor level and use the host OS network services to access directly the physical network.

# 3 Proof of Concept

We implemented a proof of concept of VAMOS for the MySQL database server, running on top of the KVM hypervisor. As can be seen in Figure 2, within MySQL storage engines are responsible for disk I/O. We adapted the default storage engine, MyISAM, to access persistent data using the host OS file system. MyISAM was selected based on the criteria defined in Section 2.2.

The KVM hypervisor is implemented as a Linux kernel module that extends the kernel with hypervisor ca-

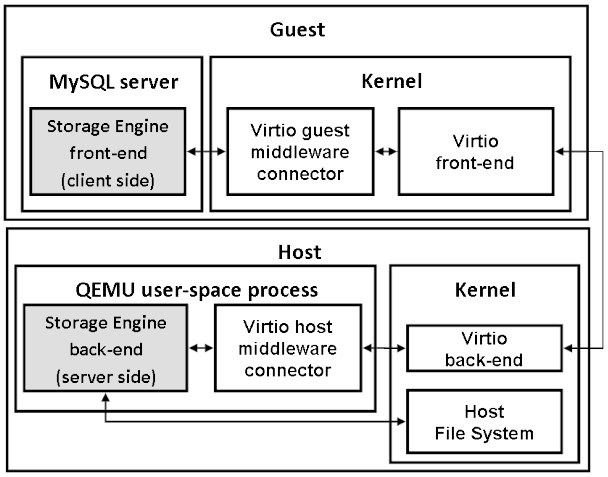


Figure 3: Virtualization Aware MySQL

pabilities, driven by a QEMU user process. KVM virtual machines are therefore represented as processes in the Linux host OS. We took advantage of this asymmetric model and moved the MyISAM storage engine into a shared library loaded by QEMU, which has direct access to host resources such as local file systems, network interfaces, and virtual memory.

The MyISAM client side module running in the guest delegates requests to the server side module running in the host. Communication between the client and the server is done using virtio [17]. To reduce the number of transitions from the guest to hypervisor context, we optimized the MyISAM client module to batch multiple requests into a single batch. This new architecture and the data-flow path are shown in Figure 3.

Using this model the MyISAM storage engine performs the I/O operations directly on the host, skipping the virtual hardware emulation, the guest OS VFS, file system, and block device, shortening overall code path lengths and reducing the number of transitions.

We analyzed MySQL performance by measuring the time needed to insert rows of different sizes to a database table. We used the following configurations: MySQL with emulated drivers (baseline), MySQL with para-virtualized drivers, VAMOS-MySQL without request batching, and VAMOS-MySQL with request batching. Figure 4 shows the results for different workloads. The Y-axis shows the normalized runtime improvement compared to the baseline. The X-axis shows the size of the database rows inserted for each workload. We can see that VAMOS improves performance for all the workloads. However, for workloads with row sizes up to 16KB, using only para-virtualized drivers we can obtain a better improvement. For sizes starting from 32KB VAMOS does better than para-virtualized drivers.

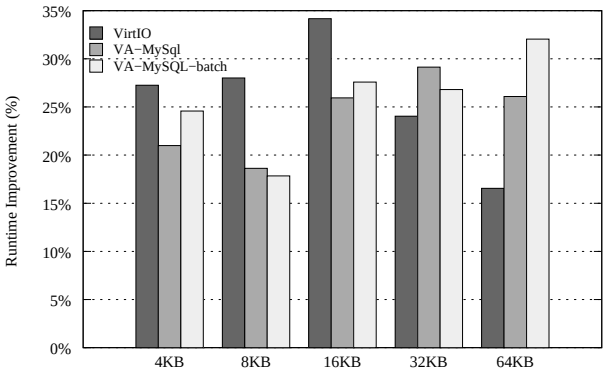


Figure 4: VAMOS MySQL runtime improvement for different workloads

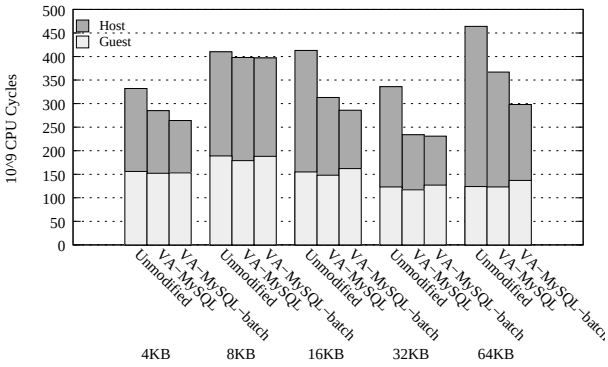


Figure 5: Cycles distribution between guest and host

Each of these workloads represent the trade-off between the amount of data the server side storage engine reads and writes to the file system and the number of transitions generated by the client side. In other words, when the developers adapt the middleware code they need to choose an interface that minimizes the number of guest/host transitions. Our implementation required too many transitions for small rows. We believe VAMOS can outperform para-virtual I/O even for small rows, by optimizing the code and selecting an alternative interface for the storage engine that requires less transitions from the guest to the hypervisor context.

Figure 5 compares VAMOS-MySQL with the traditional MySQL version running in an unmodified guest. We show, for each row size, the number of cycles spent in the guest and the number of cycles spent in the host. It is interesting to note that the number of cycles in the guest does not change, proving that the execution of the guest was not affected, leading us to believe that the storage engine consumes relatively few cycles. By moving the storage engine to the hypervisor, we reduce overall hypervisor overhead. This is because VAMOS reduced the number of transitions and part of the virtual hardware

emulation, which overall consumed more cycles than the storage engine itself.

## 4 Discussion

Running additional code at the host level raises new concerns with regards to the complexity of the hypervisor and potential security risks. We believe that with an appropriate design the hypervisor code can remain clean and simple, and the middleware modules could be loaded as isolated plugins. In our proof of concept, the KVM kernel module itself required no changes. The QEMU user-space process required only two minor changes: to load the middleware shared library and to delegate requests coming from the guest over the virtio channel.

As described in Section 2.2, to neutralize security vulnerabilities the hypervisor should restrict the privilege of the middleware modules. In the case of KVM, each virtual machine has a dedicated QEMU instance and middleware modules run in user-space as QEMU plugins. The first level of defense against compromised middleware components is for QEMU to use a plugin-isolation mechanism[7, 23]. The second level of defense is to use the standard Linux access control mechanisms (e.g., SELinux, AppArmor, seccomp, chroot) to limit QEMU instances from accessing different resources depending on the virtual machine being run. For example, each QEMU instance could be restricted to only access the parts of the file system that are relevant to the virtual machine it runs.

Virtualization systems often struggle to find the correct balance between guest transparency and performance (e.g., para-virtualized I/O reduces guest transparency but increases performance). While VAMOS improves I/O performance, it also ties the guest virtual machine closer with the hypervisor and makes the resulting system more complex. We believe that there is no single right trade-off, and VAMOS presents an interesting and useful design point in the spectrum of possible trade-offs.

## 5 Related Work

Para-virtualization is commonly used to reduce the virtualization overhead. Previous works [4, 8, 17] show how to para-virtualize devices drivers to improve the I/O performance, however, they still use low level interfaces to interact with the hypervisor which cause transitions and require emulation of virtual hardware.

SR-IOV devices [6, 12] improve the I/O performance by directly assigning hardware resources to the guest, bypassing part of the virtualization layer. This approach requires special hardware and does not completely mitigate the overhead caused by the transitions to the hypervisor context. In addition, using dedicated hardware, the hypervisor software lose the ability to intercept and control the data flow path.

The use of higher abstraction levels to interact with the hypervisor was also proposed in Virt-FS [9], Libra [3] and Scalable I/O [19]. Virt-FS presents a para-virtualized file system. While it shares some concepts with VAMOS, it still limits the abstraction level of guest/host interaction to inside the guest kernel, whereas VAMOS takes it up into userspace. Libra is a small operating system capable of running only an adapted JVM, taking advantage of the host OS services to improve the workloads performance. VAMOS does not require changes in the guest OS and take advantage of the host OS by running selected middleware modules directly at the hypervisor level, minimizing the adaption cost and re-using existing code. Scalable I/O presents a totally new architecture for the I/O stack, requiring major software modifications and targeting only I/O performance.

## 6 Conclusions and Future Work

Virtualization has a performance penalty caused by the overhead of transitions between the guest and the host context. In general, I/O intensive workloads are the most affected, because they create a significant number of transitions and require emulation of virtual hardware.

VAMOS is a new approach for mitigating I/O virtualization overhead by breaking free from traditional boundaries and running part of the middleware directly at the hypervisor level. By exploiting existing modular designs and abstraction layers, middleware can be adapted to run at the hypervisor level with modest cost. Since there are many different middleware, we are currently analyzing whether it is possible to create a layer of common services at the hypervisor level, shared across different middleware.

## References


[1] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, pages 2–13, 2006.

[2] N. Amit, M. Ben-Yehuda, D. Tsafrir, and A. Schuster. vIOMMU: efficient IOMMU emulation. In *USENIX Annual Technical Conference (ATC)*, 2011.

[3] G. Ammons, J. Appavoo, M. Butrico, D. Da Silva, D. Grove, K. Kawachiya, O. Krieger, B. Rosenburg, E. Van Hensbergen, and R. W. Wisniewski.

Libra: a library operating system for a jvm in a virtualized execution environment. In *ACM/USENIX Int'l Conference on Virtual Execution Environments (VEE)*, pages 44–54, 2007.

[4] 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. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 164–177, 2003.

[5] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour. The Turtles project: Design and implementation of nested virtualization. In *USENIX Symposium on Opearting Systems Design & Implementation (OSDI)*, pages 423–436, 2010.

[6] Y. Dong, Z. Yu, and G. Rose. SR-IOV networking in Xen: architecture, design and implementation. In *USENIX Workshop on I/O Virtualization (WIOV)*, pages 10–10, 2008.

[7] B. Ford and R. Cox. Vx32: lightweight user-level sandboxing on the x86. In *USENIX Annual Technical Conference (ATC)*, pages 293–306, 2008.

[8] A. Gavrilovska, S. Kumar, H. Raj, K. Schwan, V. Gupta, R. Nathuji, R. Niranjan, A. Ranadive, and P. Saraiya. High-Performance Hypervisor Architectures: Virtualization in HPC Systems. In *Workshop on System-level Virtualization for HPC (HPCVirt)*, 2007.

[9] V. Jujjuri, E. Van Hensbergen, and A. Liguori. VirtFS—A virtualization aware File System passthrough. In *Ottawa Linux Symposium (OLS)*, pages 109–120, 2010.

[10] C. W. Krueger. Software reuse. *ACM Computing Surveys*, 24:131–183, June 1992.

[11] S. Kumar, H. Raj, K. Schwan, and I. Ganev. Re-architecting VMMs for Multicore Systems: The Sidecore Approach. In *Workshop on Interaction between Opearting Systems & Computer Architecture (WIOSCA)*, 2007.

[12] J. Liu. Evaluating standard-based self-virtualizing devices: A performance study on 10 GbE NICs with SR-IOV support. In *IEEE Int'l Parallel & Distributed Processing Symposium (IPDPS)*, pages 1–12, 2010.

[13] J. Liu and B. Abali. Virtualization Polling Engine (VPE): Using Dedicated CPU Cores to Accelerate I/O Virtualization. In *ACM Int'l Conference on Supercomputing (ICS)*, pages 225–234, 2009.

[14] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM (CACM)*, 15:1053–1058, December 1972.

[15] H. Raj and K. Schwan. High performance and scalable I/O virtualization via self-virtualized devices. In *Int'l Symposium on High Performance Distributed Computer (HPDC)*, pages 179–188, 2007.

[16] K. K. Ram, J. R. Santos, Y. Turner, A. L. Cox, and S. Rixner. Achieving 10Gbps using Safe and Transparent Network Interface Virtualization. In *ACM/USENIX Int'l Conference on Virtual Execution Environments (VEE)*, March 2009.

[17] R. Russell. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review (OSR)*, 42(5):95–103, Jul 2008.

[18] J. R. Santos, Y. Turner, j. G. Janakiraman, and I. Pratt. Bridging the Gap between Software and Hardware Techniques for I/O Virtualization. In *USENIX Annual Technical Conference (ATC)*, pages 29–42, June 2008.

[19] J. Satran, L. Shalev, M. Ben-Yehuda, and Z. Machulsky. Scalable I/O - a well-architected way to do scalable, secure and virtualized I/O. In *USENIX Workshop on I/O Virtualization (WIOV)*, pages 3–3, 2008.

[20] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen. The structure and value of modularity in software design. In *European software engineering conference*, pages 99–108, 2001.

[21] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and performance in the denali isolation kernel. In *USENIX Symposium on Opearting Systems Design & Implementation (OSDI)*, pages 195–209, 2002.

[22] P. Willmann, J. Shafer, D. Carr, A. Menon, S. Rixner, A. L. Cox, and W. Zwaenepoel. Concurrent Direct Network Access for Virtual Machine Monitors. In *IEEE Int'l Symposium on High Performance Computer Architecture (HPCA)*, pages 306–317, 2007.

[23] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *IEEE Symposium on Security & Privacy*, pages 79–93, 2009.