# MIKELANGELO

## D2.13

## The first sKVM hypervisor architecture

| | | | |
|---|---|---|---|
| **Workpackage:** | 2 | Use case & Architecture Analysis | |
| **Author(s):** | Eyal Moscovici | | IBM |
| | Razya Ladelsky | | IBM |
| | Holm Rauchfuss | | Huawei |
| | Shiqhing Fan | | Huawei |
| | Gabriel Scalosub | | BGU |
| | Niv Gilboa | | BGU |
| **Reviewer** | Gabriel Scalosub | | BGU |
| **Reviewer** | Nadav Har'El | | Cloudius Systems |
| **Dissemination Level** | public | | |

| Date | Author | Comments | Version | Status |
|---|---|---|---|---|
| 2015-07-22 | Razya Ladelsky, Eyal Moscovici | Initial draft | V0.0 | Draft |
| 2015-07-23 | Shiqing Fan | virtio-rdma draft | V0.1 | Draft |
| 2015-08-19 | Niv Gilboa | SCAM | V0.2 | Draft |
| 2015-08-23 | Razya Ladelsky | Integration of comments | V0.3 | Review |
| 2015-08-31 | Gregor Berginc | Final version | V1.0 | Final |

# Executive Summary

sKVM, a superfast KVM-based hypervisor, is one of the central deliverables of MIKELANGELO [1]. Architecture enhancements focus on I/O sub-system improvements for HPC and big data use cases both in performance and security.

The newly introduced I/O core manager (IOcm) will dynamically tune the system by increasing or decreasing the number of dedicated cores performing I/O for virtual machines (VMs) based on the behaviour of the workload. With the adaptive algorithm, tuning decisions are made to always keep the most efficient utilization of the I/O cores. The use cases in MIKELANGELO project are analyzed and evaluated, in order to improve the dynamic tuning mechanisms.

Network I/O performance will furthermore be improved by a lightweight RDMA virtualization, for the broadest possible coverage of existing applications exploiting the new high performing interconnects with both socket and InfiniBand API for guest applications. Communication between virtual machines on the same host will be based on a shared memory shortcut providing additional performance improvements. This solution of lightweight RDMA virtualization does not only aim at improving the communication performance, but also ensuring that no additional modification is required in the guest applications.

Having higher performance in a virtualization environment normally means losing some degrees of security as a trade-off. In order to have both higher performance and higher level of security, a new security solution within the sKVM provides monitoring at the hypervisor level, profiling and mitigation mechanisms. The security component is addressing several possible security issues and examples of attacks and threats for the modern virtualization architecture. Advanced security solutions are then presented identifying the security issues and avoiding further threats from those virtual machines that are attempting to violate the security policies.

The deliverable sets the initial hypervisor design which is the basis for an overarching architecture to maximize I/O performance with enhanced security on all levels of MIKELANGELO stack.

# Table of Contents

# Table of Figures

# Table of Tables

# 1 Introduction of the Overall Architecture

In this deliverable we present the design of sKVM, superfast-kvm. The sKVM architecture is aimed at enabling HPC (High-Performance Computing) and big data providers to virtualize their workloads and deploy them into the cloud. To accomplish this goal we develop a KVM-based hypervisor with improvements to both I/O performance and security.

Modern hardware has virtualization extensions, which almost eliminate CPU and memory virtualization overheads. While CPU and memory virtualization approach bare metal performance, virtualizing the I/O remains the main bottleneck.

Our objective is to develop techniques and mechanisms to accelerate virtual I/O, and improve scalability for multiple virtual machines running on a multi-core host. By accelerating the virtual I/O we reduce the performance overhead incurred by virtualization, thereby making Cloud and HPC more efficient, which is one of MIKELANGELO's most important goals.

sKVM targets both clouds with commodity hardware, and clouds on top of specialized hardware, in particular InfiniBand and RDMA network devices. For clouds using commodity hardware the sKVM stack offers speed-ups over KVM by using the IOcm component.

The IOcm, the I/O core manager, allows dynamically utilizing the cores efficiently according to the varying behaviour of the workload, i.e. compute intensive vs. I/O intensive loads.

HPC and big data workloads are characterized by heavy compute intensive stages, followed by I/O intensive ones, and vice versa. Therefore, configuring the system to dynamically adapt to a configuration that is optimized for each such stage according to its load, can offer performance boost for such applications.

For both HPC and clouds with Network Inter-Connects supporting InfiniBand or Ethernet, the sKVM offers a lightweight RDMA virtualization solution, based on an in-guest virtio-rdma front-end virtual driver. It provides different network interfaces for the guest application (socket and RDMA verbs) and drives the communication over an InfiniBand network, GbE network or using shared memory within the same host. The main goal of the virtio-rdma is to improve the communication speed by RDMA and shared memory protocols, and to support guest applications using InfiniBand verbs [22] or socket interface for inter-VM communications on the same host or between different hosts. However, sKVM does not focus only on improving I/O performance through IOcm and virtio-rdma. sKVM furthermore addresses an important security aspect. sKVM targets securing cache side-channel attacks by presenting SCAM, a side channel attack monitoring/mitigation module built into the hypervisor itself.

Cache-based side-channel attacks allow co-located, malicious virtual machines to extract information from legitimate target VMs using shared cache-memory. These attacks are a deterrent to the adoption of cloud technologies, especially by high-value services. Moving HPC applications closer to cloud-like environments these security mechanisms form foundation of trust that has been established between HPC providers and their clients.

The role of Side-Channel Attack Monitoring & Mitigation module (SCAM) is to provide a varied granularity of monitoring, profiling, and mitigation capabilities, in order to identify VMs that are attempting to extract information from co-located VMs via cache side-channels.

Figure 1 depicts the sKVM architecture design of all three components.

In order to understand the building blocks of sKVM design, we first explain the aforementioned basic components of KVM that are targeted by our design for improvement.

sKVM is based on KVM, which is implemented as a Linux kernel module. It extends the kernel with hypervisor capabilities, and is driven by a QEMU [2] user process.

MIKELANGELO is targeting the paravirtual I/O virtualization model [33,24,40,41], which is the most prevalent technique today for providing virtual I/O devices to VMs (guests). Paravirtual I/O involves implementing an effective protocol of transferring I/O requests and responses between the guest and the hypervisor (which actually handles the I/O requests). Therefore, both the guest and the host (hypervisor) each have their respective components of the protocol.

KVM chose VIRTIO [24] as its paravirtual protocol and implements an in-kernel implementation for virtio (paravirtual) devices called vhost, currently supporting two paravirtual device types — network (vhost-net) and block device (vhost-block). The corresponding guest drivers are virtio-net, and virtio-blk as shown in Figure 1.

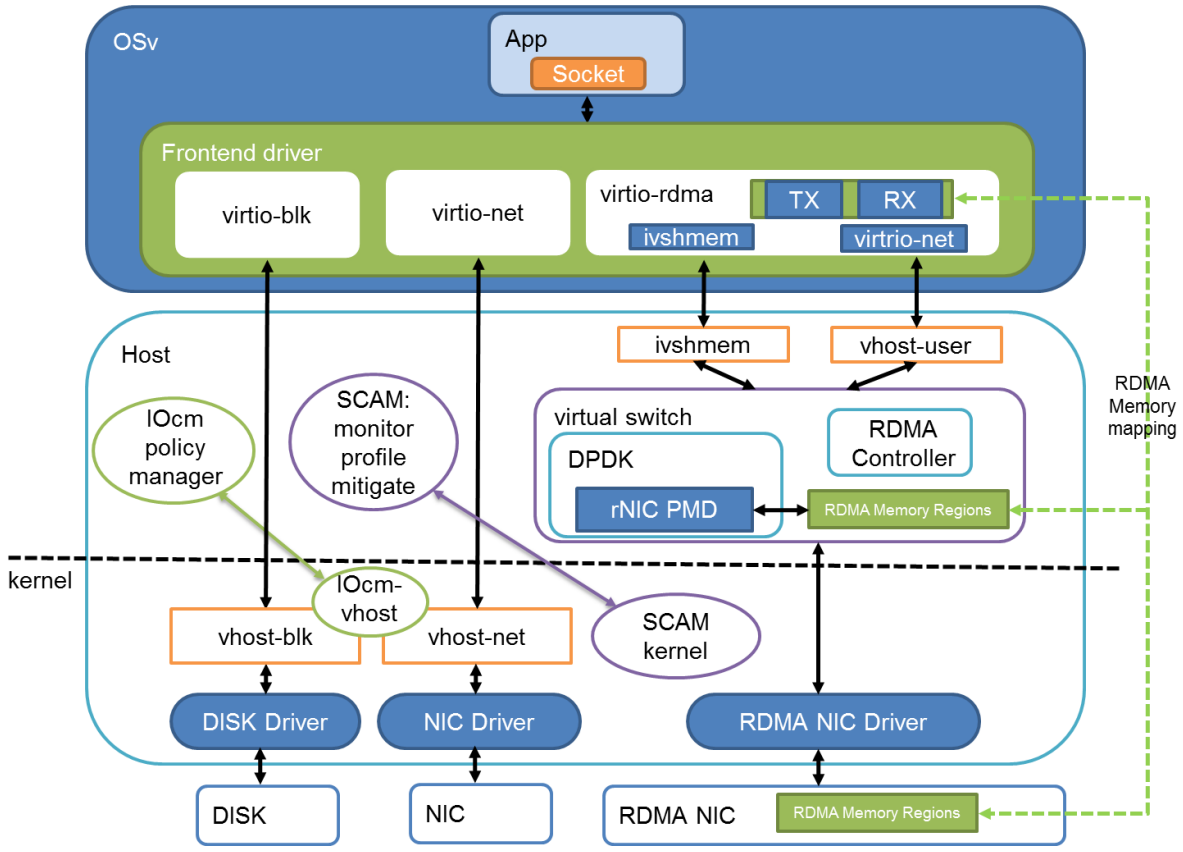


***Figure 1: The high-level architecture diagram of sKVM.***

The component diagram in Figure 1 visualises all the components that are involved in the optimisation sKVM offers over KVM hypervisor. Interactions between the guest OS, the application, user-space running on host and the kernel itself are represented with arrows.

The three components, namely the I/O core manager, the RDMA virtio driver and the security module SCAM are presented in the following subsections.

## 1.1 IOcm

IOcm is composed of two parts, a policy manager in user space, and IOcm-vhost, an in-kernel logic, based on KVM vhost as shown in Figure 1. IOcm-vhost enhances the existing (KVM) in-kernel vhost functionality and implementation, while the policy manager is a new user space utility. The two IOcm components communicate via a well-known Linux mechanism for user-space and kernel communication, called sysfs [34].

## 1.2 RDMA virtualization

The implementation of the lightweight RDMA virtualization is composed of three prototypes as discussed in section 3. The right hand side of Figure 1 shows an abstraction of these

prototypes. The virtio-rdma is a frontend driver on the guest, which takes care of the communication requests from the guest application to the underlying hardware. It drives ivshmem [21], a user-level data movement library, to provide a shared memory communication mechanism, and virtio-net for communication between the guest and the virtual switch. It may also support the InfiniBand verbs interface by paravirtualizing the InfiniBand driver on the guest, and the verb calls will be processed on the host with the help of DPDK rNIC PMD (Data Plane Development Kit RDMA Poll Mode Driver) [23]. A virtual switch based on the Open VSwitch implementation is used for managing and forwarding packets between the host and the guest. The InfiniBand driver is supported and used based on the request from the guest and passed by DPDK rNIC PMD. RDMA memory regions, including Completion Queues (CQs), Work Requests (WRs) and Queue Pairs (QPs), are mapped on the host and shared between the guest application and the RDMA device in several different ways (more details in section 3), in order to increase the communication speed, i.e. kernel bypass. WRs are communication requests with information of the peers and data. QPs are the actual data that are going to be transmitted. CQs are queues of events for notifying data arriving or completion of the communication. An RDMA Controller module allows RDMA memory regions to be processed by the guest, either by guest applications (Prototype II in 3.2.2) or by the virtio-rdma frontend driver (Prototype III in Section 3.2.3). It then forwards the InfiniBand verb requests from the guest directly to the RDMA rNIC driver. The vhost-user is a new implementation based on the kernel vhost, and it has been implemented in the latest versions of QEMU, Open VSwitch and DPDK. It works in the user space and uses kernel vhost to initialize the necessary resources that are shared between the processes in the user space. However, most of the communications are taking place in the user space. As the implementation on the host is in user space, using vhost-user for communication between the guest and the virtual switch will avoid additional switches between the kernel and user space.

## *1.3 Security*

The main sub-modules of SCAM are (1) a monitoring sub-module which collects information on cache usage by co-located VMs, (2) a profiling sub-module which aims to identify malicious activity, (3) a mitigation sub-module which employs several mechanisms in order to prevent these attacks, and (4) a kernel sub-module which services the above sub-modules. The first three sub-modules will be implemented in user space, as depicted in Figure 1. Any access to kernel-level functionality will be provided by the kernel sub-module, as depicted in Figure 1, with which the user-space modules will communicate to obtain information, and potentially take action. The execution of SCAM will be controlled by a toggle switch in sKVM, so that sKVM can decide at run-time whether SCAM operates. This will balance the trade-off between security and performance.

This document encompasses the three aforementioned components: IOcm, virtio-rdma and SCAM. The following sections describe the architecture of each component individually in full detail:

- Section 2 describes the challenges IOcm targets and the solution it proposes.
- Section 3 briefly describes the challenges RDMA virtualization faces and provides more details on our proposed lightweight RDMA virtualization solutions, including three design prototypes and how they work with different types of guest applications.
- Section 4 describes the security threats considered in this architecture, and the design proposed for tackling them.
- In section 5 we present initial benchmarks to provide an evaluation baseline for each one of the sKVM architecture components, and some preliminary measurements supporting our design goals.

# 2 IOcm Architecture

In this section we describe I/O virtualization models, the background and challenges of existing paravirtual designs, and how IOcm addresses those challenges, and targets MIKELANGELO workloads specifically. We then describe the IOcm components in more detail.

## *2.1 Background and Challenges*

### 2.1.1 I/O Virtualization Models

The three main I/O virtualization models are emulation, direct device assignment (also known as pci-passthrough), and paravirtualization.

**Emulation** is the most naive implementation of I/O virtualization, where the hypervisor implements the interface of an existing hardware device. The main advantages of emulation are that it is simple, it's hypervisor-agnostic, and the VMs are not aware that they are being virtualized.

However, the interface of the physical device being emulated was not planned with virtualization in mind. Each interaction between the VM and its emulated device requires the hypervisor's intervention, causing costly guest-host context switches (exits), thereby resulting in severely degraded performance.

**Paravirtualization**. Unlike emulation implementing an existing hardware device, paravirtualization presents a new virtual device, designed to minimize the VM-device interactions, thereby reducing the number of exits required, and thus the performance overhead incurred by emulation.

Paravirtual device drivers are hypervisor-specific, as different hypervisors can choose different interface. The guests need to install the corresponding paravirtual device drivers.

Paravirtualization has better performance than emulation, but still requires exiting from the guest to the hypervisor for every I/O request, causing performance overheads.

**Device assignment.** The host assigns physical I/O devices directly to guest virtual machines, allowing them to communicate directly with I/O devices without host involvement.

Direct device assignment provides superior performance relative to alternative I/O virtualization approaches, because it almost entirely removes the host from the guest's I/O path.

However, bypassing the hypervisor also means losing some of the of virtualization flexibility: the host software cannot offer a virtual device that has no physical counterpart (for example, a virtual disk stored as a file in the host's file system), it can't inspect or modify the guest's I/O, which can provide features such as block or packet level encryption, deep packet inspection, intrusion detection, compression, etc.

Device assignment also requires more expensive hardware (an IOMMU and SRIOV) and complicates VM live Migration [30] and memory over commitment [31].

These and other drawbacks are the reason why most enterprise data centers and most cloud computing sites refrain from using device assignment, and choose paravirtual I/O as their I/O virtualization technique.

The de-facto standard for realizing virtual I/O is through paravirtualization [33,24,40,41], exemplified by VMware VMXNET3 [32], KVM virtio [24], and Xen PV [33].

### 2.1.2 Traditional paravirtual I/O Device Model

In paravirtual I/O, each I/O request is handled by the host (hypervisor). The guest's driver, also called the front-end, writes each I/O request to a memory buffer shared by the guest and the host (request queue), and then notifies the host about the pending request.

The I/O requests are handled asynchronously, meaning that the guest doesn't block until getting the reply, and the host may handle the I/O request at a later time.

The host has a separate I/O thread to handle the requests coming from the guest. When the I/O thread completes handling the request, it writes the reply to the shared memory, and then notifies the guest that the previous request has been handled.

Each one of these notifications, from the guest to the host, and from the host to the guest involve stopping the guest VM, and exiting to the hypervisor. These switches back and forth between VM and hypervisor context introduce a lot of overhead, slowing down an I/O intensive VM.

Another I/O performance problem that traditional paravirtual implementations suffer from is lack of scalability when running multiple I/O-intensive VMs on the same host.

The cause for the lack of scalability is that the traditional approach (exemplified in current KVM) creates an I/O thread per virtual I/O device. These I/O threads are scheduled to be run on the cores by the scheduler in the host (the Linux scheduler).

Thus, when the number of I/O intensive VMs grows, more VCPU threads and I/O threads compete for the host's cores. The scheduler can choose any thread (guest and I/O thread) to run on any core. As a result, an I/O thread could be scheduled out for a long time (while other

computation or I/O threads are running), resulting in a significant reduction in throughput and increase in latency.

### 2.1.3 Elvis Paravirtual I/O Device Model

Elvis [28] improves the traditional paravirtual model by avoiding the exits associated with the I/O (request and reply) notifications, and improves the scalability by using a fine grained I/O scheduling mechanism, as explained in the following paragraphs.

**Fine grained I/O scheduling.** The traditional approach creates an I/O thread per virtual device, and lets them run using the host's thread-based scheduler. This is demonstrated in the top half of Figure 2, where every VM and I/O thread can run on any core. When there are several I/O intensive VMs, the scheduler may let a certain I/O thread run a long time, delaying I/O handling for other VMs.

Elvis uses a fine-grained approach for I/O scheduling where a single I/O thread runs on a dedicated I/O core, and handles the I/O requests of multiple VMs, as shown on the bottom half of Figure 2. Switching between handling I/O requests for multiple VMs does not involve thread context switching anymore; the I/O thread fine-grained I/O scheduler can access and examine the request queues it is serving, and can switch between them quickly and fairly.

**Avoiding I/O request notification exits.** In order to avoid the exits associated with the I/O request notification (the guest notifying the host), Elvis uses busy-polling in the I/O thread; the guests do not notify the host of new requests, but rather just queue them in the shared memory buffers. The I/O thread continuously polls the shared memory buffers of all VMs looking for new incoming requests.

**Avoiding the reply-notification exits.** Posted interrupts is a feature that has been recently added to the X86 specification which allows one core to inject a virtual interrupt into a guest currently running on a different core, without causing the guest to exit to the hypervisor first.

ELVIS avoids the reply-notification exits by emulating posted interrupts on existing x86 processors, using the Exit-Less Interrupts (ELI) technique [29].

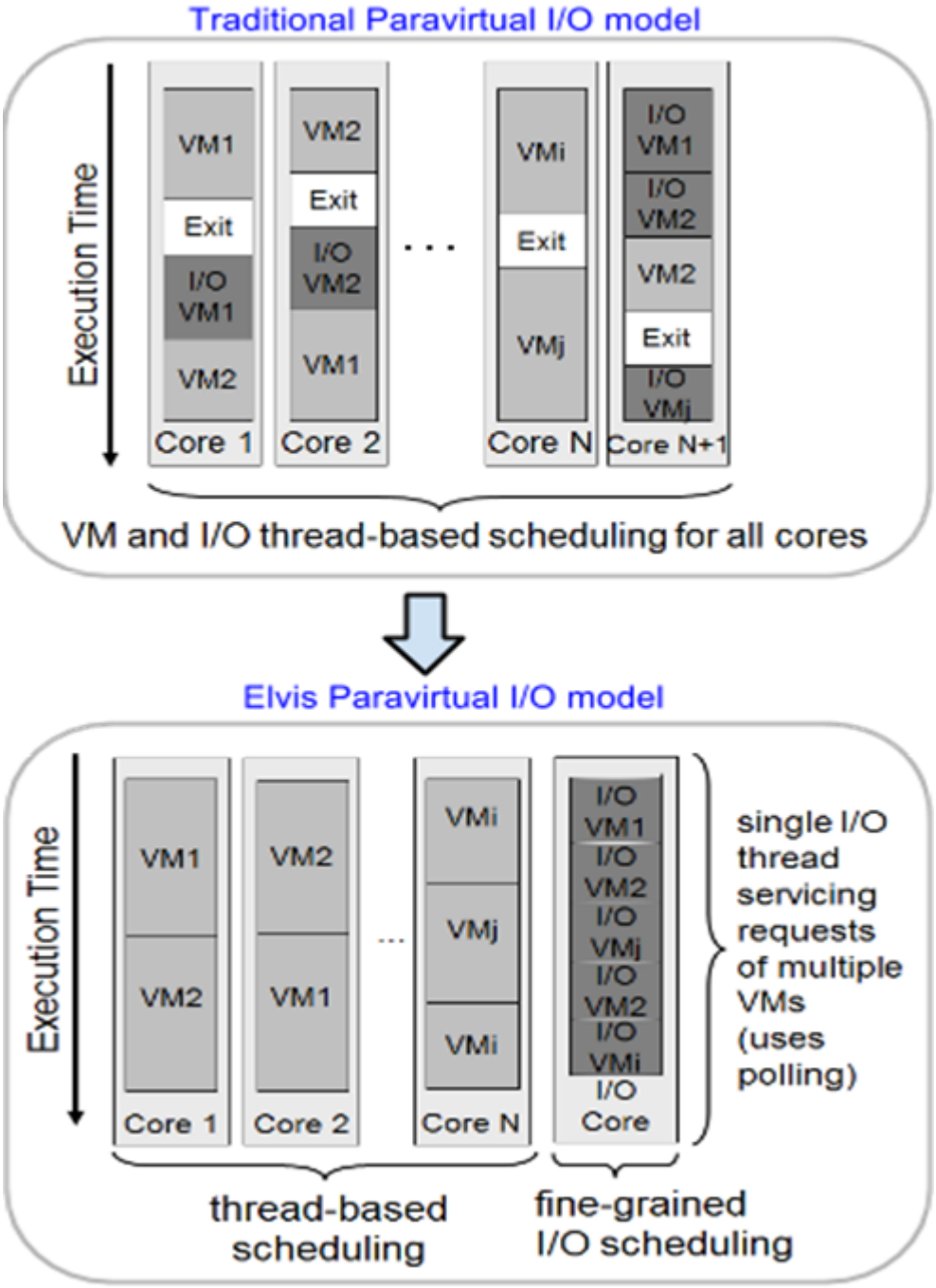


***Figure 2: Scheduling of multiple I/O intensive VMs using the traditional paravirtual approach and the Elvis approach.***

**Multiple I/O cores.** I/O intensive applications may require more than one I/O core to handle the entire I/O load. We can dedicate multiple I/O cores, each one running an I/O thread to balance the load of handling requests from I/O intensive devices.

**Elvis implementation.** ELVIS design was implemented in the KVM hypervisor. KVM offers two different implementations for paravirtual I/O devices: a user-space implementation (part of QEMU) and an in-kernel implementation, called vhost. Both implement the same protocol, VIRTIO [24], and share the same guest drivers. Elvis was based on vhost which performs better than the user-space one. It extended vhost to support fine grained I/O scheduling, together with polling, and the exitless interrupts mechanism.

**Elvis performance.** Previous evaluations of Elvis [28] show that a shared I/O thread significantly improves paravirtual I/O performance and scalability on multi-core machines. Using dedicated I/O cores handles more VMs with better throughput and latency when compared to traditional paravirtual I/O.

In Section 5.1, we describe how we evaluate and compare KVM traditional paravirtual implementation (denoted as KVM/VIRTO baseline), to the Elvis implementation (configured with a varying the number of I/O cores). The preliminary evaluation we present clearly shows that Elvis significantly improves performance as compared to the traditional paravirtual implementation. This will serve as an existing baseline for measuring improvements IOcm will offer.

### 2.1.4 Mikelangelo Use Case Challenge

The HPC and big data applications in MIKELANGELO are dynamic in nature: they consist of different computational stages with different characteristics: some phases of the program are more I/O intensive, while others are more computation intensive.

Elvis has proved to increase throughput and reduce latencies of I/O intensive VMs by dedicating cores for I/O. However, when the workload is not I/O intensive, these I/O cores deprive the VMs of the available computation power, thereby reducing the system efficiency.

Elvis uses a fixed number of I/O cores, statically dividing the cores between the I/O threads and the VMs' VCPU threads. The number of I/O cores needs to be configured manually according to the I/O load of the benchmark/application.

When the I/O load changes, the number of I/O cores may change as well, in order to maintain maximum efficiency through proper resource allocation.

Therefore MIKELANGELO's applications, having dynamic I/O loads, require that the number of I/O cores will be set dynamically, which is why Elvis is not a sufficient solution for MIKELANGELO.

## *2.2 IOcm solution and components*

We introduce IOcm, a paravirtualized I/O Core Manager that aims to improve system utilization and efficiency in comparison with other paravirtual device models.

IOcm follows the same fine grained I/O scheduler approach as Elvis, where a single I/O thread handles multiple I/O devices, running on a dedicated core.

As in Elvis, there can be more than one I/O core to balance the load of I/O traffic resulting in division of the the host's cores to 2 groups, the I/O cores, and the VMs cores.

However, IOcm does not set a fixed number of I/O cores throughout the application's life time, but rather can dynamically tune the system, by allocating and releasing I/O cores when it is beneficial to do so. Allocating an I/O core means dedicating it to an I/O thread. Releasing an I/O core means making this core available for the VMs' (vCPU) threads. In both cases we need to redistribute the devices among the set of I/O cores.

IOcm is comprised of two components: a **policy manager**, and **IOcm-vhost.**

**The policy manager** is implemented in user space. It contains the logic which uses an adaptive algorithm to make decisions regarding the allocation and release of I/O cores according to the load. These decisions are then executed by IOcm-vhost.

**IOcm-vhost** is an in-kernel module, which extends vhost module of Elvis (which itself extends KVM vhost). It implements the in-kernel mechanism that executes the allocation/release of I/O cores, as well as moving devices from one I/O thread to the other. The second role of IOcm-vhost is to provide statistics regarding the I/O activity (e.g., the I/O load produced by each device, real I/O work being done on each I/O core, etc), and expose them to the policy manager.

These IOcm user space and in-kernel components communicate via Linux sysfs mechanism, a common Linux mechanism for communication between the user space and kernel.

The interaction between these two components is shown in Figure 3.

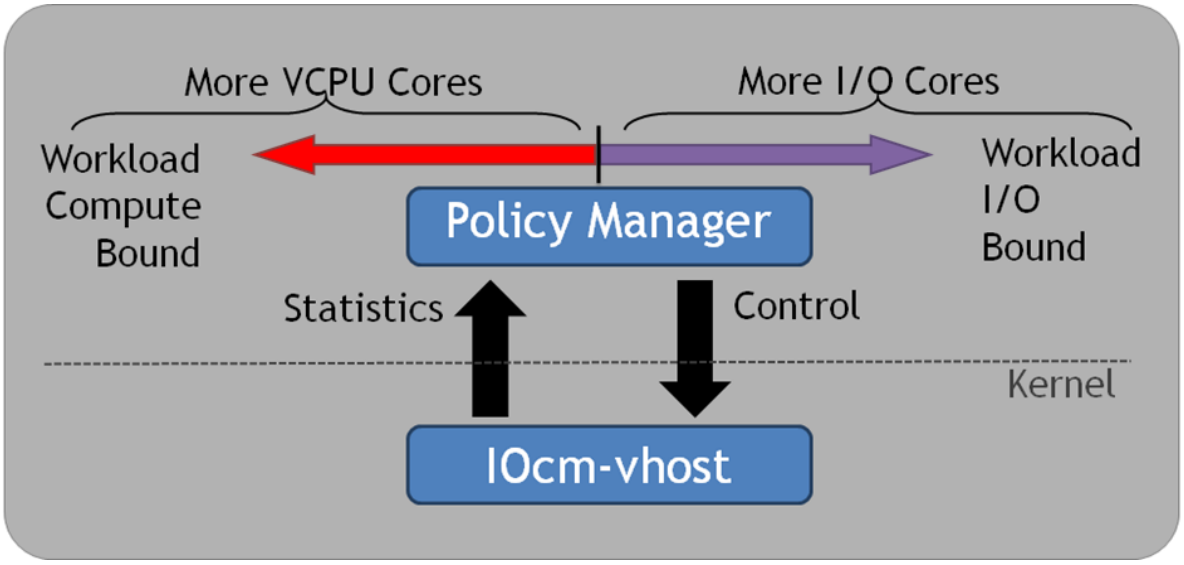


***Figure 3: The policy manager is fed by I/O statistics to make decisions. The decisions are executed by IOcm-vhost.***

**IOcm Policy requirements.** The policy manager has a system-wide view of the current I/O load on the machine via statistics and performance counters provided by the I/O threads (as part of IOcm-vhost).

The policy manger runs periodically, samples the I/O activity statistics from IOcm-vhost , and selects the appropriate number of I/O cores (0 to total_number_of_cores-1) according to its policies. If the policy manager releases all I/O cores (0 I/O cores) it falls back to the baseline model, where each virtual device has a corresponding I/O thread.

Being the controlling utility, it makes decisions on how to configure the system's cores according to the current load, adding more I/O cores when the I/O is more intensive, and releasing I/O cores, leaving them for VMs computational work when I/O becomes less intensive. When adding or removing I/O cores, it also needs to decide how to balance the I/O devices handled by these cores (more precisely, by the I/O threads running on these I/O cores).

Constructing the adaptive algorithm according to which the policy manager makes its decisions is part of the research work of this project, and will be developed during its life time.

The basic policy should consider these cases:

- if the I/O cores are loaded, and the VM vCPU cores are not, then add an I/O Core
- if the I/O cores are not loaded, and the VM vCPU cores are loaded, remove an I/O Core

The policy manager should also consider cases when ALL cores are loaded, or no cores are loaded.

Determining what "loaded" really means in real-world environment, choosing thresholds and conditions to change the configuration (and how) is part of the research questions to be studied within MIKELANGELO.

# 3 Lightweight RDMA Virtualization Architecture

## *3.1 Background and Challenges*

In order to provide better communication performance between VMs and also high flexibilities for the virtualization environment, paravirtualization of a RDMA device, namely virtio-rdma based on the virtio standard [24], is being developed in this project.

We provide several solutions to support different types of guest applications with RDMA virtualization allowing these applications to exploit benefits of virtio-rdma without any changes in the application code itself. Socket is a traditional API for TCP/IP protocol. The applications that are implemented with socket API, such as cloud applications, may still require higher bandwidth, in order to serve increasing numbers of user requests simultaneously. There are already examples of tools/libraries supporting the socket API that

enable to bypass the TCP/IP stack and directly utilize RDMA over an InfiniBand network for this purpose [43,44]. These approaches require modification of the application.

However, cloud applications, which run in a virtualized environment, may still benefit from virtualizaion on RDMA to achieve such improved performance, without switching to another API, similar to IBM Shared Memory Communication over RDMA (SMC-R) [42]. On the other hand, HPC applications normally use the InfiniBand verbs API instead of the socket API to benefit from the kernel by-pass feature of the RDMA interface.

Both of these applications should be supported with a virtualized RDMA environment and without changes to the application code itself.

## *3.2 Solutions and Prototypes*

Based on the requirements of different applications, we propose three designs of prototypes, to provide RDMA features through a paravirtualization architecture. Those three choices build both an evolutionary development and integration path for the phases of the project and trade-offs based on requirements beside performance, i.e. ability for live migration. These solutions will run with InfiniBand interconnects in either RDMA over Converged Ethernet (RoCE) [45] or InfiniBand mode.

### 3.2.1 Design Prototype I

For guest applications that are not implemented directly for an RDMA device but rather use traditional socket API, we suggest design prototype I . As shown in the following Figure 4, there is not much additional implementation or modification in the frontend driver. For design prototype I, the virtio-rdma is just an abstraction of the original ivshmem (a.k.a. Nahanni) and virtio-net [20].

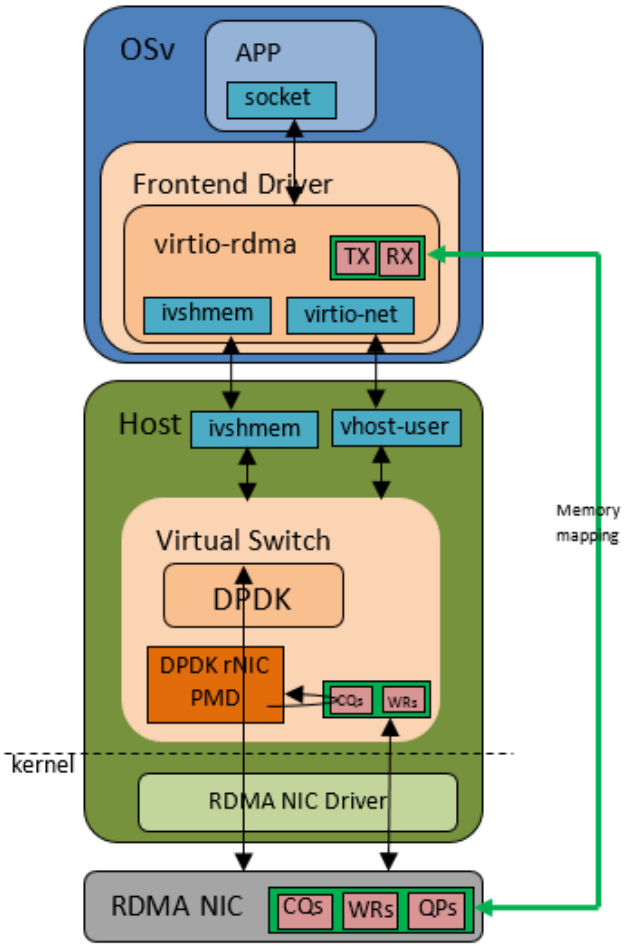


***Figure 4: Lightweight RDMA Virtulization - Design Prototype I.***

In order to manipulate the RDMA device for the upper layer protocol (mainly TCP/IP) used by the socket, the implementation has to translate the requests into RDMA verbs, while at the same time Work Requests (WRs) and Completion Queues (CQs) [25] have to be processed in the backend driver inside the Virtual Switch. The communication buffers, i.e. RX and TX rings, are necessarily pinned to memory regions to allow direct access from both the guest application and the RDMA device. This requires additional support or modification of the RDMA device driver, which is done in prototype I by including DPDK with its Poll Mode Driver implementation [23]. DPDK RDMA Poll Mode Driver (DPDK rNIC PMD) has been implemented for this purpose including the TX/RX ring allocation/deletion with pinned memory, management of communication buffers, callbacks for TX/RX queues using WRs, and most importantly events polling mechanism. It works as the backend driver in the virtual switch.

### 3.2.2 Design Prototype II

Design prototype II, as shown in Figure 5, is aimed at supporting guest applications that directly use RDMA verbs, which is normally processed by the RDMA device driver. This is supposed to be the most efficient and fastest solution of the three prototypes, because the pinned memory region is directly shared between the guest application and RDMA device. This allows the application to actively poll the CQs for the best performance. However, in order to allow the guest application to directly access the memory regions, for example, to prepare the QPs and WRs, the guest application has to know the underlying RDMA device information. The host must expose part of the hardware information to the guest, which limits the migration of the virtualization system.

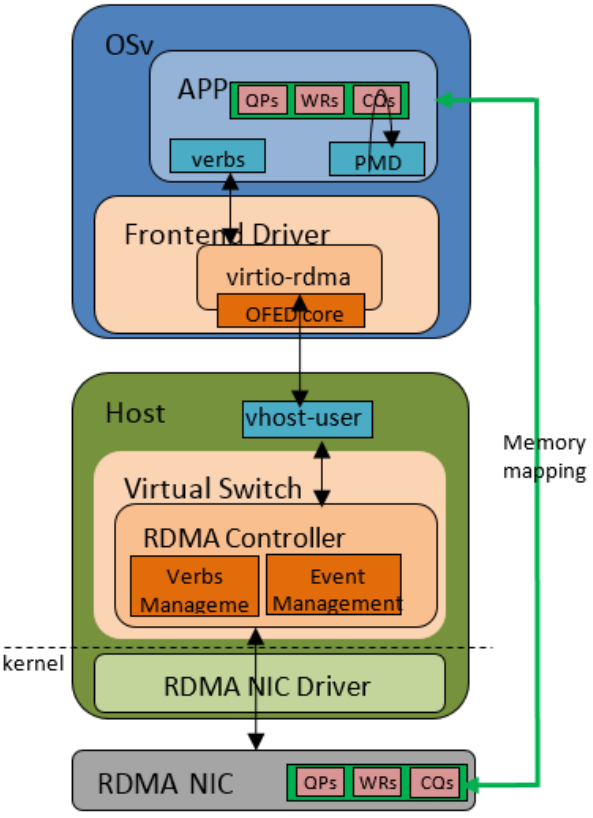


***Figure 5: Lightweight RDMA Virtulization - Design Prototype II.***

The control verbs are trapped by the virtio-rdma module and directly passed from the frontend driver to the backend virtual switch on the host. The virtual switch processes the verb commands and performs the actual RDMA operations on the device, and then passes the return data back to the frontend driver.

Preparing QPs and WRs, as well as polling the CQs are directly done by the guest application based on the OFED (OpenFabrics Enterprise Distribution) core driver. This mitigates the overhead of switching between host and guest.

Although polling CQs in the guest application has the best performance, the guest application may still use blocking mode for the completion events to save CPU cycles. In such a case, completion events have to be taken care of in the virtual switch, and sent through the user space vhost back to the frontend driver. The frontend driver will then notify the guest application.

The RDMA Controller is a module for managing RDMA verb calls and events between the guest and the kernel driver. It contains two basic components, the Verbs and Event Management modules. The Verbs Management module processes the forwarded verbs from the guest, modifies them when necessary and calls the actual RDMA verbs API provided by the rNIC (RDMA NIC) driver. The Event Management module manages the completion events from the RDMA device. When the guest application is using the polling mode of the RDMA events, this module will do nothing, because the event path is shared directly between the guest application and the RDMA device. On the other hand, if the guest application is using the blocking mode, the event management module has to either poll the completion queue or blocking wait for the callbacks from the rNIC kernel driver, and then pass the events back to the guest through the vhost path. Additional features are also possible in the RDMA controller, such as partition management, path migration, quality of service, IP security, which are basically provided by the RDMA driver. These are nice-to-have features, which may be extended or implemented for further development.

### 3.2.3 Design Prototype III

Design prototype III is a middleground solution of design prototypes I and II, which combines their advantages in order to support common APIs and also minimize the overhead (Figure 6).

The socket calls from the guest application are firstly converted to RDMA verbs by virtio-rdma and passed to the backend driver, i.e. the RDMA controller. This is similar to the functionalities of the DPDK rNIC PMD in design prototype I, but it is now moved to the guest, and more specifically, to the frontend driver. This will allow the frontend driver to actively poll the completion event and avoid sending events between the guest and the host, which is expected to show a significant improvement in performance.

Consequently, TX/RX ring buffers are directly translated to QPs in the frontend driver, and WRs and CQs are also shared between the frontend driver and the RDMA device.

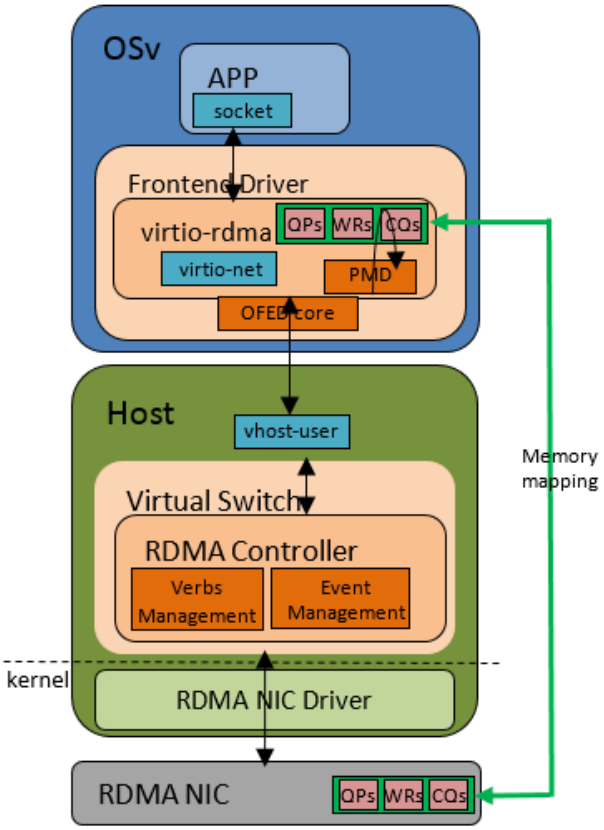


***Figure 6: Lightweight RDMA Virtulization - Design Prototype III.***

On the other hand, the event process mechanism may also be forced to use the blocking mode, which is the same as design prototype II. The RDMA controller will take over the event listening task, and send event notifications through the user vhost to the frontend driver. This mechanism is still not as performant as polling in the guest.

### 3.2.4 Comparison of the three Design Prototypes

Design prototype I has the easiest implementation as most of the modules in this case are offered as open source projects which can be directly used and integrated. It supports several communication APIs for the guest applications. However, the communication buffers are mapped between the frontend driver and the RDMA device, and WRs and CQs are processed in the backend driver. The network API calls are translated into RDMA verbs for the device by the DPDK rNIC PMD. Processing these jobs in the backend driver will introduce a lot of overhead, as every single operation requires switching between the guest and the host many times, e.g. forwarding control commands and performing buffer operations. Additionally, processing completion events on the host is also considered as a slow path, due to the fact that events have to be passed through an event queue or ring buffer via the vhost. This removes completely the benefits of polling on the completion events.

Design prototype II theoretically has the best performance with polling CQs, because communication buffers and completion events are managed and processed directly by the guest application. The involvement of the host is only for passing verbs to the RDMA device.

The main difference between prototype I and III is the position of the Poll Mode Driver. Both of them are aiming at supporting applications with socket API. But we may still have a choice between them based on the application features and requirements, in order to have the best performance. For example, for prototype I, only one PMD is running for all the virtual machines on the same host, which means the consumption of hardware resources is minimum, e.g. one core for PMD. As a result, it will have better performance when the guest application doesn't have heavy communication requests, where only one PMD is sufficient. Otherwise, prototype III should be used, because the frontend driver on each guest OS runs its own PMD, which will occupy more cores but improve the performance a lot for very heavy communication workload.

Taking design I as the starting point for the development will also benefit the development for design II and III. The DPDK rNIC PMD can be better understood and similar functionalities can be easily implemented in virtio-rdma for design prototype III. Furthermore, the implantation of the verbs pass-through can be directly reused for design prototype II.

### 3.2.5 Inter-VM communication on the same Host

Prototype I, II and III are designs of RDMA virtualization, which provide support for inter-VM communication over an RDMA network. However, for inter-VM communication on the same host, they don't present the most efficient path. However, a shared memory solution for this case is also provided using the ivshmem module. The virtual switch module will have the knowledge of the shared memory among all the VMs on the same host, and communication that takes place inside the host will simply be redirected to the shared memory. This shared memory solution is included in all the three prototypes.

# 4 SCAM Architecture

## 4.1 Background and challenges

### 4.1.1 Cache-based side channel attacks

Cache-based side-channel attacks are a specific type of attack targeting software processes executed on multiprocessing and multi-tenant systems. In such systems low-level software, either the operating system or a hypervisor, is responsible for the logical separation between different processes or virtual machines. However, it is difficult to enforce such separation in some hardware resources, specifically in cache memory.

A basic building block of such attacks is the prime+probe technique [10]. Prime+probe begins when the attacker primes the cache by writing instructions or data to the memory space that the attacker controls. Processor hardware copies this information to the cache. The attacker process then waits for the target to execute. Since the cache is much smaller than the main memory of a process, when the target process runs, its own data and instructions replace some of the information that the attacker stored. The attacker completes the process by probing, that is reading (or writing) the same data and instructions that were used in the priming stage, while measuring the time required to perform each access. If access time is relatively long then the attacker's data is no longer in the cache, which implies that the target has overwritten the data in that location and the converse if the access time is short.

The applicability of the prime and probe attack was shown in a series of works in which the attacker obtains a cryptographic key from a target process. In [10] and [13] an attacking process exploits standard implementations of the AES encryption algorithm that use lookup tables for part of the encryption or decryption process. In AES, the locations in the lookup table that the algorithm accesses depend in a straightforward way on the bits of the secret key and furthermore it is possible to infer the secret key given these locations. Since the prime and probe technique enables extraction of the accessed locations in a lookup table, standard implementations of AES are vulnerable to this attack.

Initial cache-based side-channel attacks were implemented in non-virtualized environments. In what was a surprise at the time, Ristenpart et al. [11] showed that these attacks can be reliably performed in a live cloud environment (Amazon EC2). However, the attacks in that initial work and in [15] were not sufficiently refined to extract a high-value secret such as a cryptographic key.

Recent attacks have developed a variety of tools to overcome this challenge. Authors of [18] learn a cryptographic key from the L1 cache by using inter-process interrupts to reliably synchronize the attacker's process with target execution and machine-learning methods to extract the key from the noisy cache. Yarom and Falkner [16] extract a key from the last-level cache assuming de-duplication of shared libraries. Liu et al. [8] assume only that the attacker's memory is arranged in large pages, locate the target's critical code area in the last-

level cache by looking for specific patterns of cache activity and extract the key once the required cache sets have been determined.

### 4.1.2 Mitigation of side-channel attacks

Research on mitigating side-channel attacks has been traditionally triggered by the emergence of new attack techniques, as described in the previous section. Such attacks can be performed either remotely (traversing a network) where the attacker need not necessarily be located on the same host as the target, or locally, where both the attacker and the target share the same physical host.

The overall approach in mitigation techniques of remote timing-based attacks is ensuring that observable events are independent of data (e.g., private keys). This is usually achieved using padding instructions inserted into the code in order to ensure all execution paths are of roughly the same length, thus eliminating the variance required for performing such attacks. Padding usually results in significant degradation in performance, as the target execution path length is determined by the longest such path. This latter trait has recently been improved to require padding only for secure execution paths [3]. These approaches are sometimes complemented by application-specific modifications of the code in order to obtain such independence. However, these latter approaches are extremely hard to design and implement as running-time parameters (such as cache usage) are not available in design time, which leads to insufficient information availability in order to successfully implement such approaches.

For mitigating side-channel attacks performed by co-located processes that have access to shared hardware resources, the main approaches involve limitation of access to resources / information. The following provides an illustration of the various approaches employed for performing such mitigations.

(1) Hardware-based approaches: custom hardware and hardened cache design (which are not readily available),
(2) Programming-languages-based approaches: developing programming language primitives that enable the programmer to thwart timing variations at high-level code implementations [17],
(3) Cache-based approaches: reserving allocation of cache lines to secure functions [7], flushing core-bound caches (L1-L2) upon context switch following a secure operation [19], and page coloring which allocates L3 cache segments to a specific process by restricting access of other processes to those segments [12,3].
(4) Prefetching-based approaches: introducing new cache prefetchers that obfuscate cache access by the target process [6],
(5) Scheduling-based approaches: ensuring no frequent context switches occur (which significantly reduces the rate of attack) by altering the host OS process scheduler [14],
(6) Application-specific approaches: modification of code to obfuscate secret-dependency of cache access [13],

(7) Producing various semantically-equivalent code fragments which exhibit diverse run-traces (e.g., timing, cache access, etc.) and randomly switching between them at runtime [4], and
(8) Restricted access to fine-grained timers and counters [9].

Furthermore, recent tools have begun to explore the possibility of analyzing the vulnerability of code fragments to cache side-channel attacks, by analyzing potential cache-states under various cache eviction policies [5].

## 4.2 Solutions for side-channel security threats

The following diagram provides a high-level view of our proposed architecture for SCAM -- the sKVM Side-Channel Attack Monitoring/Mitigation module. The execution of this module will be controlled by a switch in sKVM, so that sKVM can decide at runtime whether SCAM operates, thus improving security and reducing performance, or not. The role of SCAM is to provide a varied granularity of monitoring, profiling, and mitigation capabilities, in order to identify VMs that are attempting to exert information from co-located VMs via cache side-channels.

SCAM is designed as a module within sKVM, where most of its submodules reside in user-space, along with a lower-level module residing in kernel space, and its role is to facilitate the manipulation and access to kernel-level features and subsystems, most predominantly the physical memory of the host. The main modules of SCAM are the following:

(1) Monitoring module,

(2) Profiling module,

(3) Mitigation module, and

(4) Kernel module.

A detailed description of each of these modules, and the feature space that is planned to be explored within these modules, is provided in the sequel, and is summarized graphically in Figure 7.

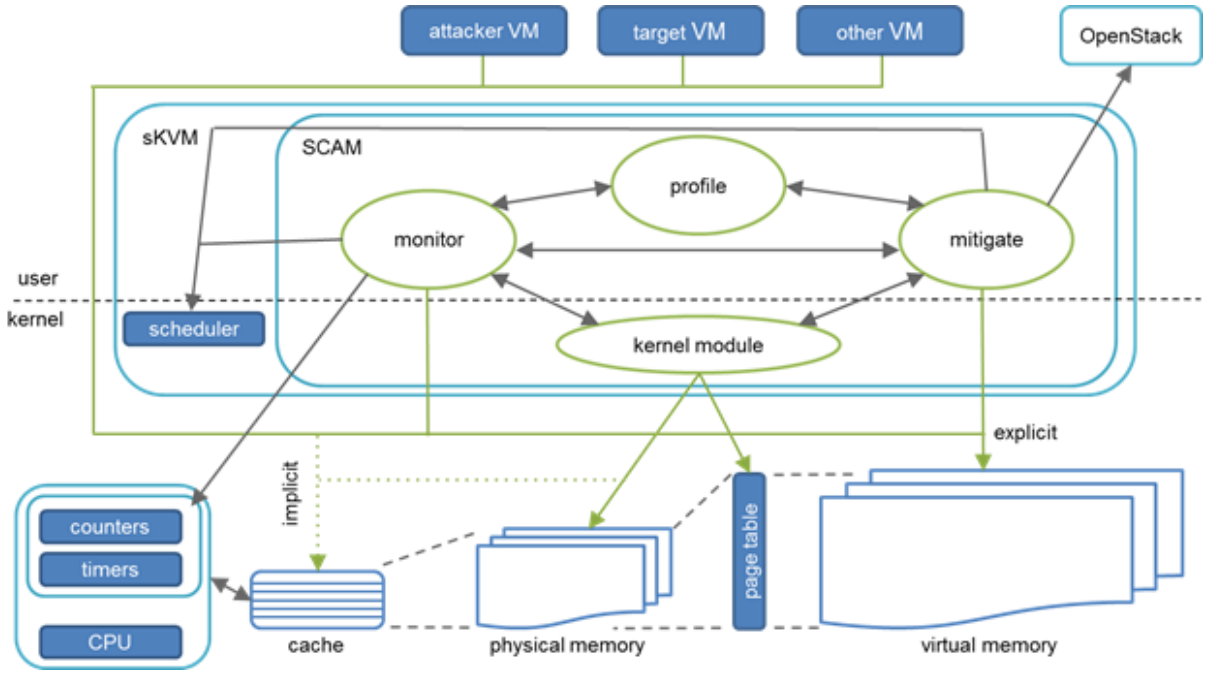


*Figure 7: SCAM Architecture*

### 4.2.1 Monitoring

The goal of the monitoring module is to collect data on the cache accesses of the virtual machines (VMs) running on the host. Since SCAM has no prior information as to the identity of a potential attacking VM, the role of this module is to collect information on the cache activity of the VMs running on the host, in an attempt to extract traces of VM cache activities that can later be profiled. The information gathered by the monitoring module is passed on to the profiling module, in order to estimate whether the activity of the monitored VM is deemed benign or hostile (see below).

The module is planned to monitor the cache access of the monitored VM. Since such access cannot be observed directly, we plan to explore several methods for performing such indirect monitoring tasks, including:

- **Memory access tracing**, where the monitoring module traces the sequence of memory access of the VM (this is enabled by interacting with the kernel module of SCAM),
- **Prime-Probe**, where the monitoring module essentially mimics an attacker's activity and obtains information about the cache access of the monitored VM,
- **Emulation**, where the monitoring module executes the code on behalf of the VM, and thus can trace its memory access.

We expect the first two options to have the least effect on performance, but these options might not suffice to obtain the monitoring granularity required by the system, in which case emulation will be considered.

The monitoring module interacts with the kernel module which is assumed to have unrestricted access to the physical memory. The monitoring module further receives privileged rights in terms of scheduling (by interacting with the sKVM scheduler), and may enforce its scheduling on any of the given cores (in order to monitor a VM that has been running on that core). Furthermore, the monitoring module requires access to counters and timers available at the kernel level in order to obtain timing and count information about the execution performed on each of the cores, along with cache access, which can later be used to infer the activity profile of the monitored VM.

### 4.2.2 Profiling

The role of the profiling module is to analyze the pattern of cache accesses of each VM and assign a score that represents the risk that a VM is conducting a cache-based side-channel attack. The input of the profiling module is the data that the monitoring module collects on each VM. The profiling module may trigger the operation of the mitigation module.

The basis for profiling VMs is a common characteristic of all currently known cache-based side-channel attacks, namely priming and probing specific cache-sets persistently. The profiling module characterizes the risk posed by a VM by the degree of similarity between the cache accesses of the VM and that of a generic attack. Given this basic assumption on profiling, the tasks that the profiling module performs are:

- Use monitoring information to identify such persistent probing of cache sets. The module may use low-overhead mechanisms such as comparing the number of probes of a cache-set with a given threshold. It may augment this approach by machine-learning methods.
- The module generates a risk score for each VM. If the risk score crosses a given threshold then the profiling module initiates the mitigation module.

### 4.2.3 Mitigation

The objective of the mitigation module is to reduce the effectiveness of cache-based side-channel attacks and prevent them completely where possible. The module takes action based on input from three possible sources. The profiling module may initialize mitigation action against a VM based on the risk score that is assigned to that VM. In addition, user applications may request protection for specific pages in memory even without any indication that there are malicious VMs running on the same hardware platform. This second option is a form of cross-layer interaction that significantly reduces the overhead incurred compared to mitigating side-channel attacks aimed at data extraction from arbitrary memory locations. Finally, the mitigation module may be configured to perform some mitigation operations on the whole system regardless of the presence of malicious VMs. The mitigation module has a number of possible tools at its disposal. These include the following:

- **Page coloring** - a form of software-configured unique assignment of part of the last-level cache to a core or a process. In this case the mitigation module requests the

kernel to assign all the pages in physical memory that map to a specific area in the cache to a specific VM.

- **Cache flushing** - the mitigation module repeatedly clears all or a number of cache sets before allowing a potentially harmful VM to run. This technique is especially effective when the attacker and the target share the same core.
- **Tweaking OS scheduler and IPI (Inter-Process Interrupt)** - the mitigation module influences the priority of a potential attacker and its inter-process interrupts to reduce the effectiveness of an attack.
- **Changing memory page size** - changing the page size of the system, e.g. from large pages to regular pages. Large pages simplify attacks on the last-level cache, due to improved visibility of the mapping of a virtual address to a cache set, whereas changing the allowed memory page size has the effect of diminishing the effectiveness of such attacks.
- **Core migration** - moving the VM of a potential attacker to a different core, e.g. to block attacks on a target executing on the same core as the attacker.
- **Eviction** - requesting the OpenStack component to migrate an attacker to a different host.
- **Changing virtual-physical memory mapping** - the mitigation module modifies the page table, changing the mapping of physical pages to virtual pages. Together with using regular pages instead of large pages, this measure has the effect of reducing an attacker's knowledge on the mapping between virtual memory and the last-level cache. This method can be used on a potential attacker, a potential target, or without even identifying attackers.

The mitigation module requires access to the kernel module and to the following services: configuration of the page table, manipulation of VM scheduling, VM memory assignment and VM core assignment. It must also have a communication link that facilitates interaction with OpenStack, presumably through libvirt. Assuming a cross-layer in sKVM, it may present a method for a VM to define a critical memory page that should be secured against attacks.

Note that both the mitigation module and the monitoring module should have a mapping of physical memory to the cache.

### 4.2.4 Kernel module

The kernel module of SCAM provides the kernel services that the other modules require. These include access to timers and counters, read and write permissions to the page table, manipulation of VM scheduling, VM memory assignment and VM core assignment.

# 5 Evaluation Baseline

In this section we provide an evaluation baseline for each one of the sKVM architecture components. We define the benchmarks, the testbed and the experimental methodology.

## *5.1 IOcm Baseline Evaluation*

Our ultimate goal is to evaluate IOcm against the other paravirtual I/O models we have mentioned—the traditional paravirtual I/O model and Elvis. Therefore, we define the evaluation for these baseline configurations.

The traditional paravirtual I/O model is the native KVM paravirtual I/O configuration, which we denote KVM/VIRTIO.

We evaluate Elvis using an Elvis enabled KVM. We want to evaluate all possible configurations for the number of I/O cores, starting from 1 I/O core to the maximum (total number of cores -1).

### 5.1.1 Used Benchmarks

The benchmarks we use are standard well known networking benchmarks. We use different configurations to demonstrate different network transportation (I/O) loads

- Netperf TCP stream, a microbenchmark stressing the network, measuring the maximal throughput sent (over one TCP connection) to demonstrate the behaviour of an I/O intensive workload. Different configurations of the benchmark (different message sizes) demonstrating different I/O loads will show the necessity for several I/O cores.
- Apache [35, 36], an HTTP web server driven by ApacheBench [37];
- Memcached [38], an in-memory key-value storage server used by many high traffic websites for caching the results of database queries, increasing the website's improved thye website's performance and scalability. We use the Memslap benchmark to drive and load the server [39]

Although our primary focus for the first iteration of sKVM will be in networking benchmarks, we will also investigate gains provided by IOcm with disk (virtio-blk) benchmarks.

Further to these benchmarks, we intend to analyse behavior of IOcm in the context of business use cases as much as possible to evaluate effects of dynamic I/O core allocation on real-world applications.

### 5.1.2 Testbed

Our testbed system consists of two servers, one hosting the VMs and the I/O cores, and the other serving as the load generator.

Each server is an IBM System x3550 M4 with: two 8-core Intel 2.2GHz Xeon E5-2660 CPUs; 56GB memory; and 2 Intel x520 dual port 10Gbps NICs

The servers are connected directly via 2 Intel x520 dual port 10Gbps NICs, allowing 40Gbps transportation between them.

Both machines currentl run Linux 3.9. The host hypervisor is KVM/QEMU, hosting VMs that also run Linux 3.9, configured with one VCPU and 1GB of memory each.

Only one 8-core CPU is used on the host. (1 socket)

### 5.1.3 Experimental Methodology

- we run 8-16 VMs (each having one vCPU and one paravirtual NIC device) on a single 8 core socket, in order to stress the system and better show the tradeoff between using the cores for the VM computations rather than I/O. Dynamically choosing whether to utilize cores for I/O threads or VMs, is best demonstrated by focusing on over-commit cases, where dedicating an I/O core necessarily means reducing the number of cores available for VMs. Running at least 8 VMs on 8 cores examines whether it is worth over-committing 8 VMs on 7 cores for the benefit of an I/O core. We wanted to examine the impact of growing the over commit even more, reaching up to 16 VMs.
- We evaluate IOcm against the other paravirtual I/O models we have mentioned —the KVM/virtio baseline, and Elvis.
    - We measure Elvis with different configurations, having a varying number of dedicated I/O cores: Elvis with 1 I/O core, Elvis with 2 I/O cores,...,Elvis with 7 I/O cores.
        - For each such configuration where we have *n* I/O dedicated cores, we set the affinity of the VMs vCPUs to be 8-*n*, allowing the VMs vCPUs to run only on the cores that are left.
        - For all Elvis configurations: Each VM is run with a paravirtual NIC device. The hypervisor is the ELVIS-enabled KVM.
    - For the KVM/virtio baseline configuration, as explained above, the KVM/virtio model created an I/O thread per VM paravirtual device, but no cores are dedicated for I/O (0 I/O cores)
        - Each of the 8-16 VMs vCPUs can run in any of the 8 core
        - All VMs vCPUs and I/O threads are allowed to run on any core (no affinity set at all)

### 5.1.4 Baseline Performance Evaluation

We ran preliminary performance evaluation comparing all the baseline configurations mentioned above, for all 3 benchmarks: netperf tcp stream, apachebench and memcached.

Each configuration was run 5 times. The first run was used as a warm up run, thus only the average of the last four runs was used. The performance numbers we refer to are an aggregation of the numbers produced by each VM.

While running the whole range (8-16 VMs), we noticed that the number of VMs does not impact the run results. Therefore, the results shown below are true to any number of VMs between 8 and 16. We still have to investigate why the number of VMs does not make any difference on performance.

### 5.1.4.1 netperf

Each VM ran Netperf TCP stream, opening a single TCP connection to the netserver, and repeatedly sending packets with a specified size (netperf message size parameter).

The netserver ran on the load-generator machine. We present the summary of the evaluation results for Netperf TCP stream in the Table 1.

Each row represents a different message size, for which we show the Elvis configuration providing the best throughput and its improvement over KVM/VIRTIO baseline.

***Table 1: Evaluation of Elvis using different message size in netper utility.***

| Netperf TCP message Size (bytes) | Number of I/O cores of the best Elvis configuration | % throughput improvement over KVM/VIRTIO baseline |
|---|---|---|
| 64 | 1 | 1.6X |
| 128 | 2 | 1.4X |
| 256 | 2 | 1.9X |
| 512 | 3 | 1.5X |
| 1024 | 3 | 1.4X |
| 2048 | 3 | 1.1X |
| 4096 | 4 | 1.1X |
| 8192 | 4 | 1.2X |
| 16384 | 4 | 1.1X |

Dedicating I/O cores proves to be beneficial in a wide range of message sizes, and we can see that different message sizes require different number of I/O cores.

In all cases Elvis outperforms KVM/VIRTIO baseline performance.

#### 5.1.4.2 apachebench

Apache is an HTTP server. We used ApacheBench to load the server.

We chose to use the same benchmark configuration as was used in the Elvis paper, i.e. ApacheBench ran on the load generator machine and repeatedly requested a static 4KB page from 2 concurrent threads per VM.

Elvis with 2 I/O cores provide the highest number of requests-per-second, which outperforms KVM/VIRTIO baseline by 2X.

#### 5.1.4.3 memcached

memcached is a key-value storage server. We used memslap to load the server.

memslap ran on the load generator machine, sending a random sequence of memcached

get(90%) and set (10%)requests to the server and measures the request completion rate.

We configured memslap to perform 16, 32, 64 concurrent requests per VM.

The evaluation shows that Elvis using 2 I/O cores is the best configuration for all measured cases, outperforming KVM/VIRTIO baseline by 2.2X 1.8X and 1.9X reqs/sec respective to 16, 32 and 64 concurrent requests configurations.

## *5.2 Lightweight RDMA Virtualization*

### 5.2.1 Benchmarks used

The following is the list of benchmarks packages that will be used to measure performance of RDMA-related components of MIKELANGELO. The list will be revised throughout the MIKELANGELO project.

- NetPerf [26], a benchmark that can be used to measure various aspect of networking performance, e.g. latency, bandwidth on TCP or UDP. It will help get the communication performance of inter-VM communication using shared memory and RDMA virtualization solutions.
- NetPIPE [27], a protocol independent network performance evaluator. It performs ping-ping test between two processes either over network or SMP with increasing message sizes. This benchmark has full support of MPI-2 application, thus is commonly used to evaluate performance of a HPC environment.

### 5.2.2 Testbed

The testbed system consists of two servers, each running 2 virtual machines. These two virtual machines are able to communicate through Ethernet or InfiniBand network inter-connects between the hosts or through shared memory inside the host.

The server is a Dell workstation with a 24-core Intel Xeon E5-2620 v2 processor, and 32GB RAM memory with the host OS being Ubuntu 14.04 LTS.

The host hypervisor is KVM/QEMU, and the VMs run guest OS the Ubuntu 14.04.2 server edition, each configured with four vCPU (assigned with four cores on the host) and 2GB of memory.

An additional testbed, provided by HPC and described in D2.19 The first MIKELANGELO architecture will be used to further evaluate the performance gains.

### 5.2.3 Experimental methodology

- For inter-VM communication on different hosts, we run single VM on 2 to 16 hosts. The hosts are inter-connected via Ethernet and InfiniBand network. Then we compare the network performance in following network modes:
  - TCP,
  - Ethernet over InfiniBand,
  - IP over InfiniBand,
  - RoCE and
  - InfiniBand.
- For Inter-VM communication of the same host, we run 2 to 8 VM on a single host.

## *5.3 SCAM*

### 5.3.1 Benchmarks used

As there is no standard benchmark for testing side channel security threats, we plan to implement the prime-probe attack model within our work in the project, in order to evaluate the performance of our monitoring, profiling, and mitigation solutions developed as part of the SCAM module. We plan to implement an attack where a malicious VM uses the cache side-channel in order to obtain private information from a co-located target VM, along the lines proposed in, e.g., [8].

We will execute the attack with no security modules installed, and provide a detailed account of the rate at which private information can be extracted, as well as the resources required for performing the attack. This will serve as a baseline implementation of the environment where security measures should be developed and installed.

Upon the development and implementation of the monitoring, profiling, and mitigation sub-modules of SCAM, we will provide a comparative study of their performance, both in terms of the resources required for their implementation, and the level of security they provide.

### 5.3.2 Testbed

Our testbed system consists of a single server, running 2 virtual machines; one hosting the attacker and the other hosting the attacked web service, each running on its own core.

The server is a Dell computer with a quad-core Intel 3.1GHz i5-2400 processor, and 3.7GB RAM memory with the host OS being Ubuntu 14.04 LTS.

The host hypervisor is KVM/QEMU, and the VMs run guest OS the Ubuntu 14.04.2 server edition, each configured with one vCPU and 0.5GB of memory.

### 5.3.3 Experimental methodology

- We consider a prime-probe L3-cache attack implementation in multi-core environments, and study the resources required for performing such an attack, as well as the rate in which private information of a co-located VM is extracted.
- We use separate cores for the attacker and the target VM, and explore the threats of providing increased resources (e.g., in terms of number of cores) to the attacking VM.
- We develop and implement the various sub-modules which SCAM comprises, including monitoring, profiling, and mitigation, by using the concepts described in Section 4. We will then evaluate the amount of resources required for implementing these components, and study the trade-offs between the resilience of the system to such side-channel attacks and any performance degradation caused by these measures being in place.
- We study the modularity of our proposed solutions, by enabling the toggling Off/On some (or all) of the sub-modules and security features developed.

# 6 Key Takeaways (Concluding Remarks)

In this deliverable, we presented the design of sKVM, superfast-kvm, which provides superior I/O virtualization solutions and advanced security mechanism for HPC and big data providers. Each component aims at different objectives and contributions for MIKELANGELO project, which are summarized in the following bullets:

IOcm, a paravirtualized I/O core manager:

- dynamically tunes the system and utilizes the cores efficiently based on the behaviour of the workload.
- implements a policy manager which constructs adaptive algorithm to make decisions regarding the allocation and release of I/O cores.
- provides statistics and performance counters regarding the I/O activity.
- implements IOcm-vhost, an in-kernel I/O core allocation and release module

Lightweight RDMA virtualization solutions:

- support for both HPC and big data applications, independent of socket or InfiniBand verbs API.

- achieve improved network communication performance without modification of the guest applications.
- are configurable for different (RDMA) network modes, i.e. RoCE or InfiniBand mode.
- advanced and transparent shortcut for inter-VM communication inside the same host using shared memory.
- can be easily updated or extended.

Security modules of sKVM within SCAM

- Provide a benchmark for prime+probe attacks, as a canonical example of cache side-channel attacks.
- Develop monitoring, profiling, and mitigation mechanisms within the hypervisor level of sKVM.
- Enabling toggling secure-mode modules.
- Present trade-offs between security solutions and performance.

# 7 References and Applicable Documents

[1] The MIKELANGELO project, http://www.mikelangelo-project.eu/

[2] BELLARD, F.QEMU, a fast and portable dynamic translator. In USENIX Annual Technical Conference (2005), pp.41–46.

[3] B. A. Braun, S. Jana, and D. Boneh, “Robust and Efficient Elimination of Cache and Timing Side Channels”. Manuscript (available at http://arxiv.org/abs/1506.00189)

[4] S. Crane, A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, “Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity,”, in NDSS, San Diego, CA, US, Feb 2015.

[5] G. Doychev, B. Köpf, and A. Rybalchenko, “CacheAudit: A Tool for the Static Analysis of Cache Side Channels,” ACM TISS, no. 18, vol. 1, Jun 2015.

[6] A. Fuchs and R. B. Lee, “Disruptive Prefetching: Impact on Side-Channel Attacks and Cache Designs,” in SYSTOR, Haifa, Israel, May 2015.

[7] T. Kim, M. Peinado, and G. Mainar-Ruiz, “STEALTHMEM: System-Level Protection Against Cache-Based Side Channel Attacks in the Cloud,” in USENIX Security, Bellvue, WA, US, Aug 2012, pp. 189–204.

[8] F. Liu, Y. Yarom, Y., Q. Ge, G. Heiser and R. B. Lee, “Last-Level Cache Side-Channel Attacks are Practical.” 36th IEEE Symposium on Security and Privacy (S&P 2015). 2015.

[9] R. Martin, J. Demme, and S. Sethumadhavan, “TimeWarp: Rethinking Timekeeping and Performance Monitoring Mechanisms to Mitigate Side-Channel Attacks,” in ISCA, Portland, OR, US, Jun 2012, pp. 118–129.

[10] D. A. Osvik, A. Shamir, and E. Tromer, “Cache attacks and countermeasures: the case of AES,” http://www.cs.tau.ac.il/~tromer/papers/cache.pdf, Nov 2005.

[11] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, “Hey, you, get off my cloud: Exploring information leakage in third-party compute clouds,” in CCS, Chicago, IL, US, Nov 2009, pp. 199–212

[12] K. Suzaki, K. Iijima, T. Yagi, and C. Artho, “Memory Deduplication as a Threat to the Guest OS,” in EUROSEC, Salzburg, Austria, April 2011.

[13] E. Tromer, D. A. Osvik, and A. Shamir, “Efficient cache attacks in AES, and countermeasures,” J. Cryptology, no. 2, pp. 37–71, Jan 2010.

[14] V. Varadarajan, T. Ristenpart, and M. M. Swift, “Scheduler-based Defenses against Cross-VM Side-channels,” in USENIX Security, San Diego, CA, US, Aug 2014, pp. 687–702.

[15] Z. Wu, Z. Xu, and H. Wang, “Whispers in the hyperspace: High-speed covert channel attacks in the cloud,” in USENIX Security, Bellevue,WA, US, 2012, pp. 159–173.

[16] Y. Yarom and K. Falkner, “FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack,” in USENIX Security, San Diego, CA, US, Aug 2014.

[17] D. Zhang, A. Askarov, and A. C. Myers, “Language-Based Control and Mitigation of Timing Channels,” in PLDI, Beijing, China, Jun 2012, pp. 99–110.

[18] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, “Cross-VM side channels and their use to extract private keys,” in CCS, Raleigh, NC, US, Oct 2012, pp. 305–316.

[19] Y. Zhang and M. K. Reiter, “Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud,” in CCS, Berlin, Germany, Nov 2013, pp. 827–838.

[20] R. Hat and R. Hat, “Virtual I / O Device ( VIRTIO ) Version 1 . 0,” no. January, pp. 1–96, 2015.

[21] C. Macdonell, X. Ke, A. W. Gordon, and P. Lu, “LOW-LATENCY, HIGH-BANDWIDTH USE CASES FOR NAHANNI / IVSHMEM,” *Forum Am. Bar Assoc.*, pp. 1–24, 2011.

[22] “InfiniBand Architecture Specification Volume 1,” vol. 1, 2015.

[23] I. Corporation, “Intel Data Plane Development Kit ( Intel DPDK ),” no. June, pp. 1–43, 2013.

[24] R. Russell, “virtio: Towards a De-Facto Standard For Virtual I / O Devices,” *ACM SIGOPS Oper. Syst. Rev.*, vol. 42, pp. 95–103, 2008.

[25] B. Woodruff, S. Hefty, R. Dreier, and H. Rosenstock, “Introduction to the InfiniBand Core Software,” *Linux Symp.*, 2005.

[26] Blum, R. (2003). Netperf. In *Network Performance Open Source Toolkit Using Netperf, tcptrace, NISTnet, and SSFNet* (pp. 61–78). John Wiley & Sons, Inc.

[27] Q. O. Snell, A. R. Mikler, and J. L. Gustafson, "Netpipe: A network protocol independent performance evaluator," in In Proceedings of the IASTED International Conference on Intelligent Information Management and Systems, 1996.

[28] N. Har'El, A. Gordon, A. Landau, M. Ben-Yehuda, A. Traeger and R. Ladelsky, "Efficient and Scalable Paravirtual I/O System", In USENIX Annual Technical Conference, pages 231–242, 2013.

[29] GORDON,A., NADAV,A., HAR'EL,N., BEN-YEHUDA,M., LANDAU,A., SCHUSTER,A., AND TSAFRIR,D. ELI:Bare-metal performance for I/O virtualization. In Architectural Support for Programming Languages & Operating Systems (2012).

[30] ZHAI,E., CUMMINGS,G.D., ANDDONG,Y. Live migration with pass-through device for Linux VM. In Ottawa Linux Symposium(OLS)(2008),pp.261–268

[31] YASSOUR, B.-A., BEN-YEHUDA, M., AND WASSERMAN, O. Direct device assignment for untrusted fully-virtualized virtual machines. Tech. Rep. H-0263, IBM Research, 2008.

[32] VMware, Inc. Performance evaluation of VMXNET3 virtual network device.

http://www.vmware.com/pdf/vsp_4_vmxnet3_perf.pdf,2009.

[33] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In ACM Symposium on Operating Systems Principles (SOSP), 2003.

[34] https://www.kernel.org/doc/Documentation/filesystems/sysfs.txt

[35] The Apache HTTP server project. http://httpd.apache.org. Accessed: Jan 2015.

[36] Roy T. Fielding and Gail Kaiser. The Apache HTTP server project. IEEE Internet Computing, 1(4):88–90, Jul 1997.

[37] Apachebench. http://en.wikipedia.org/wiki/ApacheBench. Accessed: Jan 2015.

[38] Brad Fitzpatrick. Distributed caching with memcached. Linux Journal, 2004(124):5, Aug 2004.

[39] Brian Aker. Memslap - load testing and benchmarking a server.

http://docs.libmemcached.org/bin/memslap.html.

[40] VMware. ESX server 2 - architecture and performance implications. Technical report, VMware, 2005

[41] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble.

Denali: Lightweight virtual machines for distributed and networked applications. Technical Report 02-02-01, University of Washington, 2002.

[42] Kassimis, Constantinos, Mike Fox, and Jerry Stevens. "IBM's Shared Memory Communications over RDMA (SMC-R) Protocol." (2015).

[43] Goldenberg, D., Dar, T., & Shainer, G. (2006). Architecture and implementation of sockets direct protocol in windows. *Proceedings - IEEE International Conference on Cluster Computing, ICCC*. http://doi.org/10.1109/CLUSTR.2006.311918

[44] rsocket - Linux man page: http://linux.die.net/man/7/rsocket

[45] RDMA over Converged Ethernet (RoCE) - An Efficient, Low-cost, Zero Copy Implementation: http://www.mellanox.com/page/products_dyn?product_family=79