Performance overhead of KVM on Linux 3.9 on ARM Cortex-A15

Lars Rasmusson
SICS Swedish Institute of Computer Science
Isafjordsgatan 22
SE-164 40 Kista, Sweden
Lars.Rasmusson@sics.se

Diarmuid Corcoran
Ericsson AB
Torshamnsgatan 21
SE-164 83, Kista, Sweden
Diarmuid.Corcoran@ericsson.com

ABSTRACT

A number of simple performance measurements on network, CPU and disk speed were done on a dual ARM Cortex-A15 machine running Linux inside a KVM virtual machine that uses virtio disk and networking. Unexpected behaviour was observed in the CPU and memory intensive benchmarks, and in the networking benchmarks. The average overhead of running inside KVM is between zero and 30 percent when the host is lightly loaded (running only the system software and the necessary qemu-system-arm virtualization code), but the relative overhead increases when both host and VM is busy. We conjecture that this is related to the scheduling inside the host Linux.

Keywords
Categories and Subject Descriptors
D.4.8 [**Operating Systems**]: Performance—*Measurements*; D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*

General Terms
Performance

Keywords
ARM Cortex-A15, HVM, Linux KVM, virtualization, performance

1. INTRODUCTION

Virtualization has been an important disruptive technology in the server and enterprise space for a number of years now. Within the embedded server and platform segment virtualization is gaining interest but still considered a poor match for the specific requirements of a tightly coupled, high performing software system sharing multiple hardware accelerator and digital signal processing units. However we observe that things are changing and that with the advent of cheap, powerful many-core chips the nature of what constitutes such a system is also expected to change dramatically.

To date much of the research and engineering effort around embedded virtualization has been focused on *micro-kernel* based *hypervisor* solutions [12, 3, 9, 5, 7, 2, 10]. Indeed with their special attention to high-bandwidth, low latency and isolation properties for robustness, micro-kernel based hypervisors do indeed seem to be a good match to the specific requirements of a classic embedded system. However as the Linux operating system takes an ever increasing share of embedded server platforms and displaces the more classic real-time OS's we feel it is necessary to understand how Linux and its specific ecosystem of software can be used to act as an embedded hypervisor supporting guest Linux instances with real-time requirements. We see this work based on the ARM's A15 architecture and KVM as a first step in understanding this area.

KVM [6] is a part of the Linux kernel that makes Linux capable of running not just regular application binaries, but also able to run an unmodified kernel binary in a special kind of process. The kernel inside the process is called a *guest*, and the main kernel is called *host*. The guest is scheduled by the host as a normal process, interleaved with other processes, but it cannot open files or do other syscalls to the host.

Reasons to run an an application in a KVM process may be to provide isolation of data or fault - the guest does not see any of the host's processes or resources unless so configured, and it cannot crash the host system. Or the reason could be a need to run legacy software that can not run directly on the host. Using KVM or other kinds of virtualization comes at a cost, and different hardware platforms have different costs and bottlenecks depending on how suited they are for some specific virtualization technology. A bottleneck may for instance be if a platform does or does not require the execution to pass through the hypervisor when moving between user space to kernel space. Likewise, modifying the virtual memory translation may or may not incurr overhead. And one system may support device initiated DMA to the virtual memory while another may require the hypervisor to completely virtualize and emulate the devices.

Earlier papers on KVM performance has been focused on the x86 platform [13, 1, 4]. The earlier results show that KVM has the potential to be an efficient hypervisor on x86, with an overhead of about 5-6 percent on average. But [11] have observed that KVM can show low performance on network bound work loads when compared to other hypervisors.

In this paper we give the results of measurements that were performed to find out the execution overhead of running inside a KVM guest, as compared to running directly in the host, on an ARM Cortex-A15 based platform. We are mainly concerned with networking overhead and to some degree also disk I/O.

KVM and QEMU

KVM takes advantage of virtualization support in the processor to be able to run code written for typical kernel tasks: direct access to the processor's virtual memory controller and page table, configuring interrupts, DMA, and talking to devices on system busses. KVM is a kernel infrastructure that enables connecting a KVM process with a system emulator. The emulator most commonly used with KVM is currently QEMU. KVM intercepts system level events that need to be handled by the emulator, such as reading from disk, drawing in a frame buffer, sending a packet on the network, etc. The emulator (QEMU, running in the host) is invoked, and emulates the functionality of the required hardware, i.e. the guest's virtual network card by using the host's network card. QEMU may also inject virtual interrupts in the guest when data has arrived. KVM and QEMU together constitute a Hypervisor, the software that enforces and implements security policy and virtual devices for a virtual machine.[1]

QEMU supports virtio[8], which is a virtualized mmio, or memory mapped IO, for fast communication between the host and the guest. It functions similarly to Xen's paravirtualization technique with shared memory and circular message buffers. Several pieces of data can be transferred in a batch, which is useful to speed up virtual I/O. A KVM specific device driver in the guest (instead of a normal device driver for a virtual device) is needed to take advantage of this way to signal to the hypervisor. In essence, the device driver writes data to a memory area shared with KVM, and signals the hypervisor when a batch of IP operations are ready to be handled by QEMU. This reduces the amount of copying and switching between host and guest.

Hardware support for virtualization

Hardware support for virtualization in the CPU provides additional ways to configure the CPU to detect and trap when a guest tries to perform system management tasks. They can be intercepted, to make sure that the guest does not mess up the system. This is usually done by having the host emulating a device in software, and have the guest's interaction operate on that *virtual* device. This technique works robustly, but introduces additional work to the CPU.

The Cortex A15 has support for speeding up some of the tasks by allowing the guest to perform some tasks without interrupting the guest. For instance, the A15 supports a nested page table. The host sets up the first page table for the guest, similar to how it is done for a regular process. The guest can then set up a second level page table that it controls completely. All address translations are translated first via the guest's table, and then via the host's table, and the final translation is cached in the translation buffer, TLB. There is also support in A15 for invoking virtual interrupts into the guests via the GIC. ARM has special feature called Security Extensions, which may be used to build a protected execution environment. This complicates things slightly, and the sum of it is that FIQs are used for Secure interrupts, and IRQs for Non-secure interrupts, and all interrupts have to be routed to Hyp mode, and into the hypervisor. Upon receiving an interrupt intended for a guest, the hypervisor will trigger a virtual interrupt in a running guest, or if the guest is suspended, store the interrupt and trigger it when the guest is resumed. [2]

The Cortex A15 is intended to be used in a multicore setting. One interesting use-case is to have a heterogeneous multicore setting together with the Cortex A7. They both share the same instruction set, the ARMv7, but they have different microarchitecture, i.e. the depth of the instruction pipeline, the CPU's issue width, etc., which lets the A15 run faster but to an increased power cost. A kernel can schedule processes on fast or low power cores depending on the particular demands. An alternative route to power efficiency is to scale down the frequency at which the CPU operates. This is supported by the Versatile Express platform, but is not used in this experiment.

2. EXPERIMENT HARDWARE AND SOFTWARE

ARM has released such a heterogeneous processor board for their Versatile Express development system. The Versatile Express consists of a motherboard, IO peripherals, microcontrollers control the system, debug interfaces, and two slots to plug prototypes of processor boards into. The processor board V2P-CA15_A7 consists of two A15 at 1GHz and three A7 CPUs at 800 MHz with a cache coherent interconnect between 1MB L2 cache for the A15 cluster and 0.5MB L2 cache for the A7 cluster, 2GB of DDR2 RAM. Because of lacking support in KVM, the three A7 cores unfortunately had to be turned off in the boot settings, and we were thus limited to use only the two A15 cores.

The Versatile Express network card is a SMCS LAN9118 10/100 Ethernet board. It models an asynchronous SRAM and interfaces directly to the Static Memory Bus on the AXI interconnect. [3]

Figure 1 shows the experimental setup. We used two machines with Gbit ethernet interfaces, for unrelated reasons connected via a traffic shaping switch that limited the throughput to 100 Mbit. The traffic shaping did not noticeably affect the results of the experiments, as the network interface card, NIC, on the Versatile Express was not able to reach up to 100 Mbit. While it is possible to attach faster network cards via the PCI slots, the experiments in this paper were conducted with the default SMC 9118 card.

One machine, stbox, was a fast 24 core x86_64 machine running Linux. The other machine was the Versatile Express machine, vebox, running Linux 3.9 for ARM. On the vebox

[1]Documentation on KVM: `http://www.linux-kvm.org/page/Documents`

[2] `http://infocenter.arm.com/help/topic/com.arm.doc.ihi0048b/CACJEIAI.html`

[3]`http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0503e/BABHDCHD.html`

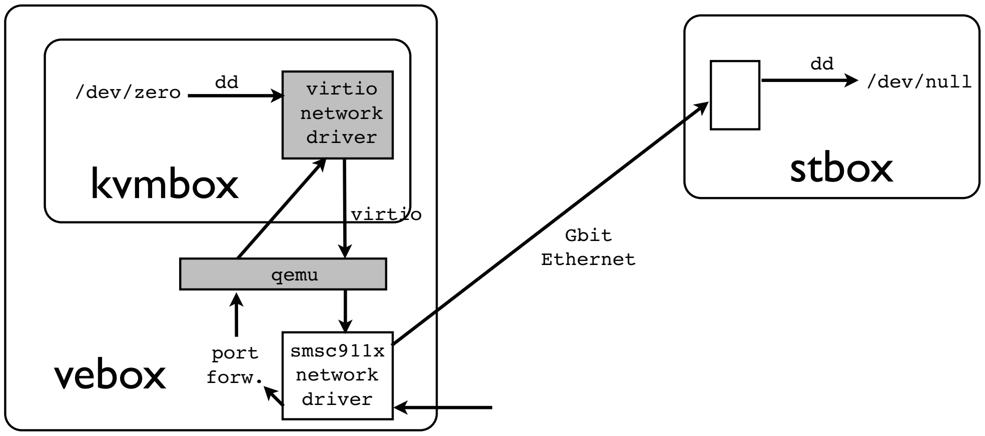

Figure 1: Experimental setup. A KVM virtual machine inside a Versatile Express machine with two Cortex A15. Both host and guest are running Linux 3.9. The KVM box uses virtio to communicate with the disk and network driver managed by QEMU. A fast machine, stbox, is attached to the network to send/receive traffic.

we also ran a KVM machine, kvmbox. Vebox was booted with 2 GB and then kvmbox was assigned 1 GB by vebox, giving them effectively 1 GB RAM each. Both vebox and kvmbox ran the same Linux 3.9 binary.

To enable KVM, the ARM processor must be booted in HYP mode. This is not supported by the onboard boot loader, so we use a special binary that is loaded together with the kernel. It sets up the CPUs in the right way before invoking the kernel.

Because the SD card interface is very slow on the Versatile Express, both the host and the guest use NFS root file systems. For that reason, the experiments were chosen to not trigger any significant NFS activity during their runs.

3. EXPERIMENTS

The experiments were conducted with quite simple Linux commands and profiling tools, such as hdparm, geekbench2, and even dd. That makes that the experiments simple to repeat. The motivation was that there is enough uncertainty in the system that more detailed measurements only provides marginally more information, and we are mainly interested in finding where the bottlenecks appear, in order to later explain them in more detail, and hopefully remedy them.

Disk read speed

There are different ways a KVM machine can be exposed to the outside world, and experiments were conducted to test their relative costs. Sometimes the effect of having virtual devices result in unintuitive results as we see in experiment 1. In this experiment we measured the read speed on three different network devices with `hdparm -Tt <dev>`. The averaged results of the disk read speed tests are shown in figure 2.

platform	**device and backing store**	**cached reads (MB/s)**	**buffered reads (MB/s)**
VE	/dev/sda	723	8.94
VE	/dev/loop0 on sda	825	115-289
VE	/dev/loop0 on nfs	827	7-289
KVM	/dev/vda	765	7-84
KVM	/dev/loop0 on vda	771	66-115
KVM	/dev/loop0 on nfs	770	7-115

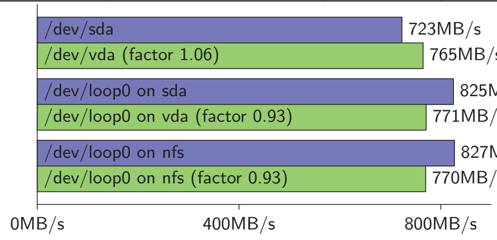

Figure 2: Disk read speeds to device, as measured by `hdparm -Tt` <*dev*>. The bar chart shows cached reads. Blue is Linux on VE (host), green is Linux in KVM (guest).

Cached reads "displays the speed of reading directly from the Linux buffer cache without disk access. This measurement is essentially an indication of the throughput of the processor, cache, and memory of the system under test." Buffered reads "displays the speed of reading through the buffer cache to the disk without any prior caching of data. This measurement is an indication of how fast the drive can sustain sequential data reads under Linux, without any filesystem overhead."

First we measured the read speed on vebox, the host, as this should be the upper limit on the performance. This showed that the SD disks was quite slow, around 9 MB/s, where a hard drive performance today usually is one or two hundreds of MB/s. To compare the effect of crossing in and out of the kernel we also created a 100MB file and mounted it as a loopback device. The result is that its content will eventually be stored in the page cache. For that reason, the "buffered reads" (which were actually cached) increased in speed for every run from 115 MB/s up to 289 MB/s when running on vebox, and the cache erased the difference between having the file was on the SD disk or on the NFS disk. However they do not reach up to the 825 MB/s as the purely cached reads reach.

Kvmbox did not have direct access to the SD disk, but was given access to a disk image on the SD disk via the virtio-mmio interface. It appears in the kvmbox as `/dev/vda`. When KVM is reading from `/dev/vda` it is invoking the host to read from the disk image backing store. Here we observe reads as low as 7 MB/s, compared to the 9 MB/s for the host vebox. However, due to caching in the host's page cache, this increases successively up to 84 MB/s.

The other read speeds measured on kvmbox corroborate the observation that for buffered reads there is large interactions between the host's page cache, resulting in heavily fluctuating read speeds, and also putting pressure on the host's page cache. There is also a considerable overhead, even with virtio-mmio, since we don't even reach half of the buffered read speed for the loopback mounted disks on vebox. For cached reads the results look more stable. The cached reads, which do not invoke the host as much, the guest achieves around 93 percent of the performance of the host.

CPU and Memory

To get a measure of the system speed for workloads that stressed the CPU and memory we ran the Geekbench2 test suite. It consists of a number of test programs that are divided into the categories "Integer performance", "Floating Point", "Memory", "Stream Processing". For each test it computes a score and computes a weighted average for each category and for the system as a whole. The results are sent to a central server to make it easy to compare against other systems. The results of the Geekbench2 tests is shown in the table below. Examples of Geekbench2 tests: Integer = blowfish, compress. Floating Point = LU-decomp., sharpen image. Memory = stdlib allocate, stdlib write. Stream = stream copy, stream scale.

In figure 3 we can see that KVM on ARM does for some reason add an overhead to Integer and Floating Point heavy computations, while Memory and Stream heavy workloads are not as affected. The result is quite unintuitive, as there should be no reason for the kvmbox to invoke the hypervisor during a purely computational workload. The reason for this behaviour is not yet known.

[4]From the hdparm man page. `http://linux.die.net/man/8/hdparm`

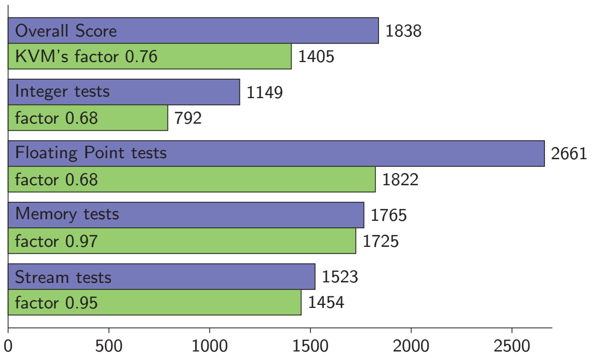

Figure 3: The scores on the CPU and memory intensive Geekbench2 test for Linux native (blue) and in KVM (green).

Network throughput

To measure the network throughput we set up one machine to listen for tcp packets on one machine and sent packets from the other machine. The listener listened on a port using netcat, while the sender sent 100 MB from `/dev/zero` to that port.

Listener: `nc -lk` <*port*> `>/dev/null`

Sender: `dd if=/dev/zero bs=1M count=100 | nc` <*listener*> <*port*>

The achieved transmission speed is printed out by dd when it has finished. The reported numbers are the average over three or more runs, and if necessary to get stable numbers, more than 100 MB was sent. The measurements vary about 0.5 MB/s up or down.

In QEMU, a multicore guest can have more cores than the number of physical cores, since each guest core is just a Linux thread. Since the host Linux kernel schedules threads on the physical cores as it pleases, there are interaction effects if both host and guest are busy at the same time. For that reason, we compared different assignments of the cores to the host and guest. Process were pinned to a specific core with `taskset -c` <*cpu nr*> <*command*>.

The table in figure 4 shows the results of the experiments. The *first* column shows how the processes on the vebox were pinned to different cores, as this was found to have a significant impact on the result.

The *second* column shows the speed when either the host, or the KVM guest, is sending to itself (localhost), and the other one is not doing anything. Here we see that the KVM guest speed is about 67 percent of the host speed, except for the case where both host and guest have two cores. In that case the guest's performance fluctuates between the same value as before, 51 MB/s and up to 83 MB/s, which is, strangely enough, *faster* than the host's native performance. We have no good explanation for this, and only point out that in this case the hypervisor should not be involved, since traffic is only sent to localhost.

Sender		Listener									
		single sender to localhost		concurrent to localhost		concurrent to vebox		single to stbox		concurrent to stbox	
host core	kvm core	host rate	kvm rate	host rate	kvm rate	host rate	kvm rate	host rate	kvm rate	host rate	kvm rate
0	0	76.5	51.6	37.7	29.5	36.0	9.2	8.2	7.7	4.1	3.7
0	1	77.4	51.3	46.1	46.0	46.1	13.4	8.2	8.0	* 3.5	6.1
1	0	76.1	51.5	44.8	46.1	46.0	13.3	9.0	7.7	6.9	2.7
1	1	76.4	51.4	37.4	30.0	36.8	9.3	8.0	* 9.0	4.1	* 4.4
0, 1	0, 1	76.4	50.9-83.3	47.8	6.5-46.0	45.6	12.4	8.9	8.7	7.4	* 1.1
0, 1	0	75.6	51.1	44.6	46.6	45.8	13.3	8.8	7.7	6.9	2.7
0, 1	1	76.1	51.6	44.5	47.1	46.0	13.3	8.8	8.0	* 3.5	6.1

Figure 4: Throughput in MB/s for sending to localhost, the KVM host, and over the network to `stbox`, while being pinned to different cores. Several unintuitive behaviours are observed. They are marked with *.

The *third* column shows a similar experiment, but this time both host and guest are running concurrently. Ideally, they would get half the throughput, minus the cost for switching between busy processes. The column shows the effect of pinning the host and guest to same core, to separate cores, and the effect of running both host and guest on both cores. We note that host and guest get the same performance (around 46 MB/s) if they are running on separate cores, and pinned to the same core the host get about half the performance as in column one, while the guest gets 75 to 80 percent of the host. Again we observe huge variability in the guest throughput when the guest has two virtual cores.

The *fourth* column measures throughput to the host. For the host that is the same as sending to localhost, and for the guest this reveals the communication overhead of the KVM virtio stack. The KVM guest throughput to the host is only a third of the throughput to localhost (i.e. divide 9.2 with 29.5, etc.), even though they are both on the same physical machine. This shows that the overhead of the KVM stack is significant, even though virtio is used.

The *fifth* and *sixth* columns show the speed of packets sent on the network to the fast remote machine denoted `stbox` in the table. In column five host and guest are running one at a time, and in column six they run concurrently. Column five and six are also presented as a bar charts in figure 5 and in figure 6 respectively. Numbers that stand out are marked with a *. The maximal throughput achieved by the Versatile Express is 9 MB/s or 72 Mbit/s, which is below the 100 Mbit/s network capacity. (It was verified 100 Mbit/s is actually reached between two faster hosts.)

When either host or guest is running on its own, the throughput of the guest matches well the throughput of the host.

The lowest throughput of 1.1 MB/s is achieved when both host and guest are running concurrently on both cores. The interaction from Linux' scheduling is quite severe in this configuration. The fastest throughput for the host is when it is pinned on core 1 while the host is pinned on core 0. Running on core 1 results in almost twice the throughput. The same holds when the host pinned on core 1 and the guest is pinned to core 0.

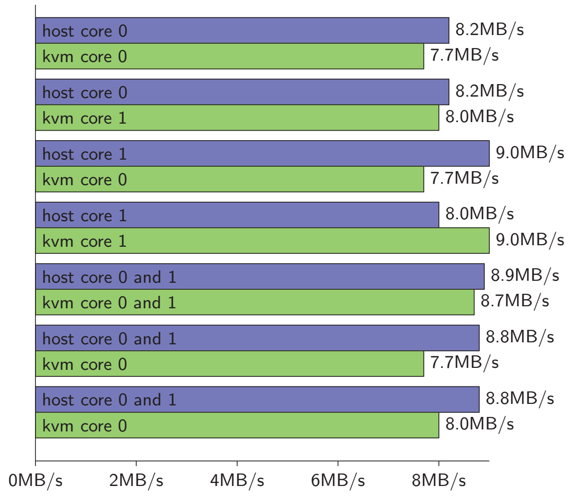

Figure 5: Network throughput for native host (blue) and KVM (green) when only one of them is sending data to stbox (fifth double-column in figure 4).

What is striking when comparing figures 5 and 6 is that the throughput of the host and the guest may fluctuate wildly depending on whether host and guest are sending concurrently. It may decrease throughput with a factor 8 (when both host and guest are scheduled on both cores). But when either one is sending alone the throughput is better than 85 percent of native performance.

4. CONCLUSIONS

We have performed a set of simple system level tests to measure the performance of disk read speed, CPU and memory performance, and network throughput for a KVM based virtual machine running Linux 3.9 on a Cortex A15 based platform. The performance was compared to a native Linux 3.9 on the same platform, and the network performance was tested while either only host or guest was transmitting alone, and with both transmitting concurrently. The guest's disk read speed was about 93 percent of the host. It was observed

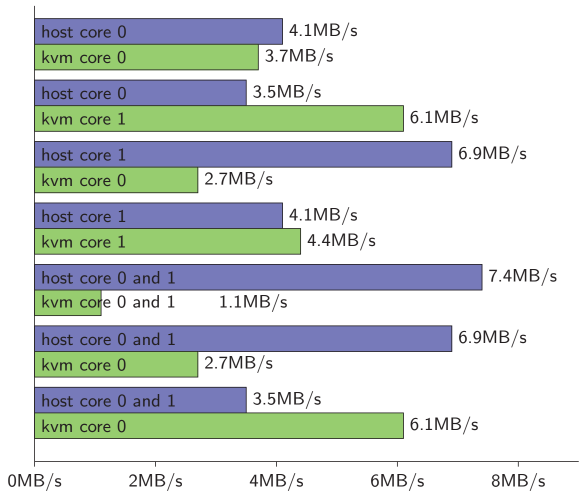

Figure 6: Network throughput for native host (blue) and KVM (green) concurrently sending data to stbox (sixth double-column in figure 4).

that the KVM machine, or rather the QEMU device manager, may cache disk data in the host's, which may affect the performance of the host's.

The CPU and memory tests showed that the memory and stream processing bound workloads run with almost native speed, around 97 percent of the host performance. However, a large, currently unexplained slowdown was observed on integer and floating point heavy workloads. Here guest performance was only 68 percent of the host performance. This begs to be investigated further, because such workloads should not invoke KVM much, and they are expected to also run with near native performance.

The network throughput was measured using the ARM supplied SMC network card. This card has less than Gbit performance and thus stresses the system in not exactly the same way that a Gbit NIC would. When either only the host or the guest transmitted, the guest achieved between 85 to 98 of the host performance. When both host and guest were busy at the same time, the aggregate performance was on par with the native host, but the split was not even between the guest and the host. The split seems to depend on how the guest is scheduled, and on which cores the programs are running. Best performance was achieved when host and guest were pinned to different cores, and worst performance for the guest was achieved when both host and guest shared all available cores. This suggests that the scheduler in Linux 3.9 interacts inefficiently with KVM and QEMU under heavy IO load.

In conclusion, KVM is a quite capable hypervisor, but it has some unexplained irregular behaviour with respect to performance. The software is under heavy development, and other hypervisors are also targeting the new HVM capable ARM CPUs, so as more hypervisors mature, we can compare them against each other, and use the results to guide the search for bottlenecks and irregular behaviour.

5. ACKNOWLEDGMENTS

This work was funded by Ericsson AB and SICS Center for Networked Systems.

6. REFERENCES

[1] A. Binu and G. Kumar. Virtualization Techniques: A Methodical Review of XEN and KVM. In A. Abraham, J. Lloret Mauri, J. F. Buford, J. Suzuki, and S. M. Thampi, editors, *Advances in Computing and Communications*, volume 190 of *Communications in Computer and Information Science*, pages 399–410. Springer Berlin Heidelberg, 2011. `http://dx.doi.org/10.1007/978-3-642-22709-7_40`.

[2] Green Hills Software. Integrity secure virtualization. `http://www.ghs.com/products/rtos/integrity_virtualization.html`, May 2010.

[3] G. Heiser and B. Leslie. The okl4 microvisor: Convergence point of microkernels and hypervisors. In *Proceedings of the first ACM asia-pacific workshop on Workshop on systems*, pages 19–24. ACM, 2010. `http://conferences.sigcomm.org/sigcomm/2010/papers/apsys/p19.pdf`.

[4] K. Huynh, A. Theurer, and S. Hajnoczi. KVM Virtualized I/O Performance – Achieving Leadership I/O Performance Using Virtio-Blk-Data-Plane Technology Preview in Red Hat Enterprise Linux 6.4. Published by RedHat and IBM. `ftp://public.dhe.ibm.com/linux/pdfs/KVM_Virtualized_IO_Performance_Paper_v2.pdf`, 2013.

[5] J.-Y. Hwang, S.-B. Suh, S.-K. Heo, C.-J. Park, J.-M. Ryu, S.-Y. Park, and C.-R. Kim. Xen on ARM: System virtualization using Xen hypervisor for ARM-based secure mobile phones. In *Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE*, pages 257–261. IEEE, 2008. `http://dx.doi.org/10.1109/ccnc08.2007.64`.

[6] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Ligouri. kvm: the Linux Virtual Machine Monitor. In *Proceedings of the Linux Symposium*, pages 225–230, June 2007. `http://www.linux-kvm.com/sites/default/files/kivity-Reprint.pdf`.

[7] Red Bend Software. vLogix Mobile for Mobile Virtualization. `http://www.redbend.com/en/products-solutions/mobile-virtualization/vlogix-mobile-for-mobile-vitrualization`, 2010.

[8] R. Russell. virtio: Towards a De-Facto Standard For Virtual I/O Devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, July 2008. `http://doi.acm.org/10.1145/1400097.1400108`.

[9] S. Stabellini. Xen on arm cortex a15. XenSummit 2012 `http://www.slideshare.net/xen_com_mgr/xensummit-na-2012-xen-on-arm-cortex-a15`, Aug 2012.

[10] U. Steinberg and B. Kauer. NOVA: A Microhypervisor-based Secure Virtualization Architecture. In *Proceedings of the 5th European conference on Computer systems*, pages 209–222. ACM, 2010. `http://os.inf.tu-dresden.de/papers_ps/steinberg_eurosys2010.pdf`.

[11] I. Tafa, E. Beqiri, H. Paci, E. Kajo, and A. Xhuvani. The Evaluation of Transfer Time, CPU Consumption and Memory Utilization in XEN-PV, XEN-HVM, OpenVZ, KVM-FV and KVM-PV Hypervisors Using FTP and HTTP Approaches. In *Proceedings of the 2011 Third International Conference on Intelligent Networking and Collaborative Systems*, INCOS '11, pages 502–507, Washington, DC, USA, 2011. IEEE Computer Society. `http://dx.doi.org/10.1109/INCoS.2011.164`.

[12] P. Varanasi and G. Heiser. Hardware-supported virtualization on arm. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, page 11. ACM, 2011. `http://www.nicta.com.au/pub?doc=4938`.

[13] B. Zhang, X. Wang, R. Lai, L. Yang, Z. Wang, Y. Luo, and X. Li. Evaluating and Optimizing I/O Virtualization in Kernel-based Virtual Machine (KVM). In C. Ding, Z. Shao, and R. Zheng, editors, *Network and Parallel Computing*, volume 6289 of *Lecture Notes in Computer Science*, chapter 20, pages 220–231. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2010. `http://dx.doi.org/10.1007/978-3-642-15672-4_20`.