Quantifying Artifacts of Virtualization: A Framework for Mirco-Benchmarks

Chris Matthews
University of Victoria
cmatthew@cs.uvic.ca

Yvonne Coady
University of Victoria
ycoady@cs.uvic.ca

Stephen Neville
University of Victoria
sneville@ece.uvic.ca

Abstract

One of the novel benefits of virtualization is the ability to emulate many hosts with a single physical machine. This approach is often used to support at-scale testing for large-scale distributed systems. In order to better understand the precise ways in which virtual machines differ from their physical counterparts, we have started to quantify some of the timing artifacts that appear to be common to two modern approaches to virtualization. Here we present several systematic experiments that highlight four timing artifacts, and begin to decipher their origins within virtual machine implementations. These micro-benchmarks serve as a means to better understand the mappings that exist between virtualized and real-world testing infrastructure. Our goal is to develop a reusable framework for micro-benchmarks that can be customized to quantify artifacts associated with specific cluster configurations and workloads. This type of quantification can then be used to better anticipate behavioral characteristics at-scale in real settings.

1 Introduction

Intuitively, virtualization allows multiple *virtual machines* to be run on one physical host. A virtual machine (VM) is defined as an efficient isolated duplicate of the real machine [13]. One of the many advantages virtualization provides is the ability to test software targetting large-scale systems by emulating clusters. Several commercial vendors offer modern virtualization platforms that use a variety of techniques to provide the illusion of multiple machines on a single physical machine. Vendors like Citrix [2], Microsoft [4], Sun [5], and VMware [6] all have offerings in the virtualization space.

Of course, there are some well-known caveats associated with this definition of virtualization. First, timing of events may be different relative to a physical platform. Second, the resources presented to any single VM may be reduced as a result of sharing. These issues need to be taken into account when mapping virtualized testing results into corresponding expectations in real-world settings. Although VMs are functionally equivalent to a physical machine, they will display some different behavioral characteristics, particularly at a large-scale, due to these differences in the amount of memory and CPU time allocated in a partitioned system.

After providing some background and related work on the practical considerations associated with virtualization, the impact of virtualization is quantified in the context of micro-benchmarks run in two environments: virtualized and real. Section 2 details the core part of framework in which a single packet is sent between hosts in order to determine latency. In Section 3 patterns of artifacts are more deeply explored based on these results. Section 4 considers the framework in terms of a platform-agnostic perspective. Section 5 further extends the core benchmark within an application-specific scenario of a webserver. Section 6 reflects on the need for this framework, and Section 7 concludes and describes future work.

1.1 Background and Related Work

On modern computer architectures like x86 [3], the task of providing an efficient isolated duplicate of the real machine is not an easy one. To provide efficient virtualization a “statistically dominant subset” of the VM’s instructions have to be executed directly by the real machine [13]; however, to keep VMs isolated, the system must be in control of what a VM does. To satisfy both the isolation and efficency properties, most Virtual Machine Monitors (VMMs) have evolved structure similar to operating systems, using the notion of privileged operations and controlled access to system resources.

In Popek and Goldberg’s paper [13], they define a *trap and emulate* pattern to control privileged operations. When a privileged operation is executed, it is trapped, and control is given to the VMM. The VMM is then able to emulate that operation appropriately. Modern virtualization systems stray a little from this definition, but employ mechanisms that accomplish the same goal. Briefly, some of those mechanisms are *dynamic binary translation*, which dynamically replaces privileged instructions with entry points into the

VMM [7]; *paravirtualization*, which the VM is explictly altered to make calls to the VMM and not perform any privileged instructions [10]; and finally, *hardware virtualization*, provided by modern hardware vendors [8, 12] which achieves the same results by adding functionality to the hardware.

A virtualization platform also has to provide mechanisms to partition the resources of the machine amongst VMs. Processor time has to scheduled, memory has to be allocated and reclaimed. I/O devices like disks and network have to be partitioned or multiplexed. The smarter the mechanisms are, the more performant the overall system becomes. However, this in turn increases the complexity of VMMs.

The reported statistics regarding performance of virtualized systems shows that the overhead of running this low-level infrastructure could be anywhere between 0%-50% [7, 9–11, 14]. Those overhead numbers depend on workload, the type of virtualization and the mechanisms the virtualization platform employs. The types of workload that costs the most are those that need the system to be in control the most — specifically, these are I/O based workloads [7].

2 Micro-Benchmarks: What are the Costs?

To assess the impact of virtualization on simple timed experiments, we first established the anticipated overheads associated with communication costs between virtual machines running on the same host, versus simple processes performing the same communication in the real system. In our benchmark we place a simple server in each virtual machine. This server is responsible for waiting for connections and, when received, reading a simple message (which in our configuration of the benchmark, is just an integer value). This message is then forwarded to another server.

To measure the latency introduced over *N* hops through VMs, we setup *N* VMs with a server in each. Each VM forwards its' message to the next VM in a ring formation, so that the message eventually returns to the server that originally sent it. The benchmark measures the time from when the first connection was initiated, to when the message has travelled through all of the servers and arrived back at the originator.

This benchmark is designed to be run in several different configurations in terms of VMs. For example, one configuration in which all the servers reside on the same VM, and another in which each server resides in its own VM. Figure 1 illustrates these two experimental configurations for 3 hops: in (a) all of the servers reside in a single VM, in (b) servers are distributed across domains.

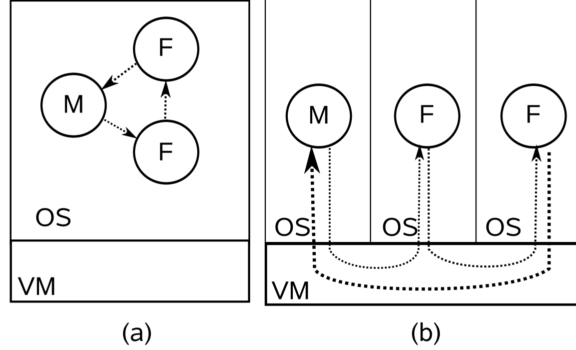


Figure 1. A master, M, and forwarders, F, in (a) a single domain configuration, and (b) cross domain configuration. Both configurations are for 3 hops.

2.1 Framework Implementation

The servers are all written in C, and a server process runs in one of two modes:

- Forwarder Mode: receives request on a port, reads the message, and then sends that message to another IP and port.

- Master Mode: sends an message once every X seconds to a specified IP:PORT, then waits for a connection in which that message comes back. The master is responsible for keeping the time difference from beginning of the send to end of the receive.

A set of programs written in Python takes the role of a supervisory system. One program reads a configuration from a file or remote location and then sets up the server processes that should be running on the VM in which that particular instance of the supervisor program is running. The supervisor makes sure that processes run without aborting from unnatural causes, and aggregates their output and data. Supervisors run on each VM in the experiment, and coordinate with a central server.

2.2 Experimental Configuration

In our case this communication benchmark was run on HP ProLiant DL320 server with a dual core Intel Xeon 3000 series processor and 4GB of memory. Hardware virtualization was available but disabled for these preliminary experiments. Three platforms were used: two different virtualization platforms and Linux. The Linux distribution used was CentOS 5.2 on all the real or virtual machines. We intend

this framework to be platform agnostic, as described in Section 4. As our results do not serve as a definitive comparison of virtualization platforms, we have anonymized their representation in this paper.

2.3 Identifying Variability

It is important to note that several factors need to be recognized as sources of variability. These factors require attention and tuning in ways that will be appropriate for conditions and workloads specific to the particular systems being benchmarked. Below we list these factors, and the way we mitigated each in our own use of the micro-benchmarks.

- Data is buffered before it is sent: TCP Naggle is turned off, and we intentionally keep the size of the messages below 1 packet.
- Other network traffic: in our case this is minimized as we run the tests on a private network.
- Other processes delaying reaction time by having the CPU: again in our case we have the control to ensure the tests are run on a lightly loaded system. All non-essential processes and daemons are disabled in the test systems.
- Exact time measurements: accuracy can be compromised as switching cores (and clocks) may introduce variability. A control test was run to see if this would be an observable factor, it was not.

2.4 Results

The communication micro-benchmark detailed in Section 2 was run 50 times, with 10,000 iterations in each run. The benchmark was set to make one hop, so it would just comunicate with itself. The resulting data set is rendered as a mesh diagram in Figure 2. The x-axis of this diagram represents the run number, the y-axis represents the iterations of each run, and the z-axis represents the time a single iteration took.

3 Analysis of Communication Costs

This data was rendered as a mesh diagram to show some of its interesting properties that are not present in the non-virtualized runs. The first artifacts of interest in Figure 2 are the horizontal lines that appear at regular intervals. Those lines are specific iterations of the experiment which took longer. They tended to be about twice as long, moving from the approximately 50 μsecs to 100 μsecs. This shows that after a certian number of operations, the system takes longer to perform the timed operation for an iteration.

The second artifact of interest is the larger spikes that occur diagonally across runs. These spikes tend to be about 4 times slower than the regular runs. What is interesting about these artifacts is that they occur between runs, and of further consequence for testing scenarios, the pattern of these spikes reveals that the system has a memory between runs. The following subsection describes further benchmarks designed to explore the patterns of artifacts such as those shown in Figure 2.

3.1 Origins of Artifacts

To further diagnose origins and possbile ramifications of these patterns, we added two more micro-benchmarks to the framework. These are specifically designed to establish characteristics of networking code and system calls in general. In these benchmarks we removed all the networking code from our timed section. In the first benchmark we left no system calls in the timed region, in the second benchmark we placed a single system call in the code where the networking was occurring. The system call used was a `puts` call, writing a simple string to standard out.

Benchmarks such as these, with no networking, provide insight into the artifacts described above. When all the system calls were removed the results had no artifacts on any platfrom. When the non-networking system call is added in, the horizontal lines reappear. Upon closer inspection, the lines are there on every platform, but on the virtualized platforms they are much larger. We hypothosize that these lines are buffer flushes from the processes standard out buffer. That would account for their pattern and regularity. A buffer flush is also likely to be an operation that would take longer in a virtualized environment.

Finally, in the benchmark where the networking call was added back into the code, the diagonal patterns reappear in the virtualized environments. This indicates the networking code is this cause of the diagonal lines. This also can likely be explained by buffers in the networking system calls. Further micro-benchmarks can be developed according to this ”null test” pattern to explore origins of similar artifacts for several different testing scenarios.

4 A Platform-Agnostic Perspective

There are a growing number of strategies used by current virtualization platforms to allow virtual machines to work correctly and efficiently [7, 10]. Due to the complexity and variation in platforms, it is important to keep the development of a framework for micro-benchmarks platform-agnostic, so as to make it possible to even understand trade-offs between platforms under different configurations and load.

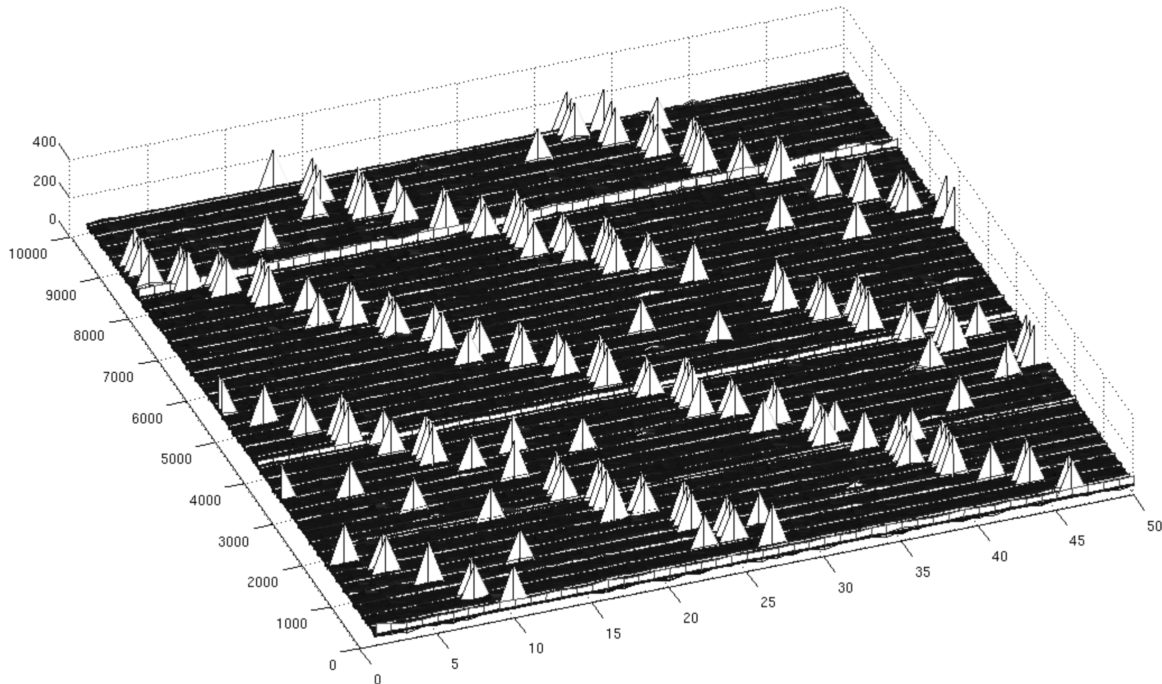


Figure 2. A mesh diagram of the round trip packet times. Iterations on the y-axis, experimental runs on the x-axis, and response time on the z-axis. Note the memory between runs, and at certian iterations

Platform	mean	std. dev.	min	max
Linux	54.40 μsec	5.05 μsec	52 μsec	421 μsec
VM1	50.36 μsec	5.53 μsec	32 μsec	269 μsec
VM2	81.85 μsec	30.82 μsec	74 μsec	2036 μsec

Table 1. Microbenchmark running on three different platforms: Linux and two modern virtualization platforms.

As a starting point, we extended the results of the framework to include two different virtualization platforms. The intent of this experiment was to see what impact the virtualization platfrom had on the benchmark. This experiment was run on two state-of-the-art virtualization packages, and on a plain Linux installation. They were all configured with the Linux distribution CentOS 5.2, and were stripped of all non-essential services.

4.1 Results and Analysis

Table 1 shows some summary statistics from the runs. All the runs were similar in distribution, though VM2 was more variable. Although VM2 was on average slower than VM1 and Linux, it actually finished its runs before VM1. We attribute that to slowdowns in VM1 from areas of code in the experiment that were not timed. In this experiment, in terms of timing VM1 behaved very closely to Linux where are VM2 was slower and more variable.

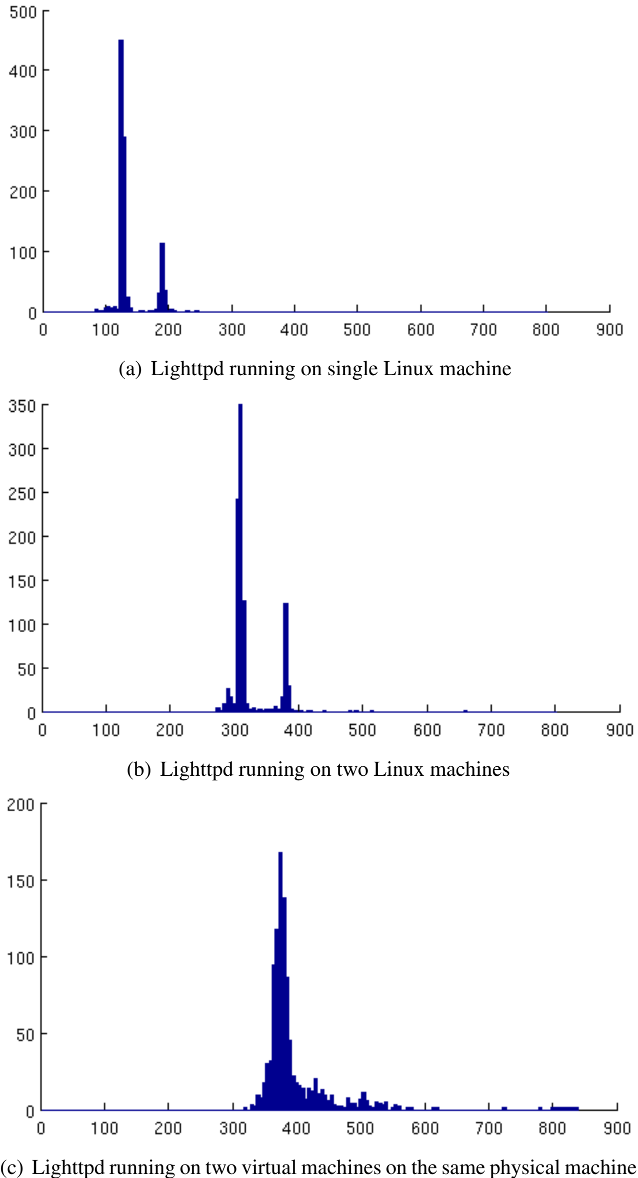
In the case of this workload and these test platforms, the choice of virtualization platform changed the magnitude of the results and their variability. In the next section we see a experiment where virtualization has a completely different effect.

5 Framework Customization

This third set of experiments was aimed to help relate the above results to a more realistic scenario. These experiments record the response times for several configurations of lighttpd [1], a popular light weight web server.

Using the same measurement framework from Section 2 we sent HTTP GET requests to lighttpd, then recorded the time for lighttpd to respond with a 44 byte *index.html* file. The experiment was run on the same systems mentioned in Section 2.2, and three configurations were tested:

Single physical machine: client and server were run on a single Linux machine.



(a) Lighttpd running on single Linux machine

(b) Lighttpd running on two Linux machines

(c) Lighttpd running on two virtual machines on the same physical machine

Figure 3. Lighttpd running on three configurations.

Two physical machines: client on one machine, server on the other. The machines were connect by a gigabit network on a switch with no other equipment.

Virtualized: client and server in different virtual machines on the same physical machine.

In these experiments the test setup was run 1000 times. Figure 3 shows histograms of the result of these three configurations.

The histograms highlight an interesting artifact of the virtualization. Figure 3(a) shows the plain Linux instance of the experiment on one host, which produces a bimodal distribution. Similarly, Figure 3(b) shows the same test spread over two physical hosts, also producing a similar bimodal distribution shifted to the left. As shown in Figure 3(c) in this experiment the virtualization actually significantly changes the distribution of the results. Furthermore, this change is not just a linear shift of the results, but an actual change in the shape of the distribution when the web server is virtualized in two domains on the same physical host.

6 A Framework for Micro-Benchmarks

By coupling a framework for micro-benchmarks with a systematic approach for quantifying artifacts in virtualized systems, we were able to establish particular tradeoffs that must be considered when timing and evaluating system performance in virtualized environments.

In a simple communication micro-benchmark, we were able to drill down more deeply into patterns of artifacts in the virtualized timing data, and more accurately consider their origins. In a application-specific extension of the micro-benchmark, we established that virtualization changed both the magnitude of the data, and the distribution. This quantification allows us to more carefully consider the ways in which tests such as these will map into real systems at scale.

Our results show that artifacts of virtualization can be quantified in meaning ways, and moreover that testing at-scale using virtualized clusters require attention to the results of these and other similar benchmarks. Our numbers do not indicate that virtualization makes systems run slower on all operations, but that certain operations do take longer, and test configuration can play into these factors. The degree to which a virtualized deployment differs from a physical system is dependant on workload characteristics, and is not scaled linearly from the physical case.

7 Conclusion and Future Work

Virtualization has enabled testing of large-scale systems in significantly smaller-scaled environments. When considering how these virtualized results will map to real machines, it is important to be able to accurately characterize and quantify artifacts that arise in the testing environment.

In this paper we have shown how the proposed framework can be used to begin to identify artifacts that can then be pursued, (a) more deeply, such as described in Section 3.1 and/or more generally in the context of application-specific needs, such as described in Section 5. The framework is platform-agnostic, and applicable to further architectures than those described here. We are currently deriving further tests in an 8 core environment. We further plan

to experiment with the ways in which these benchmarks can be effectively used to anticipate the impact of migrating fine grained VM implementations transparently between nodes in a system.

Whether a deviation in timing between virtualized and real deployments is important depends on the semantics of a particular experiment. A few places we anticipated it will be important are:

- finding race conditions in the cluster and
- accurately simulating cluster performance under load.

In our future work we hope to populate the framework with further testing harnesses to help address these specific concerns.

References

[1] Lighttpd: fly light, 2007. http://www.lighttpd.net/.

[2] Citrix XenServer 5: Virtualization for every server in the enterprise. Website, 2008. http://www.citrix.com/English/ps2/products/product.asp?contentID=683148.

[3] Intel(R) Architecture Software Developer's Manual, Volume 2: Instruction Set Reference Manual. Website, 2008. http://developer.intel.com/design/intarch/manuals/243191.htm.

[4] Microsoft Virtualization: Home. Website, 2008. http://www.microsoft.com/virtualization/.

[5] Sun Virtualization Solutions. Website, 2008. http://www.sun.com/solutions/virtualization/.

[6] VMware: Virtualization via Hypervisor, Virtual Machine & Server Consolidation. Website, 2008. http://www.vmware.com/.

[7] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 2–13, New York, NY, USA, 2006. ACM.

[8] AMD. *AMD64 Virtualization Codenamed Pacifica Technology: Secure Virtual Machine Architecture Reference Manual*, 3.01 edition, May 2005.

[9] P. Apparao, S. Makineni, and D. Newell. Characterization of network processing overheads in xen. In *VTDC '06: Proceedings of the 2nd International Workshop on Virtualization Technology in Distributed Computing*, page 2, Washington, DC, USA, 2006. IEEE Computer Society.

[10] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauery, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP'03*, pages 1–14, 2003.

[11] A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel. Diagnosing performance overheads in the xen virtual machine environment. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 13–23, New York, NY, USA, 2005. ACM.

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

[13] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, 1974.

[14] C. A. Waldspurger. Memory resource management in VMware ESX server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, 2002.