SAND2019-8232C

Lessons Learned from 10k Experiments to Compare Virtual and Physical Testbeds

Jonathan Crussell, Thomas M. Kroeger, David Kavaler, Aaron Brown, and Cynthia Phillips
Sandia National Laboratories
{jcrusse, tmkroeg, dkavale, aarbrow, caphill}@sandia.gov

Abstract

Virtual testbeds are a core component of cyber experimentation as they allow for fast and relatively inexpensive modeling of computer systems. Unlike simulations, virtual testbeds run real software on virtual hardware which allows them to capture unknown or complex behaviors. However, virtualization is known to increase latency and decrease throughput. Could these and other artifacts from virtualization undermine the experiments that we wish to run?

For the past three years, we have attempted to quantify where and how virtual testbeds differ from their physical counterparts to address this concern. While performance differences have been widely studied, we aim to uncover behavioral differences. We have run over 10,000 experiments and processed over half a petabyte of data. Complete details of our methodology and our experimental results from applying that methodology are published in previous work. In this paper, we describe our lessons learned in the process of constructing and instrumenting both physical and virtual testbeds and analyzing the results from each.

1 Introduction & Background

Cyber experimentation allows researchers to explore what-if scenarios in a scientifically rigorous manner. Cyber experimentation typically relies on virtual testbeds which run real software on virtualized hardware, also known as *emulation*. Since emulation runs the actual software it can capture unknown or complex behaviors that would not be captured in a *simulation*. In some cases, researchers may have access to physical testbeds, but these are generally more expensive and less-reconfigurable than their virtual counterparts. To date, there has been limited research into where and how virtual testbeds differ from physical testbeds.

Previous work has shown that virtualization, the basis for virtual testbeds, increases network latency and lowers throughput [11, 16, 20, 23]. Wang *et al.* [20] showed that for small bursts, buffering can even cause virtual machines to receive data at rates exceeding that of the underlying network. Differences like these and other anomalies could undermine the result of an experiment run on a virtual testbed. Furthermore, variation in the construction of virtual testbeds and the underlying hardware compounds the uncertainty.

Three years ago, we set out to quantify the differences between virtual and physical testbeds. We proposed to run representative workloads on both physical and virtual testbeds, collect and compare metrics from each to understand where and how they differ. For our workload, we used *ApacheBench* [8] as an HTTP client to request fixed pages from an HTTP server. We compared metrics from the application workload, how the workload interacts with the operating system, and how the traffic traverses the network. These different levels of abstraction allow us to understand where differences occur and can inform experiments that may be sensitive to artifacts at different levels. In doing so, we aimed to create a handbook for experimenters to provide recommendations and pitfalls when building experiments. Complete details of our methodology and our experimental results from applying that methodology are published in previous work [4]. In this paper, we focus on our lessons learned in the process of constructing and instrumenting both physical and virtual testbeds and analyzing the results from each. This includes trial and error, and how and why we ended up using the methods that we did, both of which were out of scope for the previous paper.

In total, we ran over 10,000 experiments across three clusters producing more than half a petabyte of data. We varied the payload size, network drivers, and network bandwidth and found notable differences in the system and network-level interactions. We found that network driver *offloading* behavior varied greatly between testbeds causing differences in the number of *read* system calls and packetization. When we compared the sequences of system calls using Markov chains, we found near-identical structures across testbeds. In subsequent experiments, we found many system call sequences to be identical once we collapsed read/polling loops.

We performed our experiments using the *minimega* [12] toolset. The primary tool, *minimega*, is an orchestration tool

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

that allows users to create and manage containers, virtual machines, and virtual networks using a scriptable API. It leverages a custom container implementation, QEMU [1], and Open vSwitch [7]. Auxiliary tools in the toolset include *vmbetter*, a tool to build VMs and containers, *protonuke*, a simple traffic generation tool, and *igor*, a physical node scheduling tool. All these tools are open-source and available on Github.

Several of our lessons learned overlap with the motivations for DEW (Distributed Experiment Workflows) [13]. It is encouraging to see the overlap in challenges between different experimenters using different tools. Several other papers have enumerated lessons learned from creating or operating testbeds [2,3,10,22]. These lessons include selecting physical devices, creating software to instantiate the testbed, and managing user interactions with the testbed. We were unable to find lessons learned using these testbeds to run thousands of experiments. For a description of related works on comparing virtual and physical testbeds see our full paper [4].

The remainder of this paper is organized as follows. In Section 2, we discuss our lessons learned, grouped into four topic areas: Tools Validation and Development (Section 2.1), Instrumentation (Section 2.2), Data Collection & Aggregation (Section 2.3), and Data Analysis (Section 2.4). In Section 3, we conclude by discussing the applicability of our lessons learned to other testbeds and future work.

2 Lessons Learned

To perform experiments at scale, we rely on several tools to provide orchestration, instrumentation, data collection, and analysis. We primarily used *bash* as the glue between our tools and to define our experiments, instrumentation, and analysis. We also used *Python* and *SQL* in our analysis. As noted by others [13], an experiment definition language that would enable specifying experiments, instrumentation, collection and possibly even analysis would be very useful for such efforts. That said, flexibility and extensibility are critical to the success of any such language.

2.1 Tool Validation and Development

Lesson: Using a testbed toolset for experimentation requires substantial effort and consideration to put tools together in a workable and validated fashion.

Improving *minimega*. Running thousands of repetitions of the same experiment seems simple, but we identified several issues. We use tools from the *minimega* toolset (*minimega*, *protonuke*, *igor*, and *vmbetter*), *sysdig* [9], *parallel* [19], *SQLite* [18], *vmstat* [21], and *tcptrace* [15]. We found that we needed to add several features to the *minimega* toolset before we could run our experiments. In addition to many minor bug fixes and improvements, we discovered a rare bug in the capture API that would crash *minimega*. This API is used to record network packets for analysis. The crash had not occurred during our previous experiments where we might only run a small number of long-running captures. Additionally, we added a snaplen feature to the capture API to reduce the data size by only capturing packet headers. We also found several examples where state was not entirely cleaned up after an experiment finished. For example, there was a bug which caused the container filesystem to not unmount. We view these as a gap between how *minimega* is typically used (to instantiate an environment a handful of times) and how we wished to use it (to run many experiments with varying parameters). To run thousands of experiments successfully, toolsets must be well-tested, reset all state between tests, and properly handle edge cases.

Stand-alone C2 server. In addition to improving our tools, we also added a tool to the toolset for our physical testbed: a stand-alone command-and-control server. When using the command-and-control layer with VMs, *minimega* acts as the server. We repurposed the command-and-control server from *minimega* to control the physical nodes. This had the added benefit that our scripts to run the experiment on the virtual testbed are similar to the scripts to run the physical testbed. In an ideal setting, the same experiment would run on either testbed so that separate versions did not have to be updated and maintained in parallel. This could be achieved, for example, using DEW with different experiment realizations.

Parallelizing experiments. Another challenge was parallelizing our experiments. To avoid one crash bringing down many experiments, we created many three-node reservations using the cluster provisioning tool, *igor*. Each reservation had a head node and two nodes to run the VMs. We then used *parallel* to distribute jobs across the reservations. During reservation creation, *igor* partitions the experiment network directly on the cluster switch using VLANs, so we did not have to worry about leakage between reservations. To use *parallel* in this manner, we exposed all parameters that we wanted to test as arguments to our top-level script and ensured that all the head nodes had all dependencies (achieved through a custom automated prep script). Using these methods we ran up to 16 experiments simultaneously (using 48 nodes total) and dramatically reduced the runtime.

Robust snapshot feature. We used *vmbetter* to construct our experiment images. This tool allowed us to create minimal kernel and init ramdisks containing just the desired software. To decrease the time to run experiments, we pre-installed all needed software on the VMs so that we did not have to include the install overhead in every test. In the future, a more robust snapshot feature could alleviate the need for customized images.

2.2 Instrumentation

Lesson: Instrumentation is invaluable but it is often manually added, expensive, and experiment-specific. Integrating more forms of instrumentation into the toolset could

help researchers to more rapidly develop experiments.

We needed two forms of instrumentation in our experiments: 1) to verify the functionality of the environment and 2) to understand and evaluate our experiments.

Verifying the environment can be as simple as checking that the endpoints can ping each other but may be more elaborate, depending on how the endpoints are customized after booting. We found that the *minimega* toolset has some capabilities to perform these verification checks but they must be created by each experimenter. Integrating these checks into the toolset could simplify the process for experimenters.

Instrumenting at multiple levels. To understand how the workload behaves in both the virtual and physical testbeds, we instrumented the systems at three levels: application, operating system, and network. Using *ApacheBench* as our application workload provided well-tested and detailed metrics for the application level. These metrics made testbed comparisons at this level robust and insightful but we encountered difficulties at the other levels.

At the operating system-level, we found that we could not capture system call traces for containers in the same was as we did for VMs and physical machines. Because containers have limited access to the system resources, we were unable to run *sysdig* to capture the traces from within the container. Instead, we had to capture system call traces system-wide and then filter the captures based on process namespace. When we did so, we found that *sysdig* had dropped events when there were many containers running, corrupting the results.

At the network-level, we had a discrepancy in packet-capture locations. For VMs and containers, we performed the captures on the physical hosts whereas for the physical testbed, we captured the traffic on the host running the workload. This minor variation causes changes introduced by the virtual NIC to be masked from view.

In an ideal world, we would have instrumentation to understand the provenance of all events in the operating system and underlying network resulting from the application under test. However, there were numerous constraints on capturing the level of data necessary to perform these traces. There are substantial resource costs associated with capturing the instrumentation data at this fine of a granularity. When we disabled the instrumentation, the experiments were able to run 5-13% faster, and we were still not capturing nearly the fidelity necessary to comprehensively trace the experiment. The generated data sets themselves were also enormous, with the packet captures being on the order of several gigabytes worth of data for a single run. Sampling might mitigate some of these costs, but tuning the sampling to avoid missing low probability events can be difficult.

Level of effort. Another substantial cost associated with instrumentation is the effort required to enable the level of instrumentation. With the exception of packet capture which was easily configurable in *minimega*, all of our instrumentation needed to be built or integrated by hand. More detailed tracing of the execution could also be done using tools like bpftrace [17] to track events through the Linux kernel level. However, doing so accurately would require substantial effort to handcraft the tracing. Little of this level of instrumentation has been provided in the orchestration frameworks. Most tools provide instrumentation to enable functional or performance debugging, not to capture and model application behavior.

Nevertheless, our instrumentation was invaluable to track down anomalous behaviors. For example, in analyzing the *ApacheBench* results for the *e1000* driver, we found that a small subset of results significantly deviated from the rest. From *tcptrace*, we discovered that there were stalls (typically 13 seconds) where the server behaved as if it had never received a data *ACK* even though it was sent by the client. Further investigation of our kernel instrumentation showed that these stalls coincided with driver lockups and resets.

2.3 Data Collection & Aggregation

Lesson: Testbeds can provide a wealth of data to researchers but should do more to streamline the process of collecting and aggregating it into a usable form.

One challenge with our instrumentation was collecting it all for further processing. We had two forms of data: data collected within the VM and data collected from the host running the VM. Collecting data from the host running the VM went smoothly using the *minimega* file transport APIs. Collecting the data that the VM generates was more challenging.

Data collection. We explored several options to collect data from the VMs. First, we tried using *minimega*'s command-and-control APIs to retrieve the files but that has limits on the file size. Instead, we created a *qcow2* disk where the VM would write all of its data. After the experiment, we would then process the *qcow2*s to extract the instrumentation data. However, in cases where the VM did not cleanly shutdown we occasionally had corrupted the filesystem. This was often repairable but did lead to some data loss. We recently implemented a capability to mount the guest filesystem on the host in the *minimega* toolset which could simplify this process going forward.

Data aggregation. Once we extracted data from the *qcow2* disks and from the other hosts in the experiment, we then processed it (using *sysdig* and *tcptrace*) to create results for the individual iterations of the experiment. We stored these results in an *SQLite* database. Once the iteration had been processed, we archived it to a storage cluster before continuing to the next iteration as the hosts have insufficient storage to keep hundreds of iterations. Once all iterations and parameters were complete, we aggregated the results from all of the individual iterations into a single database to use for our analysis. Processing and aggregating the results in this way allowed us to avoid reading all the instrumentation data from the storage cluster – we only needed to read the much smaller databases.

Storage reduction. We used *tcptrace* to process the

PCAPs that we collected. This tool reassembles TCP flows and computes statistics for each flow in the PCAP such as the number of bytes, packets, and *ACK*s. Our small-payload experiments, where we completed up to 500,000 requests, resulted in 500,000 flows, each with 78 flow statistics (39 for each direction). Storing the full results for later aggregation vastly increased the size of the database with little benefit; thus, we decided to compute summary statistics for the flow statistics (*e.g.* quantiles, mean, and standard deviation) in the database. When we then computed statistics over all iterations, we looked at the mean of means for these statistics. This reduces storage issues, but introduces issues with aggregation.

Information loss. With any level of aggregation, one loses information content. For example, in calculating a mean, we inherently lose information about the underlying statistical distribution. If the original distribution is parameterized solely by its mean, this aggregation is acceptable. However, the original distribution may also require a standard deviation to be adequately described. Thus, aggregation may hurt statistical robustness. With this in mind, we retained the original source in case we needed a closer look at the distribution.

2.4 Data Analysis

Lesson: Testbeds allow for many repetitions but care should be used when analyzing the data, especially in conflating statistical significance with practical importance.

As our dataset is large, we have the benefit of most measures being statistically significant. When performing statistical tests of significance, standard practice is to calculate some test statistic (*e.g.*, t-test) and associated p-values based on said test statistic [14]. Standard test statistics (and thus p-values) are proportional to sample size; *i.e.*, a larger sample size tends to force p-values lower. Thus, with many observations (as is the case in our work), most tests are likely to be highly significant *if the null hypothesis is false*. This can be seen as a positive outcome; researchers have a statistically robust argument towards validity of their stated hypotheses. However, this can be an issue when arguing *practical importance*.

For example, a researcher may wish to see if a given system configuration is associated with lower latency as compared to a baseline. They perform experiments and find this is indeed the case with a statistically significant p-value. However, this new configuration only decreases packet latency by 0.1%. Though this difference is statistically significant, the question remains: is a 0.1% decrease in packet latency of practical importance? This depends heavily on the application. Thus, statistical significance should not be the only issue in determining importance; *effect sizes* should also be considered.

Multiple comparisons. Related to the issue of p-values is that of *multiple comparisons* [5, 6]. One may wish to perform an analysis over all collected metrics in order to identify which are important for an outcome of interest. As is the case in our work, most metrics were not of importance *a priori*, and we wished to identify those which were *a posteriori*. In this situation, one may be tempted to compare each metric to the other and identify statistically significant differences. Though this process can be helpful in identifying important (and hitherto unknown) aspects, this can be dangerous if one does not adjust resulting p-values to account for multiple comparison. In essence, when comparing a large number of metrics against each other, there is a non-zero probability that a comparison will be statistically significant due to chance alone; this risk increases as the number of comparisons increase. This can lead to spurious significant results.

Small variance. During experimentation we found that variance within a given configuration for most metrics was extremely small. In most cases, standard deviation was so small it was almost numerically 0, and in some cases was numerically 0. Though standard practice suggests running as many iterations of an experiment as possible, this result suggests it may not be necessary for this type of work.

3 Conclusion & Future Work

We have presented our lessons learned from conducting over 10,000 experiments to compare virtual and physical testbeds. We have described the improvements we made to our tools, the challenges of instrumenting our experiments to understand performance and behaviors, aggregating and processing over half a petabyte of data, and analyzing it to produce meaningful results. We hope these lessons are useful to the cyber experimentation community.

While some of our lessons learned are testbed-specific (*i.e.* specific to the *minimega* toolset), we believe that many of them are applicable across testbeds. Specifically, our lessons learned around instrumentation, data collection & aggregation, and data analysis should be relevant to most general-purpose testbeds. Other, more specialized testbeds may not benefit from our lessons learned because they have been designed to instrument and analyze specific events.

The lessons learned around instrumentation, data collection & aggregation, and data analysis are also more fundamental. Creating a testbed toolset that is ready to use as an experimentation platform can likely be solved with more engineering. But creating more generalized ways to instrument and analyze the data from these experiments will require further research.

There are many more experiments to run to understand the differences between testbeds. Recently, we have begun experiments to understand the effects of contention. Specifically, instead of running a single HTTP client and server on their own dedicated physical machines, we measure effects when we run an increasing number of isolated pairs, up to 80 pairs. Preliminary results show that contention degrades the workload performance sooner than we expect. However, we have seen instances with two or four pairs where each *ApacheBench* instance has better throughput than a single pair. Through these experiments, we hope to understand how to

calibrate a cluster to the amount of workload it can support without, with some, and with prohibitive amounts of artifacts from contention. These experiments have further honed our toolset and techniques.

Acknowledgements

We would like to thank our anonymous reviewers and our paper shepherd for their helpful comments and suggestions.

Supported by the Laboratory Directed Research and Development program at Sandia National Laboratories, a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.

References

[1] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, volume 41, page 46, 2005.

[2] Terry Benzel, Robert Braden, Dongho Kim, Anthony D Joseph, B Clifford Neuman, Ron Ostrenga, Stephen Schwab, and Keith Sklower. Design, deployment, and use of the DETER testbed. In *DETER*, 2007.

[3] Mashrur Chowdhury, Mizanur Rahman, Anjan Rayamajhi, Sakib Mahmud Khan, Mhafuzul Islam, Zadid Khan, and James Martin. Lessons learned from the real-world deployment of a connected vehicle testbed. *Transportation Research Record*, 2672(22):10–23, 2018.

[4] Jonathan Crussell, Thomas M. Kroeger, Aaron Brown, and Cynthia Phillips. Virtually the same: Comparing physical and virtual testbeds. In *2019 International Conference on Computing, Networking and Communications (ICNC)*, pages 847–853, Feb 2019.

[5] Olive Jean Dunn. Estimation of the means of dependent variables. *The Annals of Mathematical Statistics*, pages 1095–1111, 1958.

[6] Olive Jean Dunn. Multiple comparisons among means. *Journal of the American statistical association*, 56(293):52–64, 1961.

[7] Linux Foundation. Open vSwitch. http://www.openvswitch.org/, 2018.

[8] The Apache Software Foundation. ab – Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html, 2019.

[9] Sysdig Inc. sysdig. https://sysdig.com/, 2019.

[10] Sushant Jadhav, Timothy X Brown, Sheetalkumar Doshi, Daniel Henkel, and RG Thekkekunnel. Lessons learned constructing a wireless ad hoc network test bed. In *First Workshop on Wireless Network Measurements*, 2005.

[11] Aravind Menon, Jose Renato Santos, Yoshio Turner, G John Janakiraman, and Willy Zwaenepoel. Diagnosing performance overheads in the xen virtual machine environment. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 13–23. ACM, 2005.

[12] minimega developers. minimega: a distributed vm management tool. http://minimega.org/, 2018.

[13] Jelena Mirkovic, Genevieve Bartlett, and Jim Blythe. DEW: Distributed experiment workflows. In *11th USENIX Workshop on Cyber Security Experimentation and Test (CSET 18)*, Baltimore, MD, 2018. USENIX Association.

[14] David S Moore, George P McCabe, and Bruce A Craig. *Introduction to the Practice of Statistics*. WH Freeman New York, 2012.

[15] Shawn Ostermann. tcptrace. http://www.tcptrace.org/, 2003.

[16] Luigi Rizzo, Giuseppe Lettieri, and Vincenzo Maffione. Speeding up packet I/O in virtual machines. In *Proceedings of the ninth ACM/IEEE symposium on Architectures for networking and communications systems*, pages 47–58. IEEE Press, 2013.

[17] Alastair Robertson. High level tracing language for linux ebpf. http://github.com/iovisor/bpftrace.

[18] SQLite Consortium. SQLite. https://sqlite.org/, 2019.

[19] O. Tange. Gnu parallel - the command-line power tool. *;login: The USENIX Magazine*, 36(1):42–47, Feb 2011.

[20] Guohui Wang and TS Eugene Ng. The impact of virtualization on network performance of amazon EC2 data center. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–9. IEEE, 2010.

[21] H Ware and F Frederick. Vmstat man pages. *Linux Systems, calling the man command for vmstat*, 2011.

[22] Kirk Webb, Mike Hibler, Robert Ricci, Austin Clements, and Jay Lepreau. Implementing the Emulab-PlanetLab Portal: Experience and lessons learned. In *WORLDS*, 2004.

[23] Jon Whiteaker, Fabian Schneider, and Renata Teixeira. Explaining packet delays under virtualization. *ACM SIGCOMM Computer Communication Review*, 41(1):38–44, 2011.