A Methodology and Framework to Determine the Isolation Capabilities of Virtualisation Technologies

Simon Volpert
Ulm University
Ulm, Germany
simon.volpert@uni-ulm.de

Benjamin Erb
Ulm University
Ulm, Germany
benjamin.erb@uni-ulm.de

Georg Eisenhart
Ulm University
Ulm, Germany
georg.eisenhart@uni-ulm.de

Daniel Seybold
Ulm University
Ulm, Germany
daniel.seybold@uni-ulm.de

Stefan Wesner
University of Cologne
Cologne, Germany
wesner@uni-koeln.de

Jörg Domaschka
Ulm University
Ulm, Germany
joerg.domaschka@uni-ulm.de

ABSTRACT

The capability to isolate system resources is an essential characteristic of virtualisation technologies and is therefore important for research and industry alike. It allows the co-location of experiments and workloads, the partitioning of system resources and enables multi-tenant business models such as cloud computing. Poor isolation among tenants bears the risk of noisy-neighbour and contention effects which negatively impacts all of those use-cases. These effects describe the negative impact of one tenant onto another by utilising shared resources.

Both industry and research provide many different concepts and technologies to realise isolation. Yet, the isolation capabilities of all these different approaches are not well understood; nor is there an established way to measure the quality of their isolation capabilities. Such an understanding, however, is of uttermost importance in practice to elaborately decide on a suited implementation. Hence, in this work, we present a novel methodology to measure the isolation capabilities of virtualisation technologies for system resources, that fulfils all requirements to benchmarking including reliability. It relies on an immutable approach, based on Experiment-as-Code. The complete process holistically includes everything from bare metal resource provisioning to the actual experiment enactment.

The results determined by this methodology help in the decision for a virtualisation technology regarding its capability to isolate given resources. Such results are presented here as a closing example in order to validate the proposed methodology.

CCS CONCEPTS

• **General and reference** → **Performance**; • **Computing methodologies** → *Modeling methodologies.*

KEYWORDS

Isolation, Virtualisation, Benchmarking, Framework

ACM Reference Format:
Simon Volpert, Benjamin Erb, Georg Eisenhart, Daniel Seybold, Stefan Wesner, and Jörg Domaschka. 2023. A Methodology and Framework to Determine the Isolation Capabilities of Virtualisation Technologies. In *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering (ICPE '23), April 15–19, 2023, Coimbra, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3578244.3583728

1 INTRODUCTION

Virtualisation has been driving the vision of software-defined infrastructure for almost two decades and is a major enabler for cloud computing. Since its early days with Virtual Machines, the landscape of tools, methodologies, approaches, and concepts has continuously increased. Currently, users and software architects are faced with a vast landscape of virtualisation concepts and technologies. Furthermore, the speed at which new approaches are researched, new toolkits come up, and new software versions are being released has not slowed down. Accordingly, it is increasingly hard to keep track of all changes in the domain and even harder to see different market claims.

There is work helping to understand the impact of virtualisation technologies and their differences. This ranges from performance impact and start-up time [26] over security considerations [3] towards isolation capabilities [44]. Yet, there is no holistic approach for their comparison and classification and in consequence no baseline for any decision in favour or against a virtualisation technique.

Besides its impact on performance, a crucial differentiator for any type of virtualisation technology is its capability to ensure isolation of concurrent workloads and to withstand competing and possibly interfering workloads. This is due to the fact that virtualisation is the core technology for e.g. server consolidation [18] in order to reduce total cost of infrastructure ownership [22]. In addition, it is also the baseline of many cloud computing services where the multi-tenant aspect causes a high incentive for proper isolation mechanisms [25].

While much work exists to determine the performance impact of a virtualisation technology, only little practical work has been done on the quantification of isolation capabilities. Literature provides insight how to compute the isolation of environments, but there is no established, widely adaptable methodology on how to actually measure it. More precisely, for being able to quantify the isolation capabilities of a wide range of different virtualisation technologies, the following research questions need to be answered: *(i)* which workloads and benchmarks are suited for driving the such an evaluation and which hardware resources should be considered? *(ii)* which measurement technologies are available to support measuring isolation for a wide range of virtualisation technologies? *(iii)* which evaluation methodology reduces disturbances and increases repeatability.

This paper proposes a generally applicable benchmark-based evaluation methodology supporting the evaluation of both performance degradation and isolation capabilities of a wide range of virtualisation technologies. More precisely, we present the following contributions:

(1) The evaluation methodology for the multidimensional evaluation of isolation capabilities and performance degradation. Much care has been taken to address typical, different types of hardware resources, but also to keep the methodology open with respect to workload generation and other types of tooling.
(2) A proof-of-concept implementation of the methodology as a benchmark-based evaluation framework with a strict focus on aspects such as reproducibility, automation, and fine-grained profiling.
(3) A validation of the proof-of-concept implementation of the methodology measuring the isolation capabilities of `podman` representing a container-based virtualisation technology with respect to CPU, storage, memory, and networking.

The remainder of this paper is structured as follows: At first, background information is presented in section 2 highlighting some essential concepts referred to across this paper. This is followed by related work in section 3. It further includes a brief reflection of their respective findings put into contrast of our work. Subsequently, an assumed system model is discussed in section 4 including a presentation of essential requirements the isolation measurement methodology imposes. This methodology is instantiated in section 5 by the implementation of a framework. This framework satisfies all previously defined requirements and describes experiment workflows in detail. Finally, the measurement methodology is validated in section 6. This is concluded with a discussion in section 7 and a summary with outlook in section 8.

2 BACKGROUND

This section explains some fundamental and to this paper essential considerations. It briefly discusses virtualisation and presents an approach for their classification. Subsequently benchmarking, profiling, and their relation in the context of this work are discussed.

Isolation capabilities of various different virtualisation technologies have been researched in the past using different methods and different terminology [20]. Therefore, this section concludes with the discussion of the term "isolation" itself. Using this and related terms, models for isolation quantification are further discussed.

2.1 Virtualisation Landscape

The possible manifestations of virtualisation technologies are manifold. This section briefly presents the fundamental categories of those including their respective enabling concepts. Namely, these are the three virtualisation classes hypervisor, container and sandbox. A fourth one named hybrid is an encompassing term to denote those, that share characteristics of all classes. This classification is represented in fig. 1.

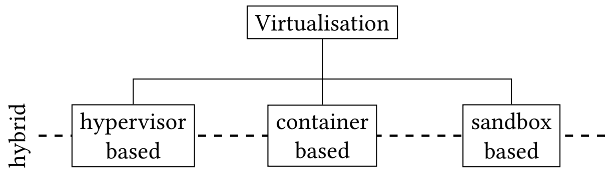


Figure 1: Virtualisation Classification Overview

Hypervisor-Based. Goldberg [12], initially subdivided Hypervisor-based virtualisation into two categories; Type-1 and Type-2. The main distinction among them is whether it runs directly on the hardware, or on top of another Operating System (OS). While these two distinctions categorise hypervisors, further significant properties are important. Hwang et al. [15] describes essential ones highlighting three approaches on how the actual virtualisation layer can be provided. These namely are *(i)* Full Virtualisation, *(ii)* Paravirtualisation and *(iii)* Hardware Assisted (HWA) Virtualisation. Simply put, they describe the degree a virtualisation technology makes use of special hardware functionality and the amount of system calls intercepted by the Virtual Machine Monitor (VMM).

Container-Based. Container-based virtualisation or sometimes called "OS-Virtualisation" is a widely applied approach on Linux systems. They leverage the kernel functionalities *(i)* namespaces, *(ii)* cgroups and *(iii)* capabilities. *(i)* Namespaces[1] isolate system specific resources. It does so by wrapping them into an abstraction, in order to present them to a process [5]. This enables processes to yield completely different views of a system compared to the host system. *(ii)* cgroups are a Linux feature that allows fine-grained control over different system resources [14]. More specifically, it enables to limit access to them. Lastly, *(iii)* capabilities[2] allow or prohibit the execution of specific operations. They can be granted to a user or group.

Sandbox-Based. Sandboxes can be created by utilising system call filtering provided by the Kernel [43]. Intercepting and thus filtering those system calls can be achieved by different levers. Among those are *(i)* ptrace and *(ii)* Seccomp-BPF.

2.2 Benchmarking

The purpose of benchmarking is the testing of performance in a controlled manner.

Definition 2.1 (Benchmarking). A benchmark is a tool coupled with a methodology for evaluating and comparing systems, or components thereof, with respect to specific characteristics, such as performance, energy efficiency, reliability, or security [20].

[1]https://man7.org/linux/man-pages/man7/namespaces.7.html
[2]https://man7.org/linux/man-pages/man7/capabilities.7.html

To achieve their goals, benchmarks apply a range of different workload ranging from artificial to real-world [13, 20] mostly classified as *(i)* micro-benchmark, *(ii)* simulation benchmark, *(iii)* replay benchmark, and *(iv)* production benchmark. Our work relies on the application of micro-benchmarks. These small, isolated, artificial benchmarks stress a single resource such as CPU, cache, system calls etc. [30]. Usually, they are simple to configure, repeatable and quick to run.

Figure 2 highlights a superficial view on how a benchmarking process looks like. The steps are *(a)* configuration, *(b)* load generation in parallel with *(c)* profiling and *(d)* resulting metric(s) export.

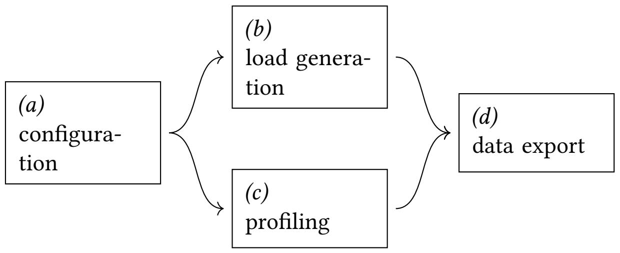


Figure 2: Superficial benchmarking workflow

Broadly speaking, the *(b)* load generation phase can be implemented as two different approaches. Both will be briefly discussed in the following.

Fixed-Work. These benchmarks keep the amount of work W_f a benchmark needs to perform fixed, while the time t_v it takes to do so is variable. The resulting speed S_W is the ratio between both of them, as illustrated in eq. (1).

$$S_W = \frac{W_f}{t_v} \tag{1}$$

Fixed-Time. benchmarks keep the duration of observation time t fixed, whereas the amount of work W being done becomes variable. The resulting speed S_T is the ratio between both of them as illustrated in eq. (2). Here it is assumed that more execution of a workload in a fixed amount of time adds value to the result.

$$S_T = \frac{W_v}{t_f} \tag{2}$$

2.3 Profiling

Profiling of systems is the act of analysing specific aspects of a workload. These include CPU cycles, memory consumption, system call frequency or disk write latency. It does so by instrumentation. This instrumentation can happen within an application's source code or its binary. Moreover, this is not limited to applications but is also possible on the OS level. Instrumentation itself is unspecific and can range from simple emitted counters to complex event tracing.

Together with benchmarking, profiling is an essential aspect of this work as already hinted in fig. 2. The possibilities regarding the sampling precision, measurement overhead and exactness are important upon crafting a method and selecting a tool or instrumentation decision.

In contrast to benchmarking, profiling does not create a workload or performs an experiment. It solely relies on available instrumentation to observe. Profiling enables the creation and determination of workload characteristics, that are already in place.

Here, a distinction between workload and OS profiling is made. The first one implies fine-grained profiling of workload or application processes, whereas the latter means a more abstract view on the whole system. As mentioned, in order to profile application specific aspects, source code instrumentation is necessary. However, the Linux Kernel offers a lot of performance instrumentation available to the user. These can be retrieved system-wide as well as per process and is thus able to precisely target a workload. The profiling subsystem of Linux is called "perf_events". It is also known as Performance Counters for Linux (PCL), Linux Performance Events (LPE) or Performance Monitoring Counter (PMC). It supports a multitude of events to be worked with, including *(i)* counter, *(ii)* tracepoints, *(iii)* kprobes, *(iv)* uprobes and *(v)* USDT [13].

2.4 Isolation Terminology

In order to define isolation in general, it is helpful to investigate on the term itself and related ones. As previously mentioned, a motivating reason for isolation, is the realisation of a fair environment among tenants. This is independent of the scope of cloud computing and holds true for any multi-tenant application. In his dissertation, Rouven Krebs [33] defines fairness by stating three aspects:

Definition 2.2 (Fairness).

(1) Tenants working within their assigned quota should not suffer performance degradation from tenants increasing their workload.
(2) Tenants exceeding their quotas more should suffer higher performance degradation than tenants exceeding their quota less.
(3) Tenants exceeding their quota should not suffer performance degradation, if tenants that comply with their quota are unaffected.

Hereby, quota is defined as the amount of resources a tenant is assigned and therefore allowed to consume. A tenant is "abiding" if operating within these quotas, whereas it is "disruptive", if it operates outside it.

Following up on this definition is the definition of isolation. Isolation occurs when two distinct workloads on shared resources do not influence each other in a perceptive way [21, 44, 28]. A workload exceeding its assigned resources or quota, will not degrade another workload if sufficiently isolated. In contrast, a non-isolated system bears the risk that distinct workloads interfere. Again, Krebs et al. [21] defines performance isolation as follows:

Definition 2.3 (Isolation). Performance isolation is the ability of a system to ensure that tenants working within their assigned quotas (i.e., abiding tenants) will not suffer performance degradation due to other tenants exceeding their quotas (i.e., disruptive tenants).

Such a disruptive tenant is also called "noisy neighbour". It describes a workload, that exceeds its assigned resources or quota so much, that its peers are no longer able to fully utilise their own guaranteed resources. It is assumed, that each workload is given a fixed amount of resources, governed by an owner. Within the scope of cloud computing they are typically written in a mutually agreed on and signed Service Level Agreement (SLA) and imposed by virtualisation technologies. According to Longbottom [25], the noisy neighbour is defined as follows:

Definition 2.4 (Noisy Neighbour). A workload within a shared environment is utilising one or more resources in a way, that it impacts other workloads operating around it.

2.5 Isolation Quantification Models

A simple approach to quantify isolation, is the calculation of the "performance loss rate"[39, 20, 44]. It foremost requires to measure a baseline performance of a workload p_a in an uncontended environment. Subsequently, the same workload in addition to an interfering workload is started, resulting in the workload performance p_b. Hereby, two workloads compete against resources. The isolation performance loss rate I_{plr} as rate between the difference of both performance measurements, can then be determined as shown in eq. (3). Additionally, the I_{ulr} in eq. (4) refers to the utilisation loss rate relative to the maximum possible utilisation a resource can achieve.

$$I_{plr} = \frac{|p_a - p_b|}{p_a} \tag{3}$$

$$I_{ulr} = \frac{|p_a - p_b|}{p_{max}} \tag{4}$$

As Rouven Krebs [33] points out, further models, approaches and metrics are considerable. However, analysing and comparing them is beyond this work and is considered as a further research challenge.

3 RELATED WORK

Literature in the domain of analysing the capabilities of virtualised workloads is very broad and has been published since the idea of virtualisation. However, the consideration of isolation among them is less pursued. It can be divided into several categories. The *(i)* analysis of isolation capabilities, the *(ii)* detection of noisy neighbour workloads and their *(iii)* workflow automation for measurements.

3.1 Isolation capabilities

Historically, benchmarking tools were not aware of virtualisation. Isolation capabilities of hypervisors left a lot to be desired. Koh et al. [19] highlight that and even claim, that low isolation is on purpose to a certain degree for debugging and development concerns. For that reason, Yuan et al. [46] developed a purpose built micro benchmark suite called VITS. This suite consists of six micro benchmark components, each benchmarking their own domain, namely: CPU, cache, memory bandwidth, memory utilisation, disk and network. While their results are primarily focused on the Xen hypervisor, they found that most isolation capabilities need improvement since resources sharing is still unfair.

Compared to our approach, they measure performance loss with a fixed-work based approach as briefly described in section 2.2. They, therefore, measure the amount of time a workload needs in an uncontended scenario and compare it with a contended scenario. Measurement takes place from within a virtual machine, to measure the impact in time. In contrast, our paper aims to express distinct resource utilisation degradation and thus pursues the black box measurement approach with a fixed-time workload.

Rahma et al. [32] follow a similar approach regarding the application of micro benchmarks. Among other characteristics they benchmark the isolation capabilities of CPU, disk and network within a Kernel Virtual Machine (KVM) Hypervisor backed OpenStack. While they do not give deep insight into the degree of isolation, they find disk isolation insufficient.

With the rise of more lightweight virtualisation approaches like container technologies, publications comparing them with traditional approaches had emerged. Tang et al. [39] compared recent container technologies isolation capabilities with KVM. Their proposed testing framework EIS gives a method of benchmarking these properties similar to [46]. Here, they applied a fixed-time workload, but performed the measurements from within the virtualisation technology. Compared to our paper's approach this limits the amount of visible resources since they are isolated. They find that CPU isolation works perfect, while other aspects suffer heavily from performance degradation, concluding that KVM is superior while still not optimal.

In order to compare isolation characteristics of virtualisation technologies, a quantifying model is necessary. A significant amount of publications apply only slightly differing variations of the performance loss ratio model, as proposed within this paper [39, 44]. Apart from that, some authors propose more complex models, that are also discussed here [20, 33]. In line with prevailing literature, we follow the performance loss ratio model.

3.2 Noisy neighbour detection

Regarding the detection of misbehaving workloads, different approaches utilising monitoring have been published. This is either supported by Machine Learning approaches [7] or by analysing raw metric thresholds. Regarding the latter approach most publications focus on a single metric [31, 9, 45]. The most prominent metric is the steal value [24] that allows detection of CPU over-provisioning. The actual detection of noisy neighbour behaviour is only slightly related to this paper.

Means to mitigate the effects caused by said resource competition is manifold. A simple approach is a very static resource allocation that is not changed later on. This tries to minimise possible contentions and overbooking in general [47]. To achieve that, some approaches make use of forecasting techniques [6]. Furthermore, some suggest algorithms that dynamically migrate VMs across the cloud infrastructure based on performance metrics [16, 40] or special metric like the mentioned CPU steal [24]. Masdari et al. [27] give a thorough overview of further and more in depth strategies within this area.

3.3 Workflow automation

Workflow automation becomes relevant, as soon as there are so many measurements performed, that they are hardly manageable by a single human. Automation reduces the risk of error and improves reproducibility as stated throughout this paper.

This fact is also considered by other works, including [34]. Here they propose a complex model of an experiment as code within the cloud. This contains everything from initial Virtual Machine (VM) provisioning over measurement enactment towards data gathering. Similarly, Seybold [35] presented a complex workflow in his dissertation about database assessment on multiple levels. Compared to those works, the approach applied in this paper does not evaluate on cloud infrastructure, but rather on bare-metal systems. The base paradigms however, are adapted to make the handling of physical servers similar to the handling of VMs.

If isolation measurements are pursued for more resources and more technologies, a final conclusion regarding their respective isolation capabilities can be drawn. Those could be put on an n-dimensional matrix, with each dimension representing a resource. Moreover, virtualisation technology vendors could use this methodology in order to distinguish themselves from competing vendors. Seybold pursued that for database applications. Based on his insights he published a ranked list for database applications and offerings [4].

4 SYSTEM MODEL AND REQUIREMENTS

This section sets the frame for further discussions and design decisions. In particular, section 4.1 defines a coarse-grained system model based literature in order clarify which hardware resources need to be considered for measuring isolation capabilities. Moreover, section 4.2 introduces five high-level requirements aiming at a flexible and open evaluation framework supporting scientific standards.

4.1 System Model

Modern computing systems feature a multitude of different resources. In order to select representative coarse-grained ones, the creation of a system model is necessary.

A large body of research work exists that assess performance and other characteristics of virtualisation technologies for one or multiple hardware resources such as CPU and disk. In the following, we use this body of work to identify which typical resources are investigated by other authors.

Wang et al. [44] review performance and isolation characteristics of modern Container Runtime Interface (CRI) compatible container runtimes. Hereby they focus on the system resources CPU, memory, disk and network regarding performance. Moreover, CPU is considered for its isolation capabilities. Similarly, Soltesz et al. [37] analyse hypervisors and propose container-based virtualisation as an alternative. They therefore discuss CPU, memory, disk and network performance and briefly discuss CPU isolation. Other than that, Carver [8] investigate the consolidation capabilities of containers and virtual machines. They focus on CPU, memory and disk. Sharma et al. [36] perform a comparative study regarding virtualisation technology overhead, also focusing on CPU, memory, disk and network.

Outside strictly academic peer reviewed publications Gregg [13] extensively discusses distinct system resources and how to assess them regarding their performance. They focus on CPU, memory, disk, network and file systems.

While there are many more publications that address this topic, this brief review shows a discernible direction: resources considered include at least CPU, memory, disk, and network. Accordingly, these should be supported by our work.

4.2 Requirements

In order to meet scientific standards, setting requirements towards the implemented method need to be made. The following will briefly gather and describe them.

Isolation Measurement. This central requirement describes the fact that the experiments need to be structured in a way, that allows for the acquisition of isolation capabilities. Assuming the performance loss ratio model in eq. (3), several measurements need to be taken individually. This includes the measurement of a base performance, as well as capturing performance under certain resource contention scenarios.

Load Generation. In order to generate meaningful data through measurements, specific load has to be generated. The configuration needs to be fine-grained to specifically target the resources to be analysed. This does not necessarily mean that a single tool is used for every resource to be analysed, the chosen tool however does need to satisfy this requirement. Moreover, it needs to be independent of the virtualisation technology used.

Data Acquisition. Next to the generation of load, there need to be means in place to gather data. Data in this case are the performance characteristics, or more specifically the utilization degree of resources. The acquisition needs to be independent of both, the applied virtualisation technology and the load generation tool. Moreover, the resource demand of the actual measurements must not have significant impact onto the resources under test in order to avoid measurement distortion.

Reproducibility. An experiment that cannot be reproduced with the same or similar results, bears only low value. Experiments need to be reproducible on a given system. Moreover, it should also be reproducible on reasonable similar systems. This is tightly coupled with concepts of resource provisioning.

Automatability. The amount of scenarios to run in order to compare isolation capabilities of two or more technologies can quickly become hard to manage. Therefore, the applied method needs to be automatable. Ideally, every part of the lifecycle of a measurement should be automized starting from resource provisioning over load generation towards data acquisition.

5 EVALUATION FRAMEWORK

This section introduces the design and implementation aspects of our evaluation framework. It takes the considerations and requirements of section 4 and elaborates the selection as well as the implementation of appropriate methods, processes, and technologies.

The structure of the section follows the requirements. Hence, in the following, section 5.1 initially present how the requirement of isolation measurement has been addressed. It is the core functional requirement and addressing it defines the overall measurement process. Subsequently, section 5.2 and section 5.3 address the remaining functional requirements of load generation and data acquisition which constitute further elementary building blocks for meaningful experiments.

section 5.4 then discusses the realisation of reproducibility by establishing immutability and Infrastructure-as-Code as well as Experiment-as-Code principles. The concluding section 5.5 addresses automation, which is the enabler for being able to run a large number of experiments for statistical significance.

5.1 Isolation Measurement

The whole process of a single measurement begins with the *(i)* spawning of a virtualisation technology process. Within this *(ii)* stress is induced by the respective load generation tools. Shortly after, the *(iii)* profiling process on the host system is started in parallel. This profiling supervises and profiles the virtualisation technology process. Upon success, data is *(iv)* acquired and *(v)* stored on external storage. Figure 3 visually represents this flow.

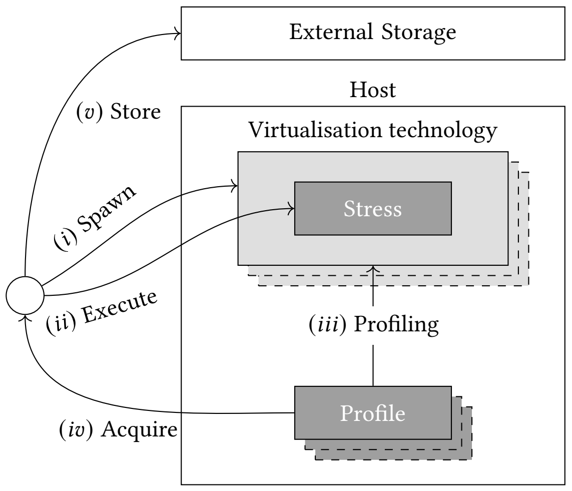


Figure 3: Flow of an abstract measurement

In order to determine the isolation characteristics for a specific resource of a certain virtualisation technology, measurements for the abiding and disruptive tenant need to be made. Consequently, the measurements need to be performed in parallel. Further, the terminology regarding experiments and scenarios will be defined as follows:

Definition 5.1 (Scenario and experiment). A single measurement run involving one or more tenants is called a scenario. All combined scenarios together form an experiment.

Scenarios exist by the combination of behaviour of tenants. Within this paper, four distinct behaviours are considered. The first one is the *(i)* undercommitted state. This state means that the workload itself is below 100% utilisation and thus does not reach saturation. The *(ii)* saturated state however describes exactly that. The resource fully utilises its granted resource and saturation might happen. The workload does however try to not exceed saturation. Whereas the *(iii)* overcommitted scenarios tries to do so. It tries to utilise more resources than there are actually available. Lastly

id	shortname	tenant *a*	tenant *b*
1	a_u	undercommited	
2	a_s	saturated	
3	a_o	overcommitted	
4	a_f	unrestricted	
5	a_ub_u	undercommitted	undercommitted
6	a_ub_s	undercommitted	saturated
7	a_ub_o	undercommitted	overcommitted
8	a_ub_f	undercommitted	unrestricted
9	a_sb_u	saturated	undercommitted
10	a_sb_s	saturated	saturated
11	a_sb_o	saturated	overcommitted
12	a_sb_f	saturated	unrestricted
13	a_ob_u	overcommitted	undercommitted
14	a_ob_s	overcommitted	saturated
15	a_ob_o	overcommitted	overcommitted
16	a_ob_f	overcommitted	unrestricted

Table 1: Experiment scenarios

there is the *(vi)* unrestricted state where there are no set boundaries for a workload, and it simply tries to utilise as many resources howsoever possible.

To systematically create all scenarios, a combination of all four states of the two tenants is performed. The workload of tenant *a* is fixed, while the workload of tenant *b* is changed. Tenant *a* cycles through the states *(i)* undercommitted, *(ii)* saturated and *(iii)* overcommitted, whereas tenant *b* iterates over *(i)*, undercommitted, *(ii)* saturated, *(iii)* overcommitted and *(vi)* unrestricted. Additionally, before any combination of tenants and their workloads happen, a baseline test is performed to measure the resource without any contention happening. This is mandatory, to prove the utilisation capabilities of a single resource. Table 1 shows all run scenarios of an experiment in a table.

5.2 Load Generation

This section discusses load generation for several distinct resources as presented in section 4.1. The tools themselves are selected according to the requirements set in section 4.2.

CPU. Stressing the CPU is done via a tool called `stress-ng`[3]. This micro benchmark allows generating load for a multitude of different system resources. Most importantly it can generate specific arbitrary load for Central Processing Units (CPUs). In its minimal form, it simply stresses `n` cores with a set of different methods over a fixed period of time. Experiments have shown that this minimal operation mode is sufficient for the load tests necessary to determine the isolation capabilities.

Memory. Analogous to the CPU load generation, memory load is generated similarly. Again, the `stress-ng` micro benchmark tool is applied. However, the configuration in this case, is a little more elaborated. `stress-ng` offers the possibility to directly target the

[3]https://github.com/ColinIanKing/stress-ng

virtual memory with n workers. Those workers continuously call the system calls mmap and munmap to allocate and deallocate memory within the allocated virtual memory. Since the idea of this stress test is to allocate as much memory as possible the munmap part is disabled by the -vm-keep flag.

Disk. Within the research and operations community, the micro benchmark Flexible I/O (FIO)[4] is rather popular. FIO allows a multitude of options to test anything file system or storage block device related. Especially the latter is of interest here. As FIO allows setting a read/write ratio, separate stress tests for each one can be performed, as well as arbitrary combinations of both. In order to circumvent any possible implemented caches or unwanted optimisations by using system memory, FIO can directly write to a disk without any redirection. In contrast to memory load generation, a single run is not supposed to fill the storage capacity of a disk, but rather utilise it to a certain degree with a set amount of Input Output Operations Per Second (IOPS). Therefore, files or even a file system existing on the target disk is of no interest.

Network. Many tools are available within the domain of network load generation. Some of those are very use-case specific and scoped more narrow, like the measurement of latency between links. Others focus on Wi-Fi signal strength or jitter on a link. Similar to the arguments for disk, utilisation is not as straight forward as it is for CPU or memory. Here, throughput is the resource to be utilised. Network throughput benchmarking as micro benchmark can be implemented with iperf3[5]. In order to generate load, iperf3 needs a client and a server. For this scenario, the server best runs on a remote machine, but directly connected with as little latency as possible. Depending on the configuration, the client either connects to the server via TCP or UDP and tries to achieve as much throughput as possible or a defined limit. Network interface cards with a maximum possible throughput of more than 40 Gbit/s have shown not to be easily utilisable with a single iperf3 process and thus needs to be parallelized if desired [42]. This is not an option of iperf3 itself and needs to be done outside its configuration, for example by running multiple instances on their own ports.

5.3 Data Acquisition

This section discusses profiling for all resources of the system model (cf. section 4.1). The tools themselves are selected according to the requirements. All the discussed benchmarking and load generation tools previously mentioned, implement an output during or after their execution. While their outputs would be theoretically usable, they fall short of most of the requirements described. As they are not decoupled from load generation, they are not able to perform black-box measurements.

General purpose system monitoring tools on the other hand offer more flexibility. They certainly are decoupled, but do not offer high resolution. Tools in question include widespread Linux typical tools such as vmstat[6], sar[7] and top[8] but also monitoring stacks like Prometheus[9] or Influx[10], which are often backed by time-series databases. Yet, the performance impact of the latter is questionable. Especially considering the fact that they usually collect way more metrics than desired.

Due to those considerations, system profiling (cf. section 2.3) becomes the natural choice. Profiling is technology independent, as it merely targets processes and not necessarily a specific implementation. Moreover, it is decoupled from load generation since it usually does not generate load itself. It is able to measure from outside the virtual workload. The frequency for sampling is completely arbitrary and only limited by local storage and the involved CPU. Of course the frequency needs to be sensibly set in order to keep it lightweight.

In order to fully leverage every modern Linux instrumentation we make use of the Extended Berkeley Packet Filter (eBPF) system. The origins of eBPF lie within the initial implementation Berkeley Packet Filter (BPF), done by McCanne et al. [29]. The idea behind this technology, was the execution of a filter expression which is passed to the kernel in order to be interpreted there. This enabled the user to filter packets without transporting every packet from the kernel to user space and back. The initial BPF was implemented as a minimal and very limited VM residing inside the kernel.

eBPF follows up on that concept by improving the capabilities of that VM significantly. Most notably, the amount of event targets increased to allow the collection and manipulation for sources like PMCs, tracepoints, kernel and user functions. Essentially, all the instrumentation points mentioned previously are directly available to the eBPF VM. This is visualised in fig. 4 as light grey boxes connected to the BPF block.

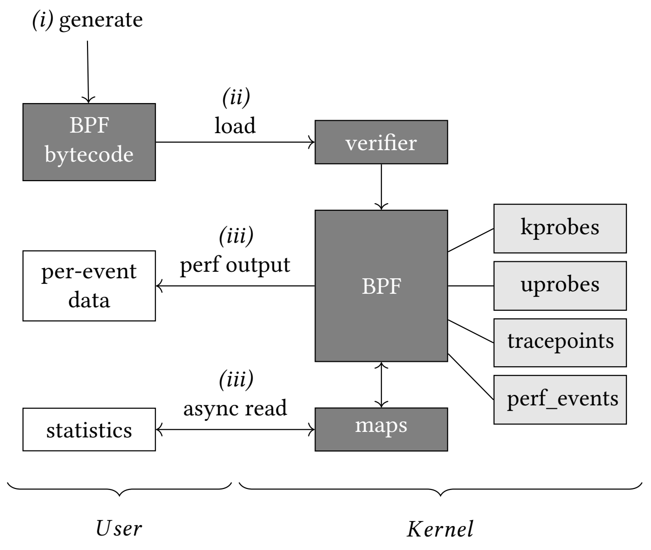


Figure 4: eBPF internals

Compared to traditional approaches, eBPF offers a lot of benefits to the user. One benefit is the availability of virtually any instrumentation point, without the need to load a specific kernel module

[4]https://github.com/axboe/fio
[5]https://github.com/esnet/iperf
[6]https://man7.org/linux/man-pages/man8/vmstat.8.html
[7]https://man7.org/linux/man-pages/man1/sar.1.html
[8]https://man7.org/linux/man-pages/man1/top.1.html
[9]https://prometheus.io/
[10]https://www.influxdata.com/

or recompile the kernel. This fact makes the usage of those instrumentations very stable and usable in production, since the Linux development project implements high quality standards.

Another benefit is the possibly low overhead compared to alternative approaches [17, 11]. The actual specific overhead cannot be easily determined since the overhead of metrics collection is a function of the type, number, and instrumentation for the collected metrics [23]. An aspect that potentially decreases overhead though, is the fact, that data rarely needs to be transferred from kernel to user space. The BPF program itself is capable of many data analytics operations and inline inspection of data points. In fact, a lot of eBPF based tools only report their metrics or findings upon program exit. Of course, this is an arbitrary limitation. Results can be transferred to user space at any time the user may do so. This output and two mentioned techniques for gathering data are annotated as *(iii)* perf_output and *(iii)* async read in fig. 4

There are a lot of possibilities to generate BPF byte-code. This is usually the first step as highlighted in fig. 4 as *(i)* generate. Upon generation, it is *(ii)* loaded into the kernel for a verifying step before being passed to the BPF VM.

Bpftrace[11] is a reasonable compromise between simple command line tools like perf[12] and the more complex compiler collection bcc[13]. It is therefore the tool of choice for the measurement methodology presented here. The tool itself is built on top of BCC and acts as a high-level programming language inspired by the `awk`[14] language. While not being as powerful as bcc, it is still capable of tracing most scenarios with adequate necessary effort. For this reason, the tool is chosen to interact with Linux events and eBPF.

5.4 Reproducibility

There are many strategies to achieve reproducibility leveraging different methodologies, tools and concepts. In order to be able to ignore any pre-existing infrastructure and configuration we decide to pursue an immutable approach. This approach does not assume any provisioned system, but rather provisions anything on demand ranging from the physical server OSs to the actual experiment. This aligns to the findings of Traeger et al. [41], who suggest rebooting a system after an experiment to minimise any traces from a previous experiment.

Reaching reproducibility by immutability, as proposed, is separated into two steps based on the involved layers. These namely are the *(i)* physical system provisioning and the *(ii)* actual experiment provisioning.

For *(i)* Fedora CoreOS[15] synergises well with the immutability paradigm. It is an immutable OS that can be provisioned directly onto bare-metal. It therefore needs an existing network boot stack. The OS itself is configured with a configuration file called "ignition". Packages can not be installed and have to be provided via containers. In order to easily re-provision the server, the OS itself is not installed into the system, but rather booted into memory. If applicable, specified parts can be persisted on disk if configured accordingly.

For *(ii)*, a reasonable choice for common container engines is anything involving `runc`. This technology is engine agnostic as long as it implements the CRI specification. These containers can be configured via environment variables and dynamic mounts. They thus satisfy everything a non containerised application would also satisfy.

Every immutable asset, being an artefact or a configuration, is versioned in a repository and thus exists as code. These range from OS configuration over runtime containers towards experiment configuration. Hence, they exist as "Infrastructure as Code (IaC)"[2] and "Configuration as Code (CaC)"[38]. Together they form "Experiments as Code (ExaC)"[1].

5.5 Workflow Automation

The missing link between provisioning the immutable runtimes, as well as the immutable experiments, is their actual enactment. Experiments involve the creation of various resources on multiple servers sequentially and in parallel. They have to be timed, and results need to be gathered. In order to handle so many experiments and to reduce the risk of human error, this has to be automated.

While simple approaches like procedural shell scripts might lead to fast results, they are merely sufficient as a prototype. The task flow necessary for the experiments within this work, outgrows the possibilities of such a simple approach. Circumventing those limitations would make them unreasonable complex. A better fitting approach would be the introduction of a workflow engine which handles these kinds of orchestrations a lot better. It brings the benefit of not only describing an experiment in a reproducible way, but also describing the whole process/workflow that is involved. It starts from configuring the runtimes, handles experiment execution and ends in results gathering.

With respect to the immutability concept and ExaC mentioned in section 5.4, workflows as code form representations of those. They include both, the runtime environment and the experiment configuration.

The workflow engine used here is Argo Workflow[16], which relies on Kubernetes[17] for container orchestration. This synergises well with the conclusion of section 5.4, that introduced the usage of immutable containers to improve reproducibility. Argo Workflow extends the capabilities of Kubernetes to enable it to schedule complex scenarios. This foremost enables the modelling of tasks as a directed acyclic graph (DAG). Tasks can be run in parallel and sequential, they can wait for each other or skip tasks if necessary. Also, concepts like scatter and gather are possible. Moreover, Argo Workflow offers the possibility to gather workflow (i.e. experiments) results on a remote storage.

6 VALIDATION

In order to validate the proposed measurement methodology described in section section 4 and designed in section 5, this section

[11]https://github.com/iovisor/bpftrace
[12]https://man7.org/linux/man-pages/man1/perf.1.html
[13]https://github.com/iovisor/bcc
[14]https://man7.org/linux/man-pages/man1/awk.1p.html
[15]https://getfedora.org/en/coreos
[16]https://argoproj.github.io/workflows/
[17]https://kubernetes.io/

implements it for an example. The chosen technology here is "Podman"[18]. Podman is a container-based virtualisation technology.

After a brief description of the experimental setup, selected measurement results are presented and discussed.

6.1 Experimental Set-up

In total, five servers are used. Each of those servers are symmetrically configured. They are equipped with two Intel CPU of the model "Intel(R) Xeon(R) CPU E5-2630 v3" with a basic clock frequency of 2.40 GHz and maximum clock frequency of 3.20 GHz. Those CPUs had a total of $16 \cdot 16 = 256$ GiB DDR4 memory clocked at 2133 MHz available. The disk involved at the IOPS isolation tests is a Samsung SM843TN, rated at 15000 IOPS "random write" performance. Involved server types are two experiment nodes, one boot-stack node that provides bare metal provisioning and one experiment head node running Argo Workflow.

Networking between all involved nodes is implemented by Mellanox Technologies network interface cards of the "MT27800 ConnectX-5" family. These are capable of a network throughput rate of 50 Gbit/s and are connected to a Mellanox Technologies "SN2100". This switch is capable of switching 16 100 Gbit/s clients at full rate. Every node mentioned before is connected to such a 100 Gbit/s port using a directly attached copper breakout cable that divides the 100 Gbit by two. This leads to two 50 Gbit QSFP28 connectors for the clients and a single 100 Gbit QSFP28 connector for the switch.

6.2 Selected Measurements

CPU. Table 2 analyses the performance impact tenant b can have on a. It therefore compares the baseline performance of a with the performance a' according to eq. (3) and eq. (4). Generally speaking, no induced workload by tenant b is able to significantly disturb tenant a. The highest degradation observable happens for scenario 11 where a is saturated and b tries to overcommit. Here a is reduced by 6.07% or 0.76% if considering the total possible CPU utilisation. The respective scenarios are highlighted in fig. 5a.

CPU isolation for Podman works very well and as expected. Only scenarios that cannot be properly scheduled due to overcommitment are not intuitive. While this certainly depends on the configuration of the Linux scheduler, this behaviour needs further investigation. In conclusion, this experiment shows, that CPU utilisation for this technology is effective.

Network. For every scenario presented here, no network isolation is imposed. This is due to the fact, this it is not available for the chosen technology. While container-based virtualisation offers network isolation in terms of abstraction it does not impose limits on its utilisation. This can be clearly seen from the undercommitted. Slightly before the Network Interface Card (NIC) becomes fully saturated, degradation starts to happen. This is visible from the last scenario $a_u b_o$ where tenant a almost degrades to zero.

This gets even worse for the saturation scenarios as seen in fig. 5b. Significant performance impact on tenant a can be seen even if both tenants are given the exact same resources. Here seemingly, the older processes get a slight performance benefit, even though this behaviour is left for further investigation. As soon as one tenant utilises more clients than another, this tenant will win on the resource contention. This is most visible on the last scenario $a_s b_o$ where tenant a with 5 clients competes against tenant b with 32 ones.

Table 2: CPU isolation for every scenario

id	shortname	a cpu	a' cpu	I_{ulr}	I_{plr}
1	a_u	12.47			
2	a_s	25.00			
3	a_o	25.00			
4	a_f	98.92			
5	$a_u b_u$	12.47	12.48	0.00	0.06
6	$a_u b_s$	12.47	12.36	0.05	0.88
7	$a_u b_o$	12.47	12.35	0.06	1.03
8	$a_u b_f$	12.47	12.20	0.14	2.23
9	$a_s b_u$	25.00	24.91	0.05	0.37
10	$a_s b_s$	25.00	24.06	0.47	3.74
11	$a_s b_o$	25.00	23.48	0.76	6.07
12	$a_s b_f$	25.00	24.71	0.14	1.15
13	$a_o b_u$	25.00	25.00	0.00	0.01
14	$a_o b_s$	25.00	25.13	0.06	0.50
15	$a_o b_o$	25.00	25.08	0.04	0.30
16	$a_o b_f$	25.00	25.09	0.04	0.33

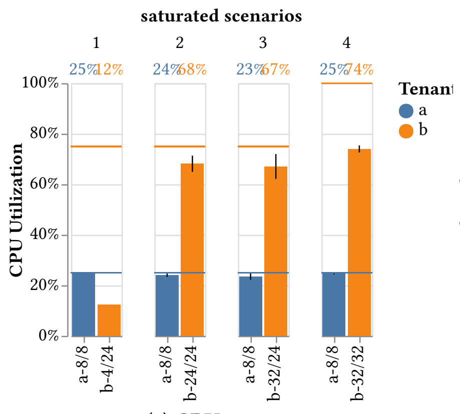
Table 3: Network isolation for every scenario

id	shortname	a network	a' network	I_{ulr}	I_{plr}
1	a_u	10.00			
2	a_s	50.12			
3	a_o	88.71			
4	a_f	93.29			
5	$a_u b_u$	10.00	10.05	0.05	0.50
6	$a_u b_s$	10.00	10.00	0.00	0.01
7	$a_u b_o$	10.00	8.29	1.71	17.08
8	$a_u b_f$	10.00	2.37	7.63	76.32
9	$a_s b_u$	50.12	49.99	0.13	0.25
10	$a_s b_s$	50.12	46.54	3.58	7.14
11	$a_s b_o$	50.12	28.38	21.74	43.37
12	$a_s b_f$	50.12	12.89	37.23	74.27
13	$a_o b_u$	88.71	79.20	9.52	10.73
14	$a_o b_s$	88.71	61.13	27.59	31.10
15	$a_o b_o$	88.71	51.99	36.72	41.39
16	$a_o b_f$	88.71	19.02	69.69	78.56

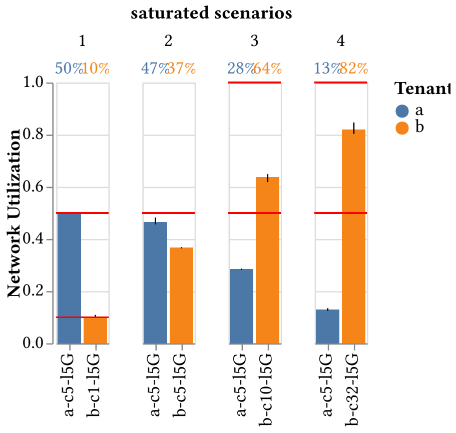
Table 3 analyses the performance impact tenant b can have on a. It therefore compares the baseline performance of a with the performance a' according to eq. (3) and eq. (4). Network isolation for Podman does not work at all, hence this was expected since there is no specific resource isolation in place. In conclusion, this experiment shows, that network isolation for this technology is not effective and needs an appropriate integration.

Memory. Finally memory isolation is analysed. These scenarios slightly differ from the previously presented. Memory, or more

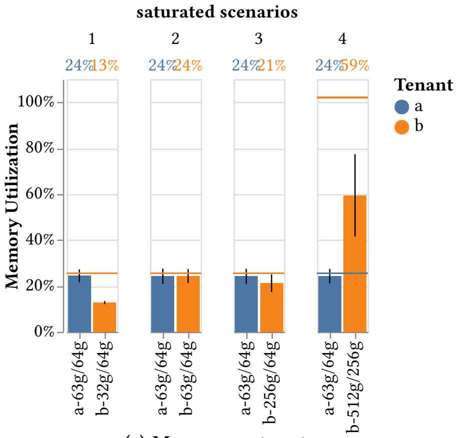
[18]https://podman.io/



(a) CPU saturate



(b) Network saturate



(c) Memory saturate

Table 4: Memory isolation for every scenario

id	shortname	a memory	a' memory	I_{ulr}	I_{plr}
1	a_u	12.80			
2	a_s	24.14			
3	a_o	21.24			
4	a_f	71.94			
5	$a_u b_u$	12.80	12.74	0.06	0.48
6	$a_u b_s$	12.80	12.68	0.12	0.91
7	$a_u b_o$	12.80	12.74	0.06	0.44
8	$a_u b_f$	12.80	12.64	0.16	1.26
9	$a_s b_u$	24.14	24.39	0.25	1.02
10	$a_s b_s$	24.14	24.18	0.04	0.16
11	$a_s b_o$	24.14	24.20	0.06	0.27
12	$a_s b_f$	24.14	24.24	0.10	0.40
13	$a_o b_u$	21.24	21.29	0.05	0.25
14	$a_o b_s$	21.24	21.21	0.03	0.13
15	$a_o b_o$	21.24	21.72	0.49	2.30
16	$a_o b_f$	21.24	21.14	0.10	0.46

specifically the Resident Set Size (RSS), is not allocated instantly. Moreover, if too many workers try to allocate a lot of memory as fast as possible, the Linux scheduler will Out Of Memory (OOM) terminate them at a high frequency possibly leading to an oscillating system behaviour. This can be seen for the last scenario in fig. 5c where the standard deviation of b is comparatively high. Most importantly though, memory isolation is almost ideal. The abiding tenant a is never disturbed by the b, even when trying to allocation 100% of the system memory. This can also be clearly seen in Table 4.

7 DISCUSSION

While we successfully validated our methodology and framework with the example given in 6 there are open questions that require a critical reflection. Most importantly is the adaptability of our framework for other classes of virtualisation. Further, we highlight possible threads to validity and briefly discuss them in the following.

7.1 Adaptability

The adaptability of the methodology proposed by us is a core consideration. Specifically the discussion of load generation and data acquisition in both section 4 and section 5 reflect that. These two aspects are decoupled from the actual virtualisation technology.

Moreover and as mentioned previously, data acquisition needs to be performed outside the virtualisation technology. This in turn creates the demand for adequate instrumentation points. Since we concentrate on the evaluation of Linux enabled virtualisation technologies we can rely on the availability of Kernel instrumentation. While there are technology specific ones available, we deliberately decide on general purpose ones to make sure adaptability is ensured.

The Linux kernel provides a vast amount of instrumentation that recently improved even further with the possibilities of eBPF. Since we heavily utilise this technology, we are very confident to be able to profile any existing and upcoming Linux based virtualisation technology.

7.2 Threats to validity

The eBPF based profiling approach is one of essential strengths of this framework, but also possible pitfall. Not every kernel instrumentation point can be used and assumed to be stable. They might change across major releases, but some dynamic ones might even change upon minor ones. In consequence, the framework needs to be changed as the kernel itself changes. Of course, this depends on the actual profiling implementation.

Further, and as mentioned throughout this paper, this framework completely depends on the Linux kernel. While the methodology can be conceptually applied to any operation system, the specific implementations cannot be easily adapted. However, there are efforts regarding the execution of eBPF code on Windows[19] in progress [10].

The load generation phase leverages micro benchmarks to generate different levels of load. While it is an important initial step to examine distinct resources on their own, possible side effects are completely neglected. More sophisticated and complex benchmarks could yield different results and might reveal some dependencies that remain undetected.

[19]https://www.microsoft.com/windows

8 CONCLUSION

The isolation measurement methodology, as well as the design considerations proposed here, can help to file decisions regarding a virtualisation technology from a practical perspective. The results can be used to compare two or more virtualisation technologies regarding specific isolation requirements. These naturally depend on the requirements the use case imposes. The isolation quantification model applied here sufficiently enables the comparison of those technologies. What it does not consider though, are other implications that do not directly relate to performance isolation. This includes virtualisation overhead in terms of resources, security isolation or licensing fees.

To conclude, this paper has developed a generally applicable methodology and implemented a framework to determine isolation characteristics of virtualisation technologies. It is highly configurable and extendable in terms of applied technologies and underpinned system model. Its flexibility further allows to craft any arbitrary scenarios within any kind of experiment.

ACKNOWLEDGMENTS

This work has received funding by the Ministerium für Wirtschaft, Wissenschaft und Digitalisierung under grant number: 03EFPBW217 BaaS. It further received funding by the Ministerium für Wissenschaft, Forschung und Kunst Baden-Württemberg under the Project bwNET2020+.

REFERENCES

[1] Leonel Aguilar et al. 2022. Experiments as Code: A Concept for Reproducible, Auditable, Debuggable, Reusable, &amp; Scalable Experiments. Version 1. DOI: 10.48550/ARXIV.2202.12050.

[2] Matej Artac, Tadej Borovssak, Elisabetta Di Nitto, Michele Guerriero, and Damian Andrew Tamburri. 2017. DevOps: Introducing Infrastructure-as-Code. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C). (May 2017), 497–498. DOI: 10.1109/ICSE-C.2017.162.

[3] M Ali Babar and Ben Ramsey. 2017. Understanding Container Isolation Mechanisms for Building Security-Sensitive Private Cloud. DOI: 10.13140/RG.2.2.3404 0.85769.

[4] benchAnt. 2022. Database Ranking - Performance & Costs (2022). Retrieved Sept. 12, 2022 from https://benchant.com/ranking/database-ranking.

[5] Eric W Biederman. 2006. Multiple Instances of the Global Linux Namespaces. In Proceedings of the Linux Symposium, 14.

[6] Norman Bobroff, Andrzej Kochut, and Kirk Beaty. 2007. Dynamic Placement of Virtual Machines for Managing SLA Violations. In *2007 10th IFIP/IEEE International Symposium on Integrated Network Management*. 2007 10th IFIP/IEEE International Symposium on Integrated Network Management. (May 2007), 119–128. DOI: 10/b98hqc.

[7] H. Bouattour, Y. B. Slimen, M. Mechteri, and H. Biallach. 2020. Root Cause Analysis of Noisy Neighbors in a Virtualized Infrastructure. In *2020 IEEE Wireless Communications and Networking Conference (WCNC)*. 2020 IEEE Wireless Communications and Networking Conference (WCNC). (May 2020), 1–6. DOI: 10.1109/WCNC45663.2020.9120635.

[8] Damien Carver. 2019. Advanced consolidation for dynamic containers, 119. DOI: 10.1109/NCA.2017.8171363.

[9] Giuliano Casale, Stephan Kraft, and Diwakar Krishnamurthy. 2011. A Model of Storage I/O Performance Interference in Virtualized Systems. In *2011 31st International Conference on Distributed Computing Systems Workshops*. 2011 31st International Conference on Distributed Computing Systems Workshops. (June 2011), 34–39. DOI: 10.1109/ICDCSW.2011.46.

[10] [SW], eBPF for Windows Oct. 19, 2022. Microsoft. URL: https://github.com/microsoft/ebpf-for-windowsRetrieved Oct. 19, 2022 from.

[11] Mohamad Gebai and Michel R. Dagenais. 2018. Survey and Analysis of Kernel and Userspace Tracers on Linux: Design, Implementation, and Overhead. *ACM Computing Surveys*, 51, 2, (Mar. 12, 2018), 26:1–26:33. DOI: 10.1145/3158644.

[12] Robert P. Goldberg. 1973. Architectural Principles for Virtual Computer Systems. HARVARD UNIV CAMBRIDGE MA DIV OF ENGINEERING AND APPLIED PHYSICS, (Feb. 1, 1973). Retrieved Feb. 8, 2022 from https://apps.dtic.mil/sti/citations/AD0772809.

[13] Brendan Gregg. 2020. *Systems Performance: Enterprise and the Cloud*. (Second ed.). *Addison-Wesley Professional Computing Series*. Addison-Wesley, Boston. ISBN: 978-0-13-682015-4.

[14] Tejun Heo, J Weiner, V Davydov, L Thorvalds, P Parav, T Klauser, S Hallyn, and K Khlebnikov. 2015. Control group v2. Retrieved Aug. 30, 2022 from https://www.kernel.org/doc/Documentation/admin-guide/cgroup-v2.rst.

[15] Jinho Hwang, Sai Zeng, Frederick y Wu, and Timothy Wood. 2013. A component-based performance comparison of four hypervisors. In *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*. 2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013). (May 2013), 269–276.

[16] C. Isci, J. Liu, B. Abali, J. O. Kephart, and J. Kouloheris. 2011. Improving server utilization using fast virtual machine migration. *IBM Journal of Research and Development*, 55, 6, (Nov. 2011), 4:1–4:12. DOI: 10.1147/JRD.2011.2167775.

[17] Jim Keniston, Ananth Mavinakayanahalli, Vara Prasad, and Prasanna Panchamukhi. 2007. Ptrace, Utrace, Uprobes: Lightweight, Dynamic Tracing of User Apps. In *Proceedings of the 2007 Linux Symposium*.

[18] G. Khanna, K. Beaty, G. Kar, and A. Kochut. 2006. Application Performance Management in Virtualized Server Environments. In *2006 IEEE/IFIP Network Operations and Management Symposium NOMS 2006*. 2006 IEEE/IFIP Network Operations and Management Symposium NOMS 2006. (Apr. 2006), 373–381. DOI: 10.1109/NOMS.2006.1687567.

[19] Younggyun Koh, Rob Knauerhase, Paul Brett, Mic Bowman, Zhihua Wen, and Calton Pu. 2007. An Analysis of Performance Interference Effects in Virtual Environments. In *2007 IEEE International Symposium on Performance Analysis of Systems Software*. 2007 IEEE International Symposium on Performance Analysis of Systems Software. (Apr. 2007), 200–209. DOI: 10.1109/ISPASS.2007.363750.

[20] Samuel Kounev, Klaus-Dieter Lange, and Jóakim von Kistowski. 2020. *Systems Benchmarking: For Scientists and Engineers*. Springer International Publishing, Cham. DOI: 10.1007/978-3-030-41705-5.

[21] Rouven Krebs, Christof Momm, and Samuel Kounev. 2014. Metrics and techniques for quantifying performance isolation in cloud environments. *Science of Computer Programming*, 90, (Sept. 2014), 116–134. DOI: 10.1016/j.scico.2013.08.003.

[22] Young Choon Lee and Albert Y. Zomaya. 2012. Energy efficient utilization of resources in cloud computing systems. *The Journal of Supercomputing*, 60, 2, (May 1, 2012), 268–280. DOI: 10.1007/s11227-010-0421-3.

[23] Joshua Levin. 2020. ViperProbe: Using eBPF Metrics to Improve Microservice Observability.

[24] Wes Lloyd, Shrideep Pallickara, Olaf David, Mazdak Arabi, and Ken Rojas. 2017. Mitigating Resource Contention and Heterogeneity in Public Clouds for Scientific Modeling Services. In *2017 IEEE International Conference on Cloud Engineering (IC2E)*. 2017 IEEE International Conference on Cloud Engineering (IC2E). (Apr. 2017), 159–166. DOI: 10.1109/IC2E.2017.29.

[25] Clive Longbottom. 2017. *The Evolution of Cloud Computing: How to Plan for Change*. BCS Learning & Development Ltd, Swindon, UK. 181 pp. ISBN: 978-1-78017-358-0.

[26] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) than your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP '17: ACM SIGOPS 26th Symposium on Operating Systems Principles. ACM, Shanghai China, (Oct. 14, 2017), 218–233. ISBN: 978-1-4503-5085-3. DOI: 10.1145/3132747.3132763.

[27] Mohammad Masdari, Sayyid Shahab Nabavi, and Vafa Ahmadi. 2016. An overview of virtual machine placement schemes in cloud computing. *Journal of Network and Computer Applications*, 66, (May 1, 2016), 106–127. DOI: 10/ggchst.

[28] Jeanna Neefe Matthews, Wenjin Hu, Madhujith Hapuarachchi, Todd Deshane, Demetrios Dimatos, Gary Hamilton, Michael McCabe, and James Owens. 2007. Quantifying the performance isolation properties of virtualization systems. In *Proceedings of the 2007 Workshop on Experimental Computer Science - ExpCS '07*. The 2007 Workshop. ACM Press, San Diego, California, 6–es. ISBN: 978-1-59593-751-3. DOI: 10.1145/1281700.1281706.

[29] Steven McCanne and Van Jacobson. 1993. The BSD packet filter: a new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings* (USENIX'93). USENIX Association, USA, (Jan. 25, 1993), 2.

[30] Nicolas Poggi. 2019. Microbenchmark. In *Encyclopedia of Big Data Technologies*. Sherif Sakr and Albert Y. Zomaya, (Eds.) Springer International Publishing, Cham, 1143–1152. ISBN: 978-3-319-77525-8. DOI: 10.1007/978-3-319-77525-8_111.

[31] Xing Pu, Ling Liu, Yiduo Mei, Sankaran Sivathanu, Younggyun Koh, Calton Pu, and Yuanda Cao. 2013. Who Is Your Neighbor: Net I/O Performance Interference in Virtualized Clouds. *IEEE Transactions on Services Computing*, 6, 3, (July 2013), 314–329. DOI: 10.1109/TSC.2012.2.

[32] Fayruz Rahma, Teguh Bharata Adji, and Widyawan Widyawan. 2013. Scalability Analysis of KVM-Based Private Cloud For Iaas. *International Journal of Cloud Computing and Services Science (IJ-CLOSER)*, 2, 4, (Oct. 13, 2013), 288–295, 4, (Oct. 13, 2013). Retrieved May 3, 2021 from http://www.iaesjournal.com/online/index.php/IJ-CLOSER/article/view/4535.

[33] Rouven Krebs. 2015. *Performance Isolation in Multi-Tenant Applications*. Retrieved Aug. 7, 2022 from https://se.informatik.uni-wuerzburg.de/fileadmin/10030200/user_upload/dissKIT_BW.PDF.

[34] Joel Scheuner, Philipp Leitner, Jürgen Cito, and Harald Gall. 2014. Cloud Work Bench – Infrastructure-as-Code Based Cloud Benchmarking. In *2014 IEEE 6th International Conference on Cloud Computing Technology and Science*. 2014 IEEE 6th International Conference on Cloud Computing Technology and Science. (Dec. 2014), 246–253. DOI: 10.1109/CloudCom.2014.98.

[35] Daniel Seybold. 2021. *An Automation-Based Approach for Reproducible Evaluations of Distributed DBMS on Elastic Infrastructures*. Ph.D. Dissertation. Universität Ulm, (May 14, 2021). ISBN: 9781757899956. DOI: 10.18725/OPARU-37368.

[36] Prateek Sharma, Lucas Chaufournier, Prashant Shenoy, and Y. C. Tay. 2016. Containers and Virtual Machines at Scale: A Comparative Study. In *Proceedings of the 17th International Middleware Conference*. Middleware '16: 17th International Middleware Conference. ACM, Trento Italy, (Nov. 28, 2016), 1–13. ISBN: 978-1-4503-4300-8. DOI: 10.1145/2988336.2988337.

[37] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. 2007. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. *ACM SIGOPS Operating Systems Review*, 41, 3, (Mar. 21, 2007), 275–287. DOI: 10/cr62t6.

[38] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. 2015. Holistic configuration management at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP '15: ACM SIGOPS 25th Symposium on Operating Systems Principles. ACM, Monterey California, (Oct. 4, 2015), 328–343. ISBN: 978-1-4503-3834-9. DOI: 10.1145/2815400.2815401.

[39] Xuehai Tang, Zhang Zhang, Min Wang, Yifang Wang, Qingqing Feng, and Jizhong Han. 2014. Performance Evaluation of Light-Weighted Virtualization for PaaS in Clouds. In *Algorithms and Architectures for Parallel Processing* (Lecture Notes in Computer Science). Xian-he Sun et al., (Eds.) Springer International Publishing, Cham, 415–428. ISBN: 978-3-319-11197-1. DOI: 10.1007/978-3-319-11197-1_32.

[40] Fei Tao, Chen Li, T. Warren Liao, and Yuanjun Laili. 2016. BGM-BLA: A New Algorithm for Dynamic Migration of Virtual Machines in Cloud Computing. *IEEE Transactions on Services Computing*, 9, 6, (Nov. 2016), 910–925. DOI: 10/f9gvtt.

[41] Avishay Traeger, Erez Zadok, Nikolai Joukov, and Charles P. Wright. 2008. A nine year study of file system and storage benchmarking. *ACM Transactions on Storage*, 4, 2, (May 11, 2008), 1–56. DOI: 10.1145/1367829.1367831.

[42] Simon Volpert, Georg Eisenhart, and Jörg Domaschka. 2022. Are kubernetes cni solutions ready for> 10 gbit/s?

[43] Zhiyuan Wan, David Lo, Xin Xia, and Liang Cai. 2019. Practical and Effective Sandboxing for Linux Containers, 41. DOI: 10.1007/s10664-019-09737-2.

[44] Xingyu Wang, Junzhao Du, and Hui Liu. 2022. Performance and isolation analysis of RunC, gVisor and Kata Containers runtimes. *Cluster Computing*, (Jan. 22, 2022). DOI: 10.1007/s10586-021-03517-8.

[45] Miguel G. Xavier, Israel C. De Oliveira, Fabio D. Rossi, Robson D. Dos Passos, Kassiano J. Matteussi, and Cesar A.F. De Rose. 2015. A Performance Isolation Analysis of Disk-Intensive Workloads on Container-Based Clouds. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. 2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. (Mar. 2015), 253–260. DOI: 10.1109/PDP.2015.67.

[46] Pingpeng Yuan, Chong Ding, Long Cheng, Shengli Li, Hai Jin, and Wenzhi Cao. 2010. VITS Test Suit: A Micro-benchmark for Evaluating Performance Isolation of Virtualization Systems. In *2010 IEEE 7th International Conference on E-Business Engineering*. 2010 IEEE 7th International Conference on E-Business Engineering. (Nov. 2010), 132–139. DOI: 10.1109/ICEBE.2010.71.

[47] Jiangtao Zhang, Zhixiang He, Hejiao Huang, Xuan Wang, Chonglin Gu, and Lingmin Zhang. 2014. SLA aware cost efficient virtual machines placement in cloud computing. In *2014 IEEE 33rd International Performance Computing and Communications Conference (IPCCC)*. 2014 IEEE 33rd International Performance Computing and Communications Conference (IPCCC). (Dec. 2014), 1–8. DOI: 10/gnkc94.