Optimal Sizing of VMs and Containers for various kernel-space loads

Tom Pawelek
University of Illinois,
Urbana-Champaign
tomaszp2@illinois.edu

Martín Becerra
University of Illinois,
Urbana-Champaign
carlosb3@illinois.edu

Hyunkyung Lee
University of Illinois,
Urbana-Champaign
hl73@illinois.edu

Keywords—Virtualization, Containers, Kubernetes, Optimization, Resource Management

I. Introduction

These days, we observe a massive shift from virtual machine based to container based deployment. This trend includes companies with an established VM presence, who are considering (or conducting) massive scale migrations of their infrastructure into containerized environments.

As their main principle, containers address the waste of resources on multiple iterations of the same kernel space. By sharing the kernel, we can allocate more memory and CPU time to userspace tasks. In theory, this should result in a higher efficiency, given the same specs of underlying hardware.

From system administrators' point of view, efficiency means that operating more applications with the same performance and limited capacity of hardware resources. From developers' point of view, efficiency with container based environments means less man-hours for development and distribution. Our focus will be on the former - infrastructural - gains.

In our experiments, we would deploy the same workload to virtual machines and containers respectively, and measure how much hardware resources are used and how good performance is. The applications will differ depending on their reliance on kernel-space resources.

Based on the results, we would find the optimal sizing of virtual machines and containers in comparison and determine whether container based environments are more efficient than virtual machine based environments.

II. Motivation

A. *Intellectual Merit*

Resource allocation and quota management are difficult problems to solve. Most teams struggle to get right - given way too many resources away or too little. Optimal sizing of VMs and containers and dynamically adjusting could help optimize overall cluster resources. So, it is important for large clusters with multiple teams and projects.

B. *Novelty*

Most literature and online narrative tries to argue general supremacy of one approach over the other. Our focus will be on kernel-space dependency and how it shapes the overall resource usage in VM and container based setups. We suspect that shared kernels might play a role in some application types.

C. *Impact*

When deciding between VM and container based application, there is no simple set of guidelines which would allow businesses to choose the most profitable solution. Usually, the influence of service providers guides companies into an environment mostly benefiting the former.

We are hoping to output a clear and concise narrative which will address this matter taking into consideration the kernel-space dependency of our business application.

Our combined experience in the IT space - ranging from hands-on to executive roles - will allow us to gauge commercial impact of proposed solutions. We're also hoping to evaluate the carbon footprint savings which can be achieved by our proposed optimization.

III. Implementation

We are planning multiple experiments. It will be further detailed as the research progresses. There will be two major categories of measurements:

A. *VMs and containers on a single host*

In a single host experiment, we would perform the bare measurements of similar distributed workload on a single server. The focus is to compare the shared kernel with the dedicated kernel. First, we would monitor CPU and memory resource usage of running VMs and configure the right size and counts of containers. Then, we would switch VMs to containers and measure performance and resource usage. Even if we allocate almost the same CPU and memory, the container will be lighter than the VM. However, there is also the capacity used by kubernetes, so we can compare how much difference it actually is. And the result will be different depending on workloads. Therefore, the experiment will proceed while deploying different kinds of workloads such as data intensive workloads.

B. *VMs and containers across multiple hosts*

In a multiple hosts experiment, we would measure the performance trade-off between live migration and restarting virtual machines and containers. Live migration is moving from a virtualization host to another. Restarting would be in the same host. The reason why people operate VMs and containers is to guarantee high availability. Therefore, live migration and restarting are very important functions in virtualization, especially in dynamically scalable environments. We would measure performance and resource usage while moving VMs and containers across multiple hosts, and we would calculate the trade-off.

In each category, we are going to simulate a different level of kernel-space dependency. Our goal is to minimize the influence of all other variables, such as external resources (storage, network).

To achieve this, we are going to use the following tasks as kernel-space heavy markers:

- GPU calculations
- loopback networking
- random number generation

As our research progresses, we are hoping to identify other kernel-intensive tasks to fine tune our measurements.

The ultimate goal is to establish a threshold for real-life applications which might trigger a choke point on the kernel space. We expect this might counter potential benefits of saved resources (mostly RAM) in overall efficiency analysis.

IV. Related works

In this section, we review the related work to clarify the problem to be addressed. We discuss virtualization and containers overhead, architectural patterns and their overhead, and simulating kernel space load.

A. *Virtualization & containers overhead*

Container-based virtualization is the latest technology to virtual machines and is quickly replacing them in the cloud environment, as in [1]. Containers are similar to virtual machines in the services they provide although containers don’t run a separate kernel and virtualize all hardware components theoretically. Reference [1] measured various parameters of containers and virtual machines. Throughput is the parameter of cpu performance and disk performance. Bandwidth is the parameter of memory performance and network performance. Latency is the parameter of application performance. Reference [1] concluded that the performance of containers and VMs were almost identical except the case of I/O bound applications. It was worth noting that these parameters were used for measurements of performance, but reference [1] did not disclose clear figures and implement any detailed experiments. We could only assume that containers and virtual machines were roughly similar and could not estimate optimal sizing.

The purpose of virtualization is efficiency, which is operating more applications with the same performance and limited capacity of hardware resources from system administrators' point of view. We can call it server consolidation, in which the maximum workload is allocated to the minimum number of physical servers. Virtualization technology drives server consolidation using virtual machines or containers [2].

The consolidation overhead is the extra workload that the system incurs for managing several VMs or containers. Reference [2] proposed a general method for quantifying and graphically representing the consolidation overhead from the perspective of the physical server. Overheads are calculated as the difference between the mean response times in the different virtualization scenarios of the combination of physical servers, virtual machines, and containers. Reference [3] found where the overheads came from in depth. There may be other reasons for overheads, but OS schedulers also make overheads because they ensure fair sharing of CPU time even for idle virtual machines and containers when multiple virtual machines and containers are running on the same physical server. Reference [2] and [3] are useful to measure the overhead of VMs and containers by CPU utilization. However, additional measurement of memory or disk is required to achieve optimal sizing.

Experiments in this paper will be carried out at the scale of single or multiple hosts, but we also referred to the papers about large-scale computing to find the measurements of compute resources. Reference [5] measured normalized core unit hours and normalized memory unit hours and CPU-to-memory ratio to measure machine utilization. It considered scheduling delays and terminations to operate stable clusters. Reference [6] presented a detailed analysis of warehouse scale computing including memory allocation and kernel. To achieve efficient resource management, reference [4], [5] and [6] took the top-down approach from clusters while we would take the bottom-up approach from VMs and containers.

B. *Architectural patterns and their overhead*

As part of our research we not only compared the difference between virtualization and non-virtualized environments, we also have taken in consideration the overhead produced by using different technologies and frameworks. Nowadays, the microservices architectural style is widely used and has become one of the standards in academia and industry for developing software. This pattern intends to compose of small independent services that communicate over well-defined APIs to encourage decomposition of monolithic applications into multiple independently deployed units.

In [11], Tuan et al. evaluates and compares different development frameworks for microservice-based applications used to improve developers’ productivity by designing reusable components, in particular Go Micro (Go), Moleculer (JavaScript with Node.js), Lagom (Scala), and Spring Boot/Spring Cloud (Java) - few of the most used frameworks. One of the key differences noted is in the use of JVM-based implementations, which have long latencies producing applications to start slowly. No major difference could be noted in end-to-end latency performance, since all four implementations show similar behavior, although to different extents. Finally, when discussing resource consumption, Go based applications showed the ability to create smaller Dockerized services compared to Lagom or Spring Boot and also needed produced cpu and ram usage. There is no one-size-fits all solution when choosing a microservice framework, but the results showed that Go applications' smaller images and footprint could be beneficial to be created and deployed quickly, leading to faster update cycles.

Comparing the performance of those three popular Web technologies, Node.js, PHP and Python-Web which are vastly used in the internet, showed the superiority of the asynchronous programming paradigm on Node.js [13, 14]. Node.js performs much better than the traditional technique PHP in high concurrency situations, no matter in benchmark tests or scenario tests. PHP handles small requests well, but struggles with large requests. At the same time, Node.js has performed better than the Python server. Node.js was better at handling multiple users submitting requests simultaneously, where the number of requests handled by Node.js were almost 250 times higher than those handled by Python.@

Ueda et al. used an open-source benchmark application for web services called Acme Air, to analyze the behavior of microservice and monolithic applications, for two widely used language runtimes, Node.js and Java Enterprise Edition (EE) [12]. They noted significant differences between the different architectures, where the microservices model can be 79.1% lower than the monolithic model on the same hardware configuration due to the application in a microservice model spends a longer time in a server runtime, such as a communication library, than that in a monolithic model for both application server frameworks. The throughput of the microservices experiment was always significantly lower than that of the monolithic model when the rest of the configuration options were the same. This suggests that we need to trade off between the benefit from agile development and the cost from performance overhead, both due to the microservice model. On the other hand, Java EE application server outperformed a Node.js application server on the 4 and 16 cores, except the 16-core bare-process configuration, while a Node.js application server outperformed a Java EE application server on a single core with the monolithic application.

Finally, to complement all of this information, Kumar et al. attempt to characterize three popular communication protocols - REST, gRPC, and Thrift, in terms of their network, memory, CPU utilization, and response time [18]. Trying different approaches to optimize the communication between client and server applications on the same host and varying the payload using these different protocols authors concluded that Thrift performs the best for communication between microservices due to rapid serialization and deserialization of the packets, followed by gRPC with its fast protocol buffers.

C. Simulating Kernel space load

The focus of our study is on Unix Kernels, with Linux based images forming the vast majority of commonly used containers hosted on Docker Hub (and therefore utilized in commercial products).

Unix Kernel was originally designed in the 1970 as a monolithic structure responsible for all low level requests with direct access to computer hardware. Since the early days, it supported the distinction of kernel vs user space. With the popularity of Open Source, the user space software quickly outpaced system calls in terms of volume and functionality. [15]

The purpose of our work is to determine how the optimal VM and container sizing changes depending on the kernel load itself. Many commercial guidelines exist focusing on virtualization itself. Separately, containerized environments are usually evaluated from the point of view of orchestration, assuming that virtualization happens independently from the business application (a layered approach). What we are trying to show is that there is no single best approach, as the shared kernel can become overloaded and undermine any performance gains from our user space scaling.

The significance of our approach is even greater if we consider numerous exploits that have recently plagued CPU architecture, such as Spectre and Meltdown. The patches for those two alone are responsible for more than doubled latency of the most common kernel operations [16]. This causes a lot of performance issues for established container clusters - both on prem and cloud. In those environments, both the underlying architecture as well as node's kernel version are always outside of the scope of software developers and maintainers. The prevalent "cloud mindset" puts the focus exclusively on the user space execution. Resulting performance decline can easily affect a delicate structure of microservices, where suddenly introduced latency breaks time-sensitive operations of a business application.

For commercial relevance, our experiments will focus on those system operations mostly affected by increased latency:

- read & write
- mmap
- poll
- select.

Our goal is to identify the most common use cases depending on these system calls and kernel functions. We will attempt to conduct experiments with their popular software implementations available on Docker Hub.

Similarly, we want to measure the performance hits across various microservice communication protocols:

- rest
- grpc
- thrift.

We will attempt to answer the question: is the fine tuning of payload size and network stack able to counter the kernel space losses. We're going to use Prajwal Kumar's paper as a reference point for our experiments [17].

Another popular trend observed in the last decade is a growing use of GPUs in computational tasks. In the second part of our study, we're going to build a vmware environment with compatible NVIDIA 3080 GPUs, as presented in [19].

In this setup, the kernel's role is limited to serving as a pass through between the user space and a PCI device. We will try to establish how badly the recently introduced patches affect our GPU setup in the most common computation scenarios (TensorFlow, MUMmerGPU).

Finally, we will try to manipulate the shared GPU memory as presented in [20] to see if we can counter the effects of extra kernel space overhead.

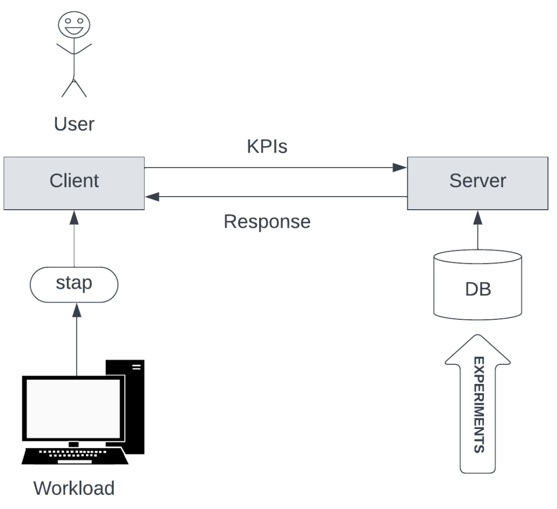
V. Solution Architecture

In order to provide network administrators with a usable deliverable, we have designed a tool called Load & Performance Evaluator (LPE).

It is a client-server model based solution, comprising of two parts:

- Client: this is an LPE script to be executed from within the environment being evaluated. It uses bash, python and stap to collect the KPIs. It also acts as a UI for the end-user.
- Server: the server contains a database of measurements and business logic which determines a final recommendation presented to the user. It is implemented in a continuous fashion: the more data is provided to the database, the better are its outcomes.

The overall architecture is presented below:



It is recommended to run LPE on a bare metal server, to ensure there is no virtualization overhead or limited kernel access which would interfere with KPI collection or mangle the data.

VI. KPI Models

The Key Performance Indicator (KPI) package collected by LPE client contains the following datasets:

- list of processes along with the proportion of time spent in User and Kernel space
- list of processes along with resources used
- version of installed packages
- IOPS stats per process
- network stats per process

While LPE does not collect any other information, such as a local file or database content, it is recommended to run it in a staging environment, without access to any sensitive information. Due to the nature of its low level access (profiling kernel requests) it has to be run as root.

The output of an LPE procedure is a workload classification. It could be categorized as either VM or Container. This is based on the assumption that modern CPUs will be used in production with virtualization support enabled in BIOS.

Our original plan was to have another possible category: bare metal. However, in our experiments we have found that the virtualization penalty does not depend on User/Kernel space utilization and is of negligible value. Even at large scale, when <1% performance gains come into play, we believe that the additional overhead cost of not being able to use virtualization would nullify any economical benefits.

It is worth noting that for the Container category, we assume container daemon running without virtualization. For hybrid VM/Container deployments, LPE will not be able to provide accurate recommendations.

Based on our experiments, we designed a classification model which is a combination of 2-dimensional continuous spectrum and a set of bias vectors.

The two dimensions are:

- proportion of time spent in User space to total time (from 0.0 when fully Kernel space to 1.0 when fully User space),
- memory usage.

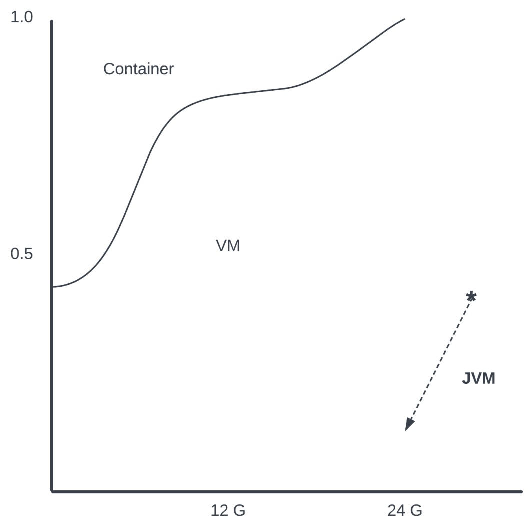
Both datasets are calculated by LPE for a dominant set of processes (based on resource allocation). The bias vectors represent:

- JVM presence,
- IOPS utilization,
- network utilization,
- per-package adjustments.

IOPS and network utilization does require bias because of imperfect implementation of passthrough drivers. In many environments, administrators keep the default virtualized network and storage controllers as default options for VMs.

Our experiments revealed serious Java bias towards virtualization. It looks like the JVM memory optimization does benefit from unrestricted kernel/OS access and constraining it to a container is counter-productive.

Finally, we kept the per-package adjustment option for future discoveries of products similar to Java, which might introduce bias to our generic model.



The JVM bias vector shown in the example model above is applied to any point in the KPI space, effectively promoting the VM solution over containers. For LPE to apply the bias, JVM needs to be a part of the dominant (i.e. highest load) processes. This is typically indicative of Java being used for the main business load rather than an auxiliary process.

VII. Experiments

When collecting data to train our models, the main focus was on not contaminating the workloads with additional overhead. In particular, all kubernetes worker nodes were bare-metal, contrary to popular deployments of virtualized k8s.

The following product versions were used in our lab (R1 & R2 servers, as per defined specs):

- ESXi 7.0 Update 3g (Build 20328353)
- Kubernetes 1.23.10

Kubernetes was installed on Ubuntu Server 22 (bookworm/sid) running 5.15.0-47-generic SMP Kernel.

One of the biggest challenges we had to address were CPU hyperthreading mitigations. Over the past few years, multiple exploits surfaced using the HT as an attack vector. Since the vulnerabilities are inherently inside the CPU hardware, there are no solutions other than removing some of the HT optimizations using the so-called mitigations.

Unfortunately, those result in serious performance penalties, sometimes exceeding 50%. In addition, they affect our measurements by introducing ambiguity, as it's not easy to determine if reduced performance comes from virtualization overhead or HT mitigations.

For that reason, to level the playfield between both technologies, we have disabled all known mitigations on both platforms. This involved:

- for ESXI - we disabled "Side Channel Mitigation" option on all the hosts as well as disabled kernel mitigations in VM guests
- for K8s - we disabled kernel mitigations in host OS

In order to disable kernel mitigations, we've used the following set of boot parameters:

noibrs noibpb nopti nospectre_v2 nospectre_v1 l1tf=off nospec_store_bypass_disable no_stf_barrier mds=off tsx=on tsx_async_abort=off mitigations=off

There were two sets of experiments performed during our research:

A. *Space & Memory Spectrum (Python / Celery)*

We chose Celery as the main tool to distribute and measure workloads across multiple VMs and containers. The reasons behind this choice were:

- easy to control the number of concurrent threads (workers)
- easy to control resource allocation
- ability to measure actual execution time.

The last point was crucial in order to eliminate the container spin-up times from time calculations. We show the summary timeline in a table below.

The actual celery manager process, as well as the communication layer (Redis 4.7.1) were hosted outside of the evaluated environment.

#	**Supervisor**	**K8s**	**VM**
1	(idle)	(idle)	Launch VMs
2	Submit tasks	Trigger workload	**Time Start**
3	(idle)	Launch Pods	(payload)
4	(idle)	**Time Start**	**Time Stop**
5	(idle)	(payload)	Return Time
6	(idle)	**Time Stop**	-
7	(idle)	Return Time	-
8	Collect Tasks	-	-
9	Save Classification Result (K8s / VM)		

The measurements were repeated numerous times, changing the simulation parameters as follows:

- **X-axis**: memory allocation per a single pod or VM, starting from 2 GB up to 36 GB with 2 GB increments. The number of pods/VMs was then adjusted to fill up the host machine keeping 2 GB free for host OS.
- **Y-axis**: changing the payload through python library calls and manipulating the time spent in kernel vs user space. For each payload, a separate measurement was performed using systemtap to confirm the ratio.

The outcome of Space & Memory experiments is the curved line shown in our sample model above dividing the Container and VM clusters (polynomial approximation).

B. *Java Bias Vector*

In order to measure the Java bias, we first observed the dominant performance of VM based deployment. Then, we've prepared a set of test business workloads with varying user/kernel ratio using off the shelf products, such as Libre Office (headless + PDF + SSL).

Then, a baseline performance was established using VM configuration. Time measurements were performed and recorded for a limited subset of 12 spectrum datapoints (3 ratios x 4 memory configurations).

Afterwards, the same workload was deployed using a container, making sure to match as many underlying libraries as possible (in particular glibc and openssl). We then tried to change X and Y axis values in order to match the median baseline outcome. The result is our JVM Bias Vector.

VIII. Conclusion

A. *Database Schema & The Classifier*

Upon conclusion of our tests, we put together an endpoint designed to store the results as well as use them to train a classifier model which will later be used to reply to LPE client requests.

Data collected during experiments has been structured in the format described above, and is stored in a relational table. The main data tape format is presented below:

Column Name	**Type**	**Description**
user_ratio	Float (0.0 - 1.0)	Proportion of time spent in user vs kernel space
mem_size	Integer (MB)	Memory allocated to a single computation unit
category	Enum (K/V)	Deployment yielding faster results (Container/VM)
is_java	Boolean (Y/N)	Is the workload predominantly Java based

In addition to raw results, we also stored auxiliary data, such as experiment IDs, hardware specs, CPU architecture, etc.

Results were collected as CSV files, and were ultimately stored in a PostgreSQL database. For the training model, we used Keras classifier based on TensorFlow decision models. It was then connected to a Flask based web service, which served as an LPE backend.

An LPE request would carry information about the memory consumed by the predominant group of processes, measured user to kernel ratio, number of predominant threads, CPU architecture and a Java flag. Note that not all of those features were ultimately used in decision making.

A sample request is shown below:

{"mem_main": 2695, "user_time": 0.9185, "java_main": 0, "cpu_arch": "x86_64", "threads_main": 9}

The backend would then reply with:

{"class": "K", "bias_applied": 0}

In this case, it is recommended to use containers for production deployment.

A sample classification result was shown in chapter VI. Over 600 tests were performed in order to train the model. We trained two separate classifiers for Java and non-Java domains and then aggregated the differences into a bias vector.

B. Real Time Application

It is worth noting that despite a seemingly dominant role of VM category, most real life applications have the following parameters:

- U/K Ratio ~ 0.95
- Memory Alloc ~ 0.75 GB

This observation was based on our commercial experience, where we were able to measure real life business applications, such as fintech ledger systems, marketing platforms and DevOps tools.

As such, it is no surprise that containers remain a valid choice for the majority of commonly used software. A notable exception here are Java based applications, which are - however - usually delivered as singletons (i.e. single node) rather than scalable clusters of pods or VMs.

References

1. Desai, Prashant., 2016, A Survey of Performance Comparison between Virtual Machines and Containers. INTERNATIONAL JOURNAL OF COMPUTER SCIENCES AND ENGINEERING.
2. Bermejo, B., Juiz, C., 2022, A general method for evaluating the overhead when consolidating servers: performance degradation in virtual machines and containers. J Supercomput
3. Djob Mvondo, Antonio Barbalace, Alain Tchana, Gilles Muller., 2021, Tell me when you are sleepy and what may wake you up!. SoCC 2021
4. Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes., 2016, Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade
5. Tirmazi, Muhammad, et al. 2020, Borg: the next generation., 15th European conference on computer systems..
6. Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks., 2015, Profiling a warehouse-scale computer, 42nd Annual International Symposium on Computer Architecture
7. Simon Eismann, Long Bui, Johannes Grohmann, Cristina L. Abad, Nikolas Herbst, Samuel Kounev, "Sizeless: Predicting the Optimal Size of Serverless Functions"
8. Yuxuan Zhao, Alexandru Uta, "Tiny Autoscalers for Tiny Workloads: Dynamic CPU Allocation for Serverless Functions"
9. Marios Kogias, Rishabh Iyer, Edouard Bugnion, "Bypassing the Load Balancer Without Regrets"
10. Ataollah Fatahi Baarzi, George Kesidis, "SHOWAR: RightSizing And Efficient Scheduling of Microservices"
11. Dinh Tuan, Hai & Mora, Maria & Beierle, Felix & Garzon, Sandro. (2020). Development Frameworks for Microservice-based Applications: Evaluation and Comparison. 10.1145/3393822.3432339.
12. Ueda, Takanori & Nakaike, Takuya & Ohara, Moriyoshi. (2016). Workload characterization for microservices. 1-10. 10.1109/IISWC.2016.7581269.
13. K. Lei, Y. Ma and Z. Tan, "Performance Comparison and Evaluation of Web Development Technologies in PHP, Python, and Node.js," 2014 IEEE 17th International Conference on Computational Science and Engineering, 2014, pp. 661-668, doi: 10.1109/CSE.2014.142.
14. S. S. N. Challapalli, P. Kaushik, S. Suman, B. D. Shivahare, V. Bibhu and A. D. Gupta, "Web Development and performance comparison of Web Development Technologies in Node.js and Python," 2021 International Conference on Technological Advancements and Innovations (ICTAI), 2021, pp. 303-307, doi: 10.1109/ICTAI53825.2021.9673464.
15. Diomidis Spinellis, Paris Avgeriou, 2021, Evolution of the Unix System Architecture: An Exploratory Case Study, 10.1109/TSE.2019.2892149
16. Xiang (Jenny) Ren et al., 2019, An Analysis of Performance Evolution of Linux's Core Operations, 10.1145/3341301.3359640
17. Prajwal Kiran Kumar et al., 2016, Performance Characterization of Communication Protocols in Microservice Applications, 978-1-6654-3545-1
18. S. Athlur, N. Sondhi, S. Batra, S. Kalambur and D. Sitaram, "Cache Characterization of Workloads in a Microservice Environment," 2019 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM), 2019, pp. 45-50, doi: 10.1109/CCEM48484.2019.00010.
19. Anshuj Garg, Purushottam Kulkarni, Uday Kurkure, Hari Sivaraman, Lan Vu, 2019, Empirical analysis of hardware-assisted GPU virtualization, 10.1109/HiPC.2019.00054
20. Mochi Xue, Jiacheng Ma, Wentai Li, Kun Tian, Yaozu Dong, Jinyu Wu, Zhengwei Qi, Bingsheng He, Haibing Guan, 2018, Scalable GPU

Virtualization with Dynamic Sharing of Graphics Memory Space,
10.1109/TPDS.2018.2789883