# bhyve - Improvements and Comparison for the live migration feature for the wired and non-wired guests

Elena Mihailescu
*Faculty of Automatic Control and Computer Science*
*University Politehnica of Bucharest*
Bucharest, Romania
maria.mihailescu@upb.ro

Mihai Burcea
*Faculty of Automatic Control and Computer Science*
*University Politehnica of Bucharest*
Bucharest, Romania
mihai.burcea2508@stud.acs.upb.ro

Darius Mihai
*Faculty of Automatic Control and Computer Science*
*University Politehnica of Bucharest*
Bucharest, Romania
darius.mihai@upb.ro

Mihai Carabas
*Faculty of Automatic Control and Computer Science*
*University Politehnica of Bucharest*
Bucharest, Romania
mihai.carabas@upb.ro

***Abstract*—Nowadays, one of the most important aspects when it comes to cluster and grid management is client's services availability. When a cluster node fails or various maintenance operations need to be done, the virtual machines are usually live migrated on other nodes.**

**FreeBSD is an operating system that is mainly used in server environments. Nevertheless, its hypervisor, bhyve, does not have a live migration feature added in upstream, yet. However, progress towards adding live migration support was made in the last years by the students of University Politehnica of Bucharest. This paper presents the latest changes proposed by the students of the same university, and measures the impact of these changes compared to the previous implementation.**

***Index Terms*—bhyve, live migration, cloud infrastructure, wired guests, non-wired guests**

## I. INTRODUCTION

In the last years, we can see a shift from on-premises hosted services to cloud-hosted ones [1] [2] [3] [4]. The advantages of running in the cloud (easier management, pay only for the resources you use, easier up or downscale when needed) weigh more to small to medium businesses than taking care of the same services on-premises.

Thus, cloud providers' work must ensure that their services (the resources they give to clients) run smoothly and without issues that may affect the clients and their end-users. A cloud provider may run their services in a:

- private cloud - the infrastructure is still on-premise, but with automatic deployment and resource management.
- public cloud - the infrastructure is public and a client pays for the resources they use.
- hybrid cloud - a combination between the two of the above

In order to have clients and a running business, a cloud provider must meet a number of requirements [5] [6]:

- resource management - each client must receive the number of resources they request; the background operations must be completely transparent for the user.
- reliability - the resources must be always available and the services must be running continuously.
- scalability - the architecture must scale up or down (i.e., allocate more or less resources), depending on the client's needs.
- high availability - even if one of the nodes crashes or needs to be replaced, the client's resources and services must remain available.
- distributed environment - a high-available infrastructure must be implemented using nodes distributed across multiple data centres.
- security - the clients' resources must be isolated; in case of a security breach in one client's services, the rest of the clients must not be affected.

When choosing between various methods of setting up a cloud environment, the cloud providers must take into account their needs and the available technologies [7] [8]. In order to ensure the above-mentioned requirements, they may choose between native virtualization using virtual machines or containerization. Because of the flexibility, security and resource isolation they provide [8] [10] [11], cloud providers usually choose native virtualization for their business. Moreover, they may choose various orchestration technologies such

The authors would like to thank Matthew Grooms and Modirum for their financial support in forms of scholarships for the students that worked on these features

as OpenStack[1] or OpenShift[2] for easier usage.

One of the features that come in help when balancing the load on the nodes or ensuring high-availability is the virtual machine migration functionality. Hypervisors such as Hyper-V, VMWare, Xen or KVM allow their users to migrate the virtual machines from one host to another. Nowadays, migration is a necessity when managing a cloud infrastructure.

Virtual machine migration means to move a guest from one node to another while the guest is running. There are several methods for migration: cold migration (which means to move the guest image while the guest is powered off), warm migration (pause the guest's execution and move it to another node and, then, restore its execution) and live migration (move the guest while it is still running, with the smallest possible downtime).

When choosing between operating systems (and hypervisors) that are deployed on the physical nodes, the system administrators need to choose between various operating systems. Usually, they choose between BSD-based and Linux-based operating systems.

Considering the advantages it brings (mature, complete operating system with both kernel and user space implementations, BSD license, network stack, security enhancements such as CAPSICUM etc.), FreeBSD is a good candidate as an operating system in cloud infrastructure. However, bhyve, its hypervisor, lacks in providing a migration implementation yet.

This paper presents the improvements added to the FreeBSD's hypervisor in order to support the migration of non-wired guests as well as a comparison between the times needed for wired versus non-wired guests. The end goal is to create a robust migration functionality that can help system administrators to use FreeBSD as an operating system when setting up their cloud infrastructure.

With this study, we aim to measure the impact of the current implementation in the migration process and to check the overhead brought by the swap in-swap out operations. Moreover, this paper aims to detail the scenarios we have used and their results to confirm that our implementation [15] can be further taken into consideration for upstreaming.

This paper is structured as follows: the first section shows the current global context related to the virtual machine migration in cloud infrastructures; the second section describes the migration mechanism that is implemented in bhyve and used as a reference and measurement baseline for the current paper; the third section presents the improvements we have added to bhyve. The fourth section depicts the setup and testing procedure, while the fifth section shows our results. In the last section, we conclude our paper.

[1]https://www.openstack.org/, last accessed 3rd of March, 2023

[2]https://www.redhat.com/en/technologies/cloud-computing/openshift, last accessed 3rd of March, 2023

## II. Baseline migration implementation in bhyve

As presented in [12] [13], students from University Politehnica of Bucharest have implemented warm and live migration features for bhyve. The later works only for wired guests (i.e., their memory is allocated beforehand and cannot be swapped). However, a full live migration feature must work on non-wired guests as well.

Between the migration types, the most complex one, but with the smallest downtime (i.e., the time in which the guest is unavailable) is live migration. To allow the guest's memory movement from one host to another, the migration is done in rounds. While the guest runs on the source node, its memory is migrated to the destination node. In the first round, all the guest memory is moved across the network. In the next ones, only the pages that were modified during the previous rounds are migrated. The memory changes are tracked using a custom dirty bit [12].

The previous live migration implementation [12] [13] needs the physical pages to be allocated and present in the main memory in order to traverse and retrieve the pages that were modified between rounds. Moreover, it copies the pages' content from kernel space to user space and, then, sends it over the network. However, this approach adds overhead through:

- context switches - the implementation switches from user space (bhyve process) to kernel space to inspect the pages that are dirty, then switches back to user space to report the results, then back to kernel space to start copying the modified pages.
- buffers - the implementation duplicates the memory content: various buffers are allocated to retrieve the page content from kernel space while the guest's memory is already mapped in the user space bhyve process (indicated by `vmctx->baseaddr`.

## III. Improvements of the live migration feature in bhyve

This section proposed various changes to improve the live migration feature in bhyve.

### A. Non-wired guests support

One of the most important improvements added to the live migration feature is the extension to the non-wired guests. This addition offers bhyve a robust feature that works without memory allocation constrains.

For this improvement, we checked the page state. If it is not allocated, the page does not need to be migrated, thus being ignored. If the page is allocated, but swapped out, the page is brought into the memory and migrated, taking into consideration that the VMM dirty bit is also saved on disk.

### B. Remove multiple copies of the same data

The baseline implementation relayed on multiple copies of the same information between kernel space and user space. As depicted in Figure 1, the baseline implementation [12] needed `ioctl` calls to copy the page content from the kernel space to the user space. Then, this data was sent through a socket

to the destination where, using another `ioctl` call, the page is written in the kernel space.

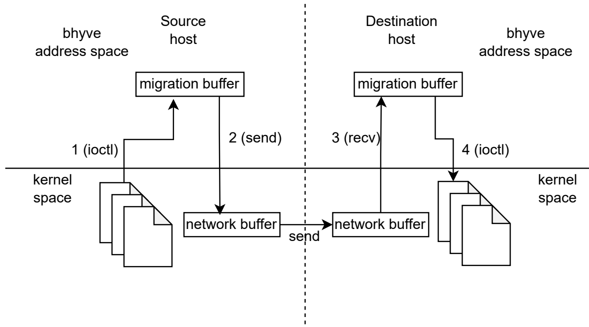



Fig. 1: Memory migration in the previous implementati

The previously presented approach may not be suitable s it generates multiple context switches to access the des pages. Moreover, modifying the pages directly in the ke space may be considered a dangerous operation and can to vulnerabilities (e.g., accessing the wrong pages may lea kernel panic).

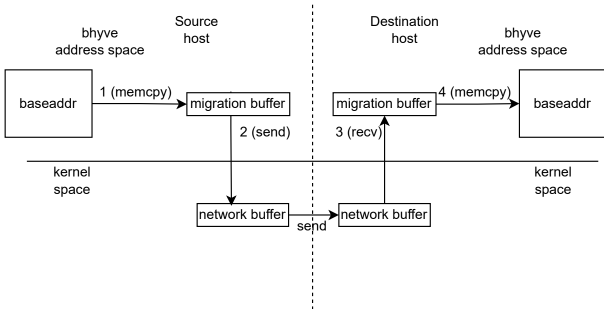



Fig. 2: Memory migration in the current implementation

Thus, we changed the approach. Instead of copying the page's content from the kernel space, we are using the guest's memory-mapped area in the bhyve user space process. Based on the starting memory address (`vmctx->baseddr`), the page index, and the page size, we can determine the memory area corresponding to the desired page. The new process is depicted in Figure 2.

### *C. Link pages to bhyve's memory*

In theory, the changes proposed in Section III-B should have improved the migration time. However, during tests, and as presented in Section V, we observed that Round 1 took as much time as Round 0, even if the guest's memory activity was almost nonexistent (see Figure 5a and 5b).

Further debugging showed that this behaviour is caused by the bhyve's dual memory view [12]. Figure 3 hints the process: the same physical pages are once accessed by the guest through the nested page table [14] and once by the bhyve user space process that also maps the guest's memory (starting from `vmctx->baseaddr`).

The guest interacts with the physical pages, through the nested page tables, in a normal fashion. However, the link between the bhyve's virtual address and the guest's physical page is not created when the memory segments are mapped. Thus, acting similar with on-demand paging, the link is created the first time that page is accessed from the user space (e.g., from the emulated devices that need to interact with the guest's memory).

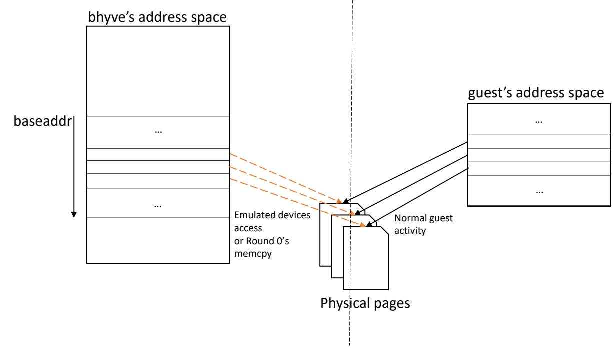



Fig. 3: Dual memory view

In our case, the round duration anomaly we observed was determined by this dual memory view: the first time the guest's memory was accessed from the bhyve user space process was during migration's Round 0. Thus, the dirty bit was set again for all the guest's pages. The solution was to hint the virtual memory subsystem, using the `madvise()` function with `MADV_WILLNEED` as behaviour, that those pages need to be immediately mapped, without page faulting since they are already allocated.

## IV. Setup and Testing Procedures

The testing setup is designed to mimic a cluster architecture: we have multiple same-type nodes that are connected in the same network. In addition, we have a distributed storage between them where the guest's disk image is kept.

Due to the fact that one of our main goals is to compare the live migration enhancements to the previous version, we propose the same setup as presented in [13]. The setup is depicted in Figure 4 and contains:

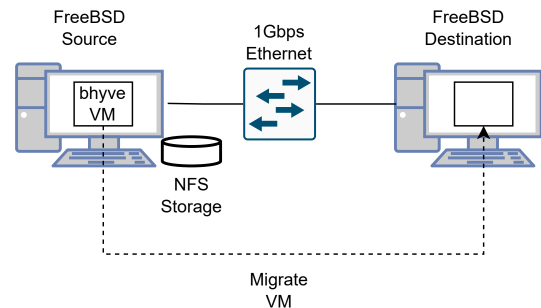



Fig. 4: Testing setup

- two identical FreeBSD hosts with 4 cores and 16GB of RAM. As CPU, both run on an Intel(R) Core(TM) i7-4790 @ 3,60GHz. They run the same version of FreeBSD with the same migration code. Both systems use HDDs for storage, and both have configured up to 16GB of swap.
- a distributed (NFS) storage that contains the guest's image file. As opposed to the setup described in [13], the NFS storage is hosted on one of the two FreeBSD systems described above.
- a 1Gbps Ethernet network through which the hosts and the distributed network storage can communicate.

For testing, we used a FreeBSD guest that is started on the FreeBSD source with various parameters and, then, is migrated to the FreeBSD destination host. Since we aim to check our changes over the old test-bed, we run the same two types of tests:

- a simple test that starts the virtual machine, logs in, and, waits for commands.
- a memory stress test that starts the virtual machine, allocates a lot of memory and, then, continuously reads and writes one byte from each allocated memory page.

While the former does not impact the guest's virtual memory, the latter heavily modifies almost all the available memory. These two are the most extreme use-cases: the first one almost does not require more than 2 rounds, while the second one, in a real-life situation, would probably require a warm migration procedure.

In terms of migration parameters, we tested both wired and non-wired guests, adjusting the virtual machine's memory from 1 to 12GB of RAM for the former and from 1 to 14-16GB of RAM for the latter. For each test, we measured each round and the downtime. The downtime is considered to be the time in which the guest is stopped. It is composed from the following times:

- the last round duration;
- the guest's state (CPU's, emulated and virtualized devices' state) snapshot time;
- network transfer time;
- memory and guest's state restore time;
- migration completed message time.

## V. Results and Current limitations

As stated, this paper also offers a comparison between the old and the new approach in terms of downtime in various scenarios. The scenarios are based on the ones defined in [13] and are meant to test extensive guest functionality. We are measuring the total migration time and downtime for wired and non-wired guests.

Figures 5f and 6 present each round of migration for each test comparing the times each of them took. Each graphic presents a comparison between the previous implementation and the new one. Moreover, we considered the non-wired live migration scenario.

From both figures, we can observe that:

- Round 0 lasts almost the same in the previous implementation as in the new one for the wired guests. Since in the first round, all the guest's memory is considered to be dirty and transferred to the destination, it means that the current implementation does not bring additional overhead.
- Round 1 takes almost as long as Round 0 for both test case scenarios, for both wired and non-wired guests. However, this did not happen in the previous implementation. This is the result of the dual memory view presented in Section III-C and is corrected in Figure 7.
- All the rounds for the non-wired guests take less time than their wired-memory counterpart tests (both with old and the new implementation). This behaviour is normal since the memory that is not used (or allocated beforehand) needs to be migrated. Moreover, this also shows that the non-wired implementation does not bring additional overhead.
- The memory stress test (Figure 6) shows that all the rounds take almost the same time to complete since almost all the guest's memory is migrated in each round.
- The non-wired memory guest migration support allowed us to extend our testings for guests with more than 12GB of RAM. For the simple test scenario, we managed to create and migrate virtual machines with up to 16GB of RAM, while for the memory stress test, we managed to allocate up to 14GB of RAM. As seen, the behaviour is similar to the previous tests.
- In all the cases, the migration bottleneck is the network since the 1Gbps Ethernet connection is saturated (e.g., for 6GB, the minimum transfer time over the 1Gbps network is around 50 seconds, like in our results).
- When testing the memory stress test scenario for guests with more that 13GB of RAM, part of the memory was swapped. Thus, the live migration works even if the pages are swapped out.

These extensive testing procedures (with more than 100 run tests) allowed us to fine-tune our solution and solve some non-trivial issues. However, while testing we also observed various limitations to our implementations that will be presented in the following paragraphs.

The first issue we found is related to the dual-memory view that is presented in Section III-C. Due to the fact that we moved the memory transmission part in the user-space, each copy from Round 0 would generate a page fault from the bhyve's user space tool. Thus, the guest's page would be marked as dirty again and copied in the next round (Round 1). The solution presented in Section III-C improved the migration time.

A comparison before and after this improvement with regarding the first two rounds can be seen in Figure 7. The time drops significantly in Round 1 for the simple test scenario after we added the improvements. For the memory stress test scenario, we can see that the memory that is migrated is the one that is modified by our testing executable.

Another issue we were facing is testing the memory stress

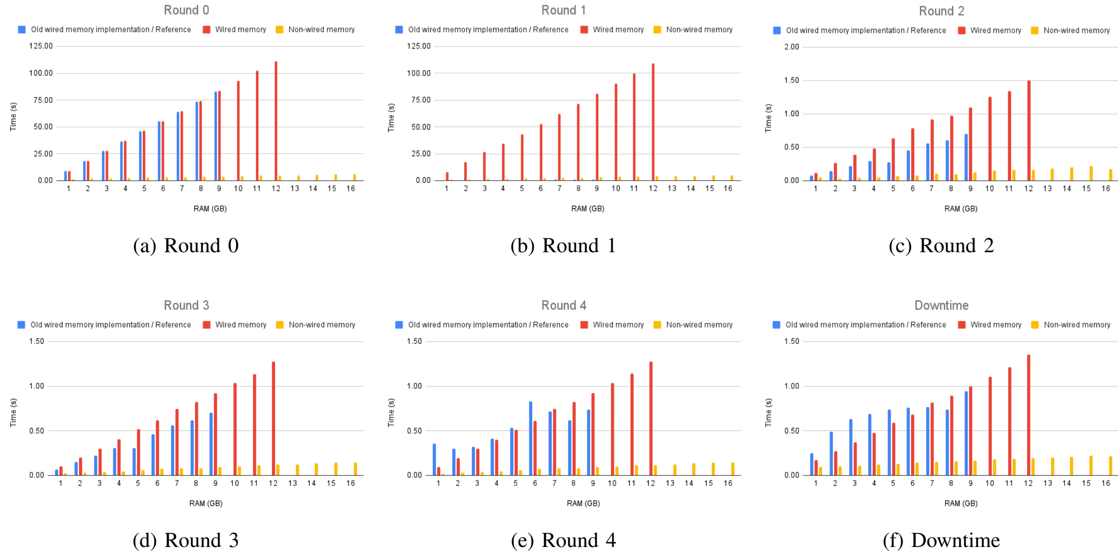



Fig. 5: Simple Test - Rounds Comparison

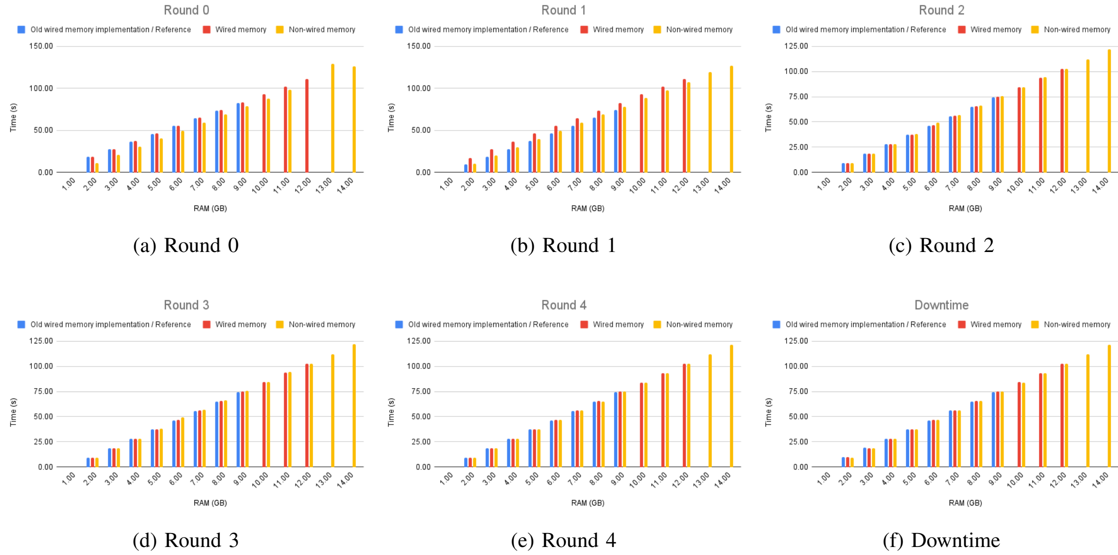


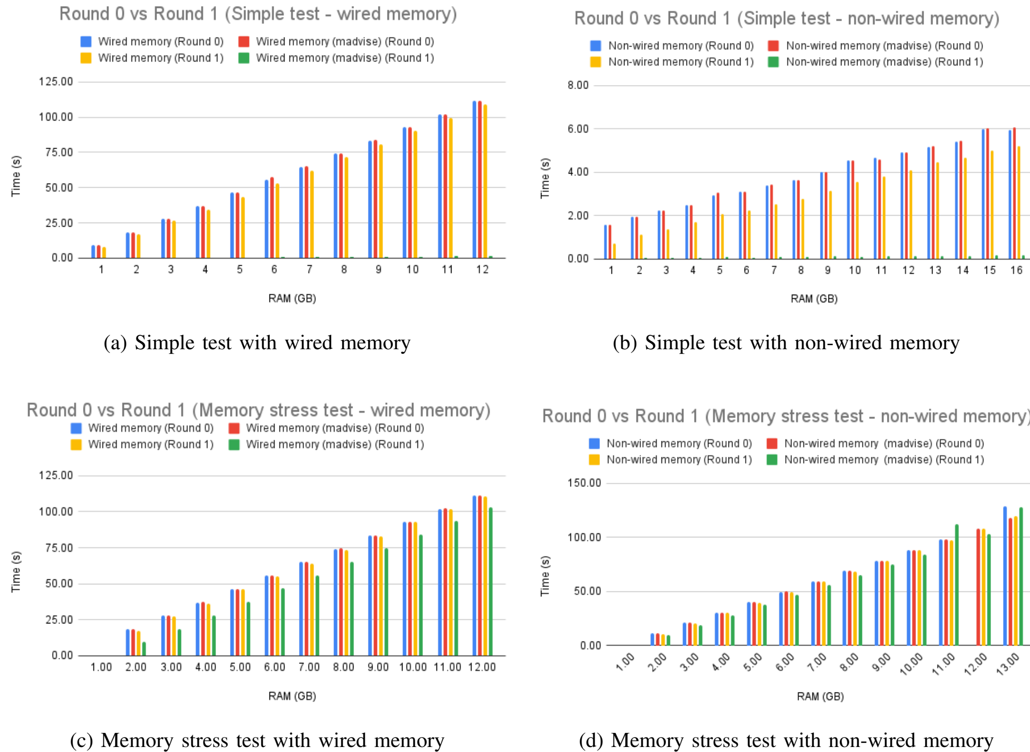

Fig. 6: Memory Stress Test - Rounds Comparison

test scenario with guests with more that 15GB of RAM in hosts with 16GB of RAM. Allocating 15GB for the virtual machine and constantly writing at least 14GB of RAM inside the virtual machine, led the operating system to swap out either virtual machine pages or bhyve's user space tool pages which led the migration process to be very slow. The tests were inconclusive since sometimes the rounds took as expected (e.g., 120-130 seconds per round) and other times the rounds would take up to 2h. Moreover, the host system would start crashing (out of memory) or be unresponsive (the services would be killed) due to the swap trashing process. Furthermore, increasing the guest's RAM size and allocating $RAM_SIZE - 1$ GB leads to less than 200MB of RAM available in the guest.

## VI. Conclusion and Further Work

This paper presents the improvements added to the bhyve's live migration feature. We extended the live migration functionality to non-wired guests and fixed various issues (e.g., double buffering, on demand paging for Round 1). Then, we tested the implementation using two extreme use-cases: a simple test in which the guest does almost nothing (i.e., very few pages are modified between rounds) and a memory stress test where almost all the guest's memory is continuously written (i.e., almost all the guest memory must be migrated in each round).

Our aim is to have a fully functional live migration support




(a) Simple test with wired memory

(b) Simple test with non-wired memory

(c) Memory stress test with wired memory

(d) Memory stress test with non-wired memory

Fig. 7: Rounds 0 and 1 comparison before and after using `madvise()`

for bhyve that can be used by the FreeBSD's community. Thus, based on the results we have obtained, we can conclude that the recent changes do not impact the performance of the live migration feature. The bottleneck in migration is either caused by the network (1Gbps Ethernet in our case) or the swap area (the swap out/swap in operations are very time consuming due to the slow HDD read/write operations).

During the time this paper is written, we have a series of eight open reviews on Phabricator [15] that split our implementation in smaller parts: the first five are related to warm migration, and the last three are for live migration. The implementation is open-source and available on GitHub [16].

As future work, we plan to further improve our implementation by adding various fine-tuning options such as bandwidth rate limitation, dynamic number of rounds or maximum migration time. Moreover, we want to test our feature using real-life test scenarios (e.g., web servers) and measure the user experienced downtime.

## References


[1] Jelassi, Mariem and Ghazel, Cherif and Saïdane, Leila Azzouz, "A survey on quality of service in cloud computing", 2017 3rd International Conference on Frontiers of Signal Processing (ICFSP), pages 63–67, 2017, IEEE.

[2] Gelenbe, Erol and Lent, Ricardo and Douratsos, Markos, "Choosing a Local or Remote Cloud", 2012 Second Symposium on Network Cloud Computing and Applications, pages 25–30, 2012, doi=10.1109/NCCA.2012.16.

[3] Fisher, Cameron and others, "Cloud versus on-premise computing", American Journal of Industrial and Business Management, volume 8, no 09, pages 1991, 2018, Scientific Research Publishing

[4] Pahl, Claus and Xiong, Huanhuan and Walshe, Ray, "A comparison of on-premise to cloud migration approaches", European Conference on Service-Oriented and Cloud Computing, pages 212–226, 2013, Springer

[5] Cong, Peijin and Xu, Guo and Wei, Tongquan and Li, Keqin, "A survey of profit optimization techniques for cloud providers", ACM Computing Surveys (CSUR), volume 53, number 2, pages 1–35, 2020, ACM New York, NY, USA

[6] Rashid, Aaqib and Chaturvedi, Amit, "Cloud computing characteristics and services: a brief review", International Journal of Computer Sciences and Engineering, volume 7, number 2, pages 421–426, 2019.

[7] Tomarchio, Orazio and Calcaterra, Domenico and Modica, Giuseppe Di, "Cloud resource orchestration in the multi-cloud landscape: a systematic review of existing frameworks", Journal of Cloud Computing, volume 9, number 1, pages 1–24, 2020, Springer

[8] Bittencourt, Luiz F and Goldman, Alfredo and Madeira, Edmundo RM and da Fonseca, Nelson LS and Sakellariou, Rizos, "Scheduling in distributed systems: A cloud computing perspective", Computer science review, volume 30, pages 31–54, 2018, Elsevier

[9] Mavridis, Ilias and Karatza, Helen, "Combining containers and virtual machines to enhance isolation and extend functionality on cloud computing", Future Generation Computer Systems, volume 94, pages 674–696, 2019, Elsevier

[10] Yadav, Anuj Kumar and Garg, ML and others, "Docker containers versus virtual machine-based virtualization", Emerging Technologies in Data Mining and Information Security, pages 141–150, 2019, Springer

[11] Doan, Tung V and Nguyen, Giang T and Salah, Hani and Pandi, Sreekrishna and Jarschel, Michael and Pries, Rastin and Fitzek, Frank HP, "Containers vs virtual machines: Choosing the right virtualization technology for mobile edge cloud", 2019 IEEE 2nd 5G World Forum (5GWF), pages 46–52, 2019, IEEE.

[12] Mihailescu, Maria-Elena and Carabas, Mihai, "FreeBSD-Live Migration feature for bhyve", AsiaBSDCon, 2019

[13] Mihailescu, Maria-Elena and Mihai, Darius and Carabas, Mihai and

Tapus, Nicolae, “Improving and testing live migration for bhyve“, 2022 21st RoEduNet Conference: Networking in Education and Research (RoEduNet), pages 1–5, 2022, IEEE

[14] Natu, Neel and Grehan, Peter, “Nested paging in bhyve“, The FreeBSD Project, 2014

[15] FreeBSD-UPB, Reviews for the migration feature in bhyve, Online: https://reviews.freebsd.org/D34717, https://reviews.freebsd.org/D34718, https://reviews.freebsd.org/D34719, https://reviews.freebsd.org/D34720, https://reviews.freebsd.org/D34721, https://reviews.freebsd.org/D34722, https://reviews.freebsd.org/D34811, https://reviews.freebsd.org/D34813, Last Access: 1st of March, 2023

[16] FreeBSD-UPB, Live migration code. Online: https://github.com/FreeBSD-UPB/freebsd-src/tree/projects/bhyve_live_migration, Last Access: 1st of March, 2023