An Unikernels Provisioning Architecture for OpenStack

Luis Augusto Dias Knob*†, Bruno Gomes Xavier*, Tiago Ferreto*
*Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Porto Alegre, Brazil
†Federal Institute of Rio Grande do Sul - Campus Sertao
Rodovia RS 135, Km 25 – Sertao, Brazil
Email: {luis.knob, bruno.xavier}@acad.pucrs.br, tiago.ferreto@pucrs.br

Abstract—**Recently, the development of Unikernels, extremely light operating systems constructed exclusively for virtualized environments, has emerged as another alternative to cloud platforms. Unikernels encourage new paradigms and provisioning techniques, such as immutable servers and microservices. This paper proposes an optimized architecture for Unikernels provisioning in OpenStack. The architecture is based on OSv Unikernels and experiments demonstrated that it can attain satisfactory results, reducing the time of the entire deployment workflow, from the compilation to the initialization of the Unikernels in the compute node.**

I. Introduction

The concept of immutable servers [1] has emerged as a practice to solve problems inherent in configuration deviations arising from systemic changes in environments [2], [3]. The main characteristic desired in a context of immutable instances is the agility in the reconstruction of systems in order to guarantee a reliable state. From this point of view, any change linked to the service or upgrade configuration implies in the complete restoration of the system with new images, thus avoiding the accumulation of configuration residues. Once put into production, an instance is never altered; but replaced. This method takes advantage of application architectures evolution to a new paradigm, in which systems formerly composed of a set of components allocated in dedicated machines, are redesigned into smaller, specialized and easily maintainable services. This microservices-based architecture [4] enhances elasticity, portability, scalability, and fault isolation.

At the same time, lighter virtualization mechanisms are being developed as alternatives to popular operating systems running on hypervisors [5], for instance, Linux Containers and Unikernels [6]. Containers, as well as traditional virtual machines, rely on heavy images depending on the application. In a cloud environment, these images need to be downloaded and imported to their provisioning destination. Assuming that an application is made up of many services, this transfer and handling time is expected to increase when these services are provisioned at the same time. Likewise, although Unikernels have an extremely lightweight architecture and small image size, they can still be impacted depending on the number of services or instances provisioned simultaneously.

This paper proposes an optimized architecture for Unikernels provisioning in OpenStack. The architecture is based on OSv [7] and focus on reducing the provisioning time when deploying several instances. Based on the immutable feature of unikernels, where images are always rebuilt under the need of changes, the compilation time and image transfer must be included in the total provisioning time. The proposed architecture is evaluated and compared to the default OpenStack driver.

II. Unikernels provisioning architecture

Based on the modular architecture of OpenStack, the architecture core is based on a driver for distributing Unikernels on the cloud platform. This work involves a design to take advantage of the main benefits of the Unikernel architecture for accelerating the provisioning of service stacks. A Unikernel represents an immutable appliance, which, once compiled, has an image that is not modified on any aspect, either in terms of service configuration, or modifications in libraries. Thus, the reprovision of one or more services requires the complete reconstruction of the images and their subsequent deployment flow in the cloud, which comprises the following steps:

1) Image compilation;
2) Sending images to the storage service (Glance component in OpenStack);
3) Copying images to destination machine (a compute node);
4) Provisioning the new images in virtual instances;

This provisioning model presents bottleneck points depending on the available network and computing resources. With the increase in provisioned services, there is an increased use of the network link between the image storage service and the computing nodes. Similarly, Unikernels require a location to be rebuilt, which would normally be accomplished by a client server outside the cloud. Although Unikernels produce small size when compared to traditional images, such as Linux, the increase in the demand for services tends to cause a gradual increase in the consolidated size of the images that will be provisioned. Moreover, the reconstruction of a service stack also involves the reconstruction of these images.

From these observations, an architecture was designed based on a version control system, with the reconstruction of the images occurring inside the node where they are provisioned (Figure 1).

In this image, Glance replaces the images physical storage by the reference to the version control repository corresponding to each Unikernel. This reference is then queried by the driver for the pull execution in case of changes. Each computing node contains a Git server, responsible for storing Unikernel codes (Unikernel repository). Hence, the Glance project was modified to accept these addresses instead of the name of the images, keeping the unique identifiers already used by the platform (Table I).

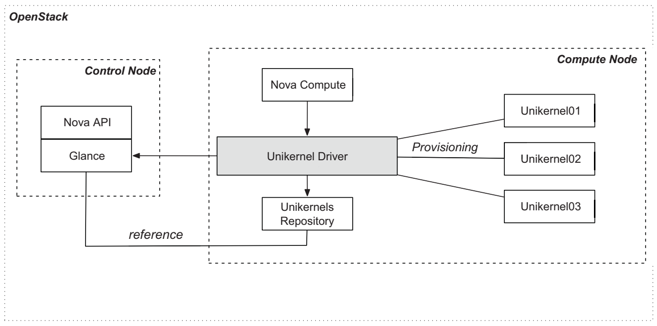


Fig. 1: Proposed Architecture

Identifier	Name
63c62fb4-3d9a-47a2-992e-26051982e865	git@10.32.45.217:/opt/git/apache-spark
101156bd-b4da-4473-b9cb-31e1c11cd314	git@10.32.45.217:/opt/git/apache-zookeeper
c8db95fc-8534-4a32-ae9a-9748988b1e90	git@10.32.45.217:/opt/git/haproxy

TABLE I: Unikernels repository references in Glance

Due to the utilization of OSv, images are based on the QCOW2 format. Therefore, the driver extends the functionality of libvirt Driver, already prepared for this type of image. In this way, the optimized driver is responsible for pulling the Unikernels code, compiling and converting to the RAW format. Then, libvirt driver is called, resizing the image for the chosen template and converting it back to the QCOW2 format, in the instance directory that will be provisioned.

In the event of competition in the provisioning of a single Unikernel, a synchronization mechanism was inserted on the method responsible for checking changes in the repository and reconstructing the image, if necessary. This synchronization uses the oslo_concurrency library, native to OpenStack and used to control the concurrency between threads and processes of all platform projects.

The platform already has a native disk caching mechanism for images copied from the Glance service. Therefore, the images resulting from the compilation, in RAW format, are recorded at this location for subsequent provisioning. These images have the same unique identifier represented by a hash, generated at runtime by the Nova API. However, the Unikernels repositories cloned from Git, are stored in a separate location, defined as the driver configuration parameter (Listings 1).

The Unikernels cloned and placed in Git remain on disk for change checks in the remote repository; the artifacts are retained for later incremental rebuilds. Each project is stored in a named directory with the identifier of the Glance repository, already presented in Table I. In this way, Unikernels of the same name, but from different sources, can be provisioned independently.

```
cfg.StrOpt('repo_base',
           default='/opt/stack/data/Unikernel',
           help='Unikernels repo path'),
```

Listing 1: Configuration of Unikernels repository path

A. *Resource isolation*

To prevent the Unikernels compilation interference to the creation of virtual instances, the driver was written with API access to Control Groups, native in the Linux kernel. Therefore, each compilation process is started respecting pre-configured limits of memory and CPU. These settings are entered as input parameters for the driver in the overall configuration of the Nova project. (Listing 2).

```
cfg.StrOpt('compile_core_limit',
           default=10,
           help='Percentage of CPU server total
    capacity'),
cfg.StrOpt('compile_mem_limit',
              default=2000,
              help='Memory used for compilation in
    Megabytes'),
```

Listing 2: Limit configuration of memory and CPU for compilation

The hierarchy defined for the compilation is shown in Figure 2. The stack group has recursive permission settings for the user who owns the OpenStack processes; while the osv subgroup was created only for the compilation processes, inheriting the attributes of the stack group. This design allows other subgroups to be created next to the osv group in an independent way, since different Unikernels architectures can be implemented and abstracted in the same driver in future works.

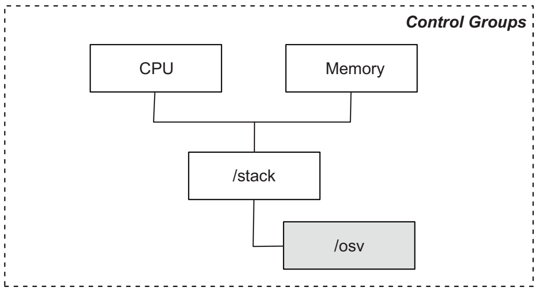


Fig. 2: Control groups hierarchy for Unikernels compilation

When started, the OpenStack nova-compute service, which manages the virtual instances, initializes the Unikernels driver, which in turn creates the hierarchy of control groups and applies resource constraints set by the administrator. In order for the OpenStack process to have authorization on the creation of control groups, a new filter file (Unikernel.filters) has been added next to nova, which allows the execution of operating system commands as the root user (Listing 3).

```
[Filters]
user_cgroups: CommandFilter, /usr/local/bin/
    user_cgroups, root
```

Listing 3: Filter for Control Groups of creation authorization

B. Image Reconstruction

The reconstruction of the images is made by the Capstan tool, provided by OSv to build Unikernels and test execution. Through a YAML descriptor, parameters are passed defining modules dependencies, configuration files, startup command and the compiler used. Once this process is called by the driver, its identifier (PID) is inserted into the control group of the isolation subsystems already described. Thus, the Unikernel is compiled with the desired amount of resources (Listing 4). This operation is done through the _add_pid_to_cgroup function, which is inserted into the execute method, which in turn encapsulates operations performed within the computing node's operating system.

Capstan is written in the Go language, and its code has been modified so that instances can be compiled concurrently. By default, OSv depends on the ZFS file system [8] for the internal storage of configuration files and application binaries. After the compilation step, the Unikernel image is mounted locally so that these files are copied. This is done using the qemu-nbd tool, which is used to manipulate QCOW2 images. However, this process opens a socket on port 1099 for client access. Since compiling Unikernels can occur in parallel, the code has been changed so that each build process uses a distinct port and there is no access conflict.

```
def _add_pid_to_cgroup():
    pid = os.getpid()
    self.cg.add(pid)

utils.execute('capstan', 'build', image_name,
              preexec_fn=_add_pid_to_cgroup,
              env_variables=dict(environ,
    CAPSTAN_ROOT=base_dir),
              cwd=Unikernel_repo)
```

Listing 4: Resource isolation during compilation code

III. Evaluation

In order to evaluate the proposed architecture, two use cases were defined for the load tests:

1) **Differential provisioning**: Provisioning from the reconstruction of a set of service images. With this test, we sought to identify the efficiency in provisioning time using the optimized driver under a Unikernels modification scenario.
2) **Provisioning with cache**: Provisioning without changes to images, using the caching mechanism of the platform. This was intended to evaluate the overhead of the driver developed when compared to the default driver.

In this stage of evaluation, a load composed of 10 Unikernels was constructed, representing real services and with different image sizes. Although Unikernels are recognized as small systems, the size of its images vary according to the size of the compiled service (Table II). This heterogeneous provisioning was necessary to show that the sum of a large number of services tend to impact the concurrent provisioning. As a premise for the optimized driver, each Unikernel had its own Git repository created and its code sent to the computing node.

Service	Image Size	ID
Apache ActiveMQ	151M	1
Apache Zookeeper	140M	2
Apache Spark	368M	3
Cassandra	118M	4
HAproxy	29M	5
Service	**Image Size**	**ID**
Memcached	29M	6
Redis	31M	7
Solr	133M	8
Tomcat	111M	9
MySQL	36M	10

TABLE II: Used services in the experiments

For the concurrent start of independent services, the Heat [9] service was incorporated into the Rally tool. Heat is an OpenStack project for orchestration of virtual instances, whereby different images can be provisioned in parallel, simulating an application formed by a set of services. For comparison purposes, the tests were run with the developed driver and the default OpenStack driver. The GNU Parallel tool[10], which enables setting entry tasks for simultaneous execution, assisted in the collection times that are not part of the platform's scope. These values refer to the time of reconstruction of the images, the transport of these images to Glance and the modifications of the Unikernels transferred to the Git Server.

A. Differential provisioning

This case begins with the premise of a reconstruction of Unikernels from possible configuration changes in stacks that are already running. Thus, all Unikernels were previously provisioned at the computing node. Similarly, its images are also already allocated in Glance for testing with the default driver. Also based on this premise, this first experiment involves a distinct provisioning flow for the default OpenStack driver and the developed driver. While at the moment the Unikernels need to be recompiled beforehand and off the platform, in the second, the recompilation occurs during the provisioning process started by the OpenStack API (Figure 3).

The test comprises provisioning all services concurrently. In order to force update of the repositories, a minimal change of configuration in each Unikernel was applied before the executions. This change involves the exchange of a configuration attribute for each service. In the case of redis, for example, the client connection timeout was changed in the redis.conf file. Figure 4 displays the results of execution.

The results showed a time saving of approximately 45 seconds of provisioning using the implemented optimization. This represents, in total terms, an overhead of 216% of the

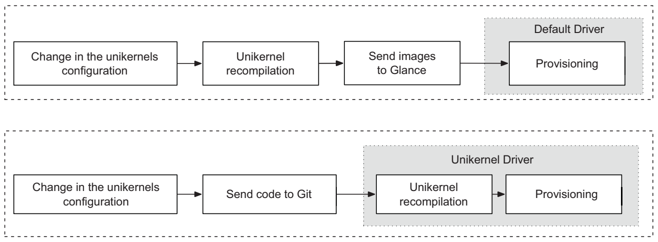


Fig. 3: Comparison between the provisioning flows evaluated

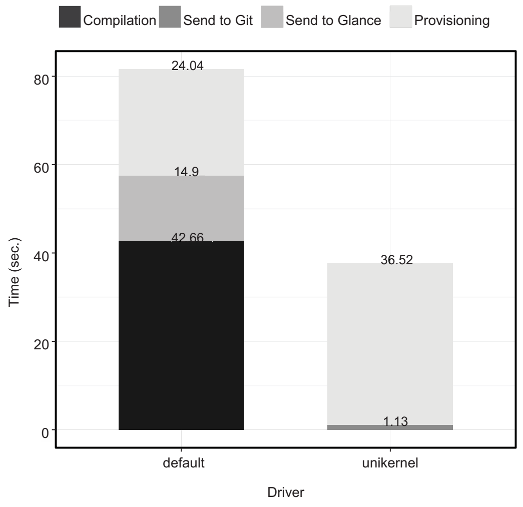


Fig. 4: Provisioning time for the service stack with the default driver and the optimized (Unikernel)

default flow on the implemented optimization. Whereas native provisioning requires a complete new copy of the images for Glance and then for the computing node, the optimized driver only copies the code changes from the local Git server. For this reason, its provisioning is optimized inside the node.

The network consumption, in the case of the default driver, has its impact influenced by the size of the images, whereas in the developed driver, it has to do with the amount of code changed. This specific test tried to show that a minimal modification in a service does not demand a new full copy of the image. Considering the reprovisioning of the 10 services load, in which only one configuration item was modified per Unikernel, it was necessary to transfer 1.2 Gigabytes of data concurrently to the computing node. This value can be doubled if the previous copies of the images of a client machine are considered for Glance. On the other hand, with the optimized driver, only the code differentials are sent to the repositories. Because these repositories are located locally, traffic between the image storage services (Glance) and the computing node is eliminated (Figure 5).

B. *Provisioning with cache and the Optimized driver overload*

As a second experiment, we evaluated the provisioning of services taking into account the use of images already

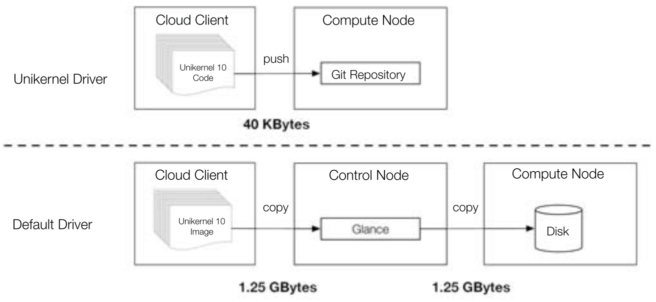


Fig. 5: Approximated network usage during provisioning

stored in the OpenStack cache. This test aimed to identify the overhead of the custom driver over the default driver. Because optimization requires a series of checks, both on repositories and on disk, efficiency in image recreation could be affected. In this case, we evaluated an incremental load, when the provisioning increased from one to ten competing instances (Figure 6).

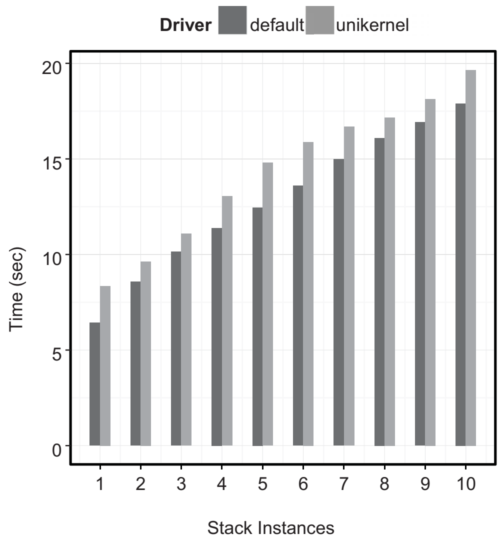


Fig. 6: Overload of optimized driver over the default driver

As expected, there is an overload in the recreation of virtual instances. However, this overload does not show a significant impact on total time, with an average increase of 1.5 seconds over the default driver provisioning time. The custom driver also proved stable and with a constant difference during the addition of competing services.

C. *Resources utilization*

As a last evaluation, the CPU and memory utilization from the computation node was collected during provisioning. These samples were conducted using dstat tool [11] with a margin of 5 seconds before the load startup and after finalization. Figure 7 shows CPU usage for 60 seconds, sufficient for initialization and removal of instances. In this first moment, the consumption was evaluated with the optimized driver compiling only the differential of the Unikernels, which did not represent some remarkable difference between the two drivers, with a peak of approximately 16% of use. The same was reflected in memory usage (Figure 8), with consumption of approximately 5 GBytes at the time all instances reached the state of execution. These results were expected, since there were no major changes in the Unikernels codes that justified a longer compilation time and intensive CPU.

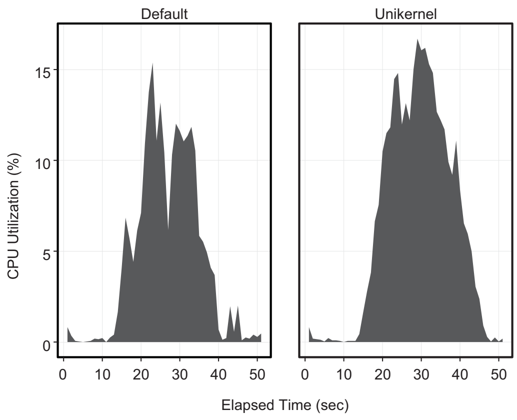
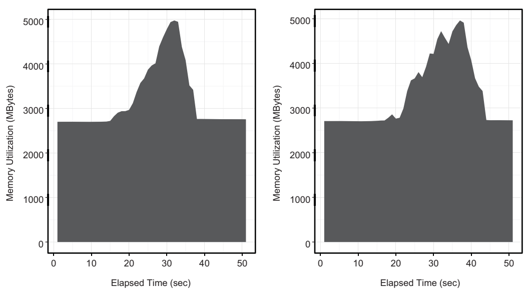


Fig. 7: CPU usage (Compute node)

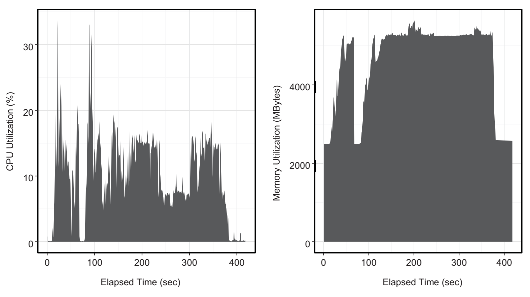
However, to verify the efficiency of the implementation of CPU and memory isolation in the reconstruction of the images, a second load was executed, forcing a complete compilation of the Unikernels inside the node. Thus, all caches of repositories have been removed, reproducing a scenario where the load is instantiated for the first time. The provisioning was done with an allocation of 12% CPU, this represents, in values, approximately 4 virtual cores. Also 2 GB of memory was reserved in the driver configuration. Figure 9 displays the results.



(a) Memory usage with the default driver (b) Memory usage with the optimized driver

Fig. 8: memory usage (Compute node)

In fact, the overload can be observed at peaks around 32%, at which time all instances are being compiled in parallel. The same overload can be observed in the memory consumption, around 5.6 Gbytes, evidencing an increase of approximately 600 MBytes in relation to the previous tests. This small variation of the latter is explained by the fact that the compilation



(a) CPU (b) Memory

Fig. 9: CPU and Memory utilization under stress (Compute node)

does not need all the allocated memory. The CPU overload, on the other hand, was fully utilized during times of increased competition, demonstrating an increase of approximately 15%. Still, none of the tests exceeded the resource quotas established in the driver.

IV. Related Work

Some works have been developed based on the aspects evaluated during this research. These works can be classified into two categories according to their study objective: (i) Provisioning time evaluations; (ii) Optimization proposals. In the second category are techniques for provisioning time optimization related to caching mechanisms [12] [13] or read and write optimizations in computing nodes [14] [15]. These studies were concentrated in the transfer and imaging stage, proving that phase represents the major factor of latency during the provisioning of virtual machines. However, this research has more interest in the first category, in which evaluations are performed.

Another evaluation [16] compared the provisioning time in three public cloud providers: Amazon EC2, Azure and Rackspace. This work considered the following analysis variables: Provisioning Time, the authors investigated the relationship between the provisioning time and the time of day when virtual machines are requested; Image Size, the results of the research showed that this was the biggest impact factor in the provisioning time, presenting a linear degradation as the image size increased; Templates Resources, public providers offer different types of resource instances in the form of templates. Thus, the authors investigated the performance of provisioning in this regard; Location, in this test, there was a similarity between almost all regions, except for the case of Amazon, which demonstrated a greater time in the new regions established during the research; Concurrency, This experiment was performed with simultaneous provisioning, starting from one to 16 virtual machines in the Azure and EC2 providers.

Razavi etal. [17] made a scalability assessment in the cloud, but focused on the OpenNebula platform. In this research, the authors investigated the behavior of the platforms in their default configuration and suggested optimizations, making performance comparisons. The optimizations were applied on the following aspects: Scheduling, the authors used a load of 512 virtual machines and identified that the largest bottleneck point was in the scheduler configuration. Thus, they reduced the consumption scheduling interval queue from 30 to 1 second, increasing the number of machines provisioned in each interval to 1024; Caching, the authors note that OpenNebula, by default, does not cache shared images via the NFS protocol. As a workaround, the KVM disk-caching policy was enabled, causing the average provisioning time to be reduced by half. However, the variations between virtual machine provisioning increased, which was not later investigated; Parallelism, The amount of threads allocated to the provisioning drivers has been increased from 10 to 32; Code, The authors identified a number of optimization points in the platform code. The first, regarding the consumption of objects between the components. Thus, they changed the way of transferring requests with a horizontal thread service model. The second was the change the access way to host nodes by the platform. SSH communication over persistent TCP sockets was replaced. The authors also modified the method of requesting new virtual machines, previously serialized, for a parallel service.

In [18], there was an evaluation between the platforms Eucalyptus and OpenNebula with the public cloud provider Amazon EC2. Among the many evaluations, the authors investigated the provisioning time of 1,2,4,8 and 16 virtual machines using the same image. In this work, the provisioning time was investigated considering the following variations of OpenNebula image storage: NFS Eager, Lazy NFS, LocalDisk Eager and Lazy LocalDisk in comparison with Eucalyptus. Using only one virtual machine, the Eucalyptus platform presented the worst performance and the variation NFS Lazy was the best. From 2 to 4 virtual machines the NFS Lazy has begun to deteriorate, while LocalDisk and Eucalyptus have remained constant. The latest evaluation, over 4 virtual machines, showed the worst performance for OpenNebula with NFS Eager. In this scenario, there was also a degradation in the LocalDisk Eager and Eucalyptus times. In this case, the authors infer that the times were impacted by the disc writing contention exerted by the 4 host nodes used in the tests.

In [19], the authors evaluate a set of performance factors on virtual machines and containers, among them the system startup latency. In the paper there was no use of a cloud platform, but the direct provisioning on the KVM and LXC drivers. The research applied only the startup times of the two virtualization systems. In one of them the complete initialization of one virtual machine, in another, from a checkpoint restore. In both cases, a container on LXC achieved better performance. This was the only study that mentioned the use of library-based systems, although it did not make any kind of evaluation.

V. Conclusions

This paper presents an optimized architecture for provisioning OSv-based Unikernels in OpenStack. It was noticed that the solution presented satisfactory results, reducing the time of the entire deployment flow, from the compilation to the initialization of the Unikernels in the compute node. It was also noted that the implementation of memory isolation and CPU through control groups was successful, presenting overloading expected within the node.

As future work, we intend to extend the architecture with the following contributions:

- **Compilation**: Generalization of compilation methods, allowing that architectures written in other languages to be provisioned in the same way;
- **Abstraction layer**:A modular driver that allows not only OSv provisioning, but other Unikernels architectures for new evaluations;
- **Driver isolation**: Implementation of an independent Nova project dedicated to Unikernels provisioning and serving as a practical contribution to the OpenStack project.

References

[1] K. Morris. (2011) Configuration drift. [Online]. Available: http://kief.com/configuration-drift.html/

[2] C. Fowler. (2013) Trash your servers and burn your code: Immutable infrastructure and disposable components. [Online]. Available: http://chadfowler.com/blog/2013/06/23/immutable-deployments/

[3] M. Pais. (2014) Virtual panel on immutable infrastructure. [Online]. Available: http://www.infoq.com/articles/virtual-panel-immutable-infrastructure/

[4] S. Newman, *Building Microservices.* " O'Reilly Media, Inc.", 2015.

[5] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Communications of the ACM*, vol. 17, no. 7, pp. 412–421, Jul. 1974.

[6] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, *Unikernels: library operating systems for the cloud.* ACM, May 2013, vol. 41.

[7] A. Kivity, D. Laor, G. Costa, P. Enberg, and N. Har'El, "OSv—Optimizing the Operating System for Virtual Machines," *2014 USENIX Annual . . .*, 2014.

[8] J. Bonwick and B. Moore, "Zfs: The last word in file systems," 2007.

[9] Openstack heat. [Online]. Available: https://github.com/openstack/heat

[10] O. Tange *et al.*, "Gnu parallel-the command-line power tool," *The USENIX Magazine*, vol. 36, no. 1, pp. 42–47, 2011.

[11] D. Wieers. Dstat: Versatile resource statistics tool. [Online]. Available: http://dag.wiee.rs/home-made/dstat/

[12] P. De, M. Gupta, M. Soni, and A. Thatte, "Caching techniques for rapid provisioning of virtual servers in cloud environment," *Network Operations and Management Symposium (NOMS), 2012 IEEE*, pp. 562–565, 2012.

[13] K. Razavi and T. Kielmann, "Scalable virtual machine deployment using VM image caches," in *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis.* ACM Request Permissions, Nov. 2013.

[14] D. Zheng, H. Jin, X. Liao, and Y. Zhang, *Accelerating the Massive VMs Booting Up.* IEEE, 2014.

[15] Z. Zhang, Z. Li, K. Wu, D. Li, H. Li, Y. Peng, and X. Lu, "VMThunder: Fast Provisioning of Large-Scale Virtual Machine Clusters," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 25, no. 12, pp. 3328–3338, 2014.

[16] M. Mao and M. Humphrey, "A Performance Study on the VM Startup Time in the Cloud," in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on.* IEEE, 2012, pp. 423–430.

[17] K. Razavi, S. Costache, A. Gardiman, K. Verstoep, and T. Kielmann, "Scaling VM Deployment in an Open Source Cloud Stack," in *ScienceCloud '15: Proceedings of the 6th Workshop on Scientific Cloud Computing.* ACM Request Permissions, Jun. 2015, pp. 3–10.

[18] Y. Ueda and T. Nakatani, "Performance variations of two open-source cloud platforms," *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, pp. 1–10, 2010.

[19] K. Agarwal, B. Jain, and D. E. Porter, "Containing the Hype," *system.*