BOOSTING THE CLOUD META-OPERATING SYSTEM WITH HETEROGENEOUS KERNELS: A NOVEL APPROACH BASED ON CONTAINERS AND MICROSERVICES

R Subba Rao[1], Dr. Anil Kumar Mishra[2]
[1]*Computer Science & Engineering, Gandhi Engineering College (GEC, Bhubaneswar)*
[2]*Computer Science & Engineering, Gandhi Engineering College (GEC, Bhubaneswar)*

Abstract

A high SLA is a big challenge for the Cloud providers as they have to ensure the sustainability of their customers' workloads that depend closely on the underlying OS. The kernels are the cores of the OS, and the monolithic kernels are still the most performing despite of their fragility and unreliability. We think that the superposition of the kernels in such way that a healthy kernel replaces the vulnerable services of another kernel is a good track to operate. This replacement is accomplished via the transfer of system calls from the vulnerable kernel to a more reliable and efficient remote discovered kernel. We propose the architecture of a Meta-OS based on heterogeneous monolithic kernels in order to ensure reliability and performance. The features of this Meta-OS are encapsulated in microservices hosted on containers. Two technologies are used to implement our solution: Virtualization (hardware and OS based) and Web Services.

Keywords: Operating Systems, Monolithic Kernels, System calls, Mission critical and intensive applications, Virtualization, Docker, Containers, Web Services, Microservices**.**

1. INTRODUCTION

The Operating Systems provide services to run the applications. These services are invoked via system calls implemented into libraries and APIs. With the emergence of the Cloud, the service providers undertake on SLA levels too high to run the customers' workloads. This SLA can be closely linked to the quality of the OS that must be both reliable and efficient. The model without exceptions [1] for example has shown how the design of the OS can improve the performance of applications. The operating systems are based on the kernels which form their cores. Different approaches of design of the kernels are proposed. Nevertheless, the monolithic kernels like Linux kernels are the most powerful. Their fragility is due to the fact that the kernels include the majority of the critical features (to avoid the switches of context and gain performance). The slightest flaw in a module of the kernel may entail its interruption and the break of the entire operating system.

Today, hardware virtualization is required as a compelling solution for the consolidation of workloads and the sharing of physical resources. This technology is even supported natively in the hardware. Technologies like Intel-VT/AMD-V and SR-IOV can be cited as examples. The virtualization, in addition to the benefits of consolidation, offers by conception an ideal isolation of virtual machines. Each kernel of a given virtual machine runs in a secure and separate space independently of other kernels. Consequently, thanks to the virtualization, multiple heterogeneous systems can coexist on the same host.

Hardware virtualization can offer a solution to improve the reliability of the OS. This hypothesis is supported by VirtuOS [2] which defines architecture of an operating system subdivided functionally in service domains. Each service domain manages functionality, like storage, network, etc. and is executed in a virtual machine on the Xen hypervisor to offer a perfect isolation and security. As well, the system calls of the applications that run in the primary domain are routed to the adequate service domain. Thanks to this architecture a flaw that is registered at a service domain like a malfunction of a driver does not cause the break of the entire system. The restart of the failed domain does just affect partially the process waiting for replies from it. Other solutions implement the same priciple like the domain-0 disaggregation in the new generation of Xen Server called Windsor. Qubes is also an OS containing the same architecture as VirtuOS but it is more mature and even workable in practice.

On the other hand, Docker [6] is considered as a very promising OS virtualization technique that acts at the OS level by abstracting the process execution. Although the containerization is an old technology in the Unix/Linux systems, docker provides a very interesting layer of features like the containers migration from a version of the kernel to another version or the application of resources constraints on the containers, etc. via friendly tools.

The combination of the two virtualization technologies (hardware and OS) by running containers in virtual machines provides better isolation and reliability.

The purpose of our work is to extend the architecture of VirtuOS by making collaborate several heterogeneous kernels considered as domain services. We should apply the principle of the scale cube [5] by splitting the APIs according to the Y axis in small APIs executed in containers as microservices [6] or simply as processes hosting Web Services. These small APIs will be executed on lightweight virtual machines (domain services) based on heterogeneous kernels.

This heterogeneity may engenderer a Meta-Operating System more reliable and efficient in bringing the benefits of heterogeneous kernels. The Meta-Operating System can be seen as an overlay of kernels where the kernel X offers the services of the network via its drivers as well as the implementation of the TCP stack while the kernel Y offers the service of storage for access to files for example.

Our Meta-Opertaing System's kernels could be virtual machines on the same host but we intend to extend this feature to discover other machines and domain services in the network. The Web Services as the implementation technology of microservices are a good candidate to respond to this need through standards like WS-Discovery.

The defined objetcives in our work are the following:

- Define the architecture of a Meta-Operating System based on several domain services hosting heterogeneous kernels in the form of virtual machines co-existing on the same host or discovered in the network.
- Define the architecture of an API subdivided in small domain services APIs. These small APIs will be developed as microservices and executed within containers or as Web Services hosted into process. The system calls routing will be abstracted by remote calls of methods of Web Services.
- Define a proactive supervision mecanisme so that the Meta-Operating System isolates the faulted domain services prior to prevent any performance degradation and increase reliability. This same mechanism would be able to migrate any microservice during the execution from a suspected service domain to another domain more reliable even on the basis of kernels of different versions.
- Define an intelligent mecanism for dispatching the remote system calls towards local or remote service domains taking into account two parameters: Reliability and Performance.

The rest of the paper is organized as follows:

- Part 2 : Meta-Operating System architecture design.
- Part 3 : Experimental tests results.
- Part 4 : Quick related work presentation.
- Part 5 : Conclusion.

2. THE META OPERATING SYSTEM BASED ON HETEROGENOUS KERNELS

VirtuOs [2] and WSNFS [8] are two different solutions that have in common the exetrnalization of the execution of system calls with a possibility to reuse non existing functionalities on the original environnement that are offered by the remote domains.

VirtuOS externalizes the system calls towards the service domains to improve the whole system reliability. Whereas, WSNFS aims to abstract heterogenous file systems by developping a single driver using Web Services. This will also allow to the solution to be more flexible and reliable when adding more WSNFS gateways and drivers.

Both solutions are the basis of the Meta-Operating System whose kernel consists of several heterogeneous kernels co-existing on a hypervisor or remote kernels discovered in the network. The Meta-OS caracteristics are as follows:

- The Meta-OS is composed of varied and heterogenous service domains. Every domain implements a feature or set of features based on a given kernel.
- The offered features are exposed via entry points that could be invoked using modified APIs.
- Automatic discovery mechanism of new service domains or new features in existing domains.
- Intelligent system calls routing from the domain where the users' processes are running towards remote service domains, taking into account two parameters: reliability and performance.
- A proactive monitoring mechanism so that the Meta-OS isolates the faulty domain services before recording a performance degradation event.
- To reuse remote service domains' functionalities, an intelligent API is designed to dispatch system calls to the adequate domain service. This API has to abstract the dispatching system end ensure the existing applications portability.

Consequently, the Meta-OS can be seen as an overlay of multiple kernels.
The Meta-OS is based on two categories of domains:

- Service domains: they aim to provide services to applications and other domains. These domains can be instanciated, started, stopped, paused, cloned and migrated upon request by the Meta-OS sybsystem.
- Applications domains: they aim to provide the execution runtime for the users' applications. They can be considered as customer domains consuming services from service domains but they can play the role of service domains.

A. Principle of system calls' externalization

Monolithic applications present some drawbacks like debugging difficulties and the Single Point of Failure design. Consequently, some patterns are invented to divide monolithic applications into small manageable modules. The microservices [6] are one of the paradigms that aim to achieve these objectives. The microservices are the small modules that collaborate and communicate using standard protocols like HTTP/Rest.

Each microservice is developed and deployed independently. Applying some principles like the Single Responsibility Principle, changes made to a microservice don't affect the integrity of other microservices.

In our case, the APIs can apply the microservices architecture, where each API is composed of two parts (Fig. 1):

- Forontend API: this part encapsulates the system calls invocation routines to local and remote service domains and a load balancer engine (LB) that aims to dispatch syscalls to backend APIs. As these latteres are executed into Web Services, the Frontend API implements proxies to invoke them. The Frontend API aims also by abstracting the backend routines to ensure the portability of existing applications.
- Backend APIs: each backend API implements a subset of functionalities inside a service domain. They will be executed into Web Services hosted on processes or inside containers. Different protocols can be used to invoke the frontend APIs like TCP, HTTP, REST, etc.

In the following figure, we show an example of a system call externalization principle.

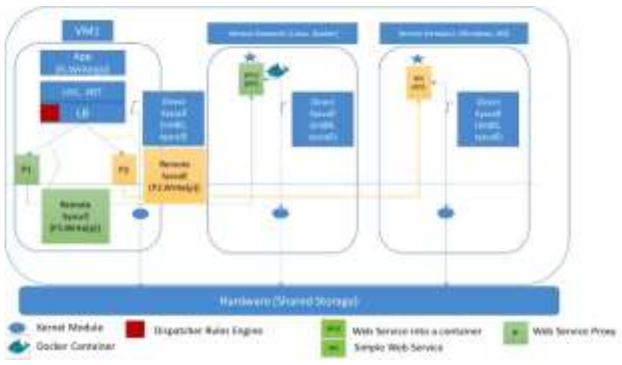
VM1
App
LB
P1
P2
Remote Syscall (P1.Write())
Remote Syscall (P2.Write())
Hardware (Shared Storage)
Kernel Module
Docker Container
Dispatcher Rules Engine
Web Service into a container
Simple Web Service
Web Service Proxy

Fig. 1. Write method invocation styles in the Meta-OS

Calling the **Write** method on the object **fs** (FileStream type) can generate the routine of the system call directly on the local host, an invocation of the backend API as a Web Service hosted in a container via the proxy (P1.Write) or finally an invocation of the backend API as a Web Service hosted in a process via the proxy (P2.Write).

B. The Meta-OS architecture

The following figure presents the architecture of the Meta-OS grouping two scenarios:

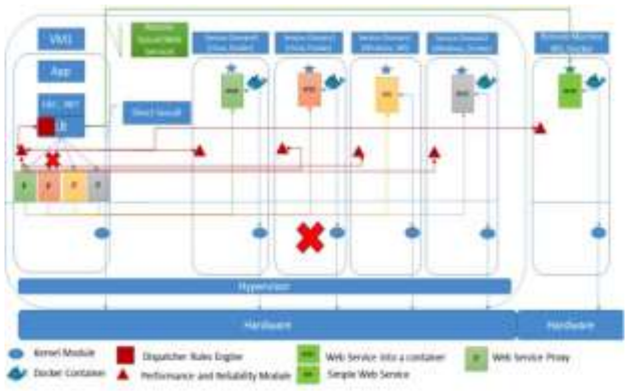

Fig. 2. The Meta-OS architecture

1) ***Scenario1: co-existed domains on the same host*** This sceanrio is based on the architecture of VirtuOS. Thanks to advanced hardware based virtualization technologies, the domain services can own a part of hardware resources like in SR-IOV. The hypervisor ensures critical tasks like memory management, tasks scheduling, etc. Each domain integrates a performance and reliability module to instrument the syscalls without causing any overhead. Some tools could be used to achieve this goal like Sysdig. The performance module ensures also the discovery of other domains performance modules, consolidating all the data and mataining references to the discovered service domains.

Unlike VirtuOS, the syscalls dispatching system is not accomplished in kernel mode but is abstracted by the remote Backend APIs invocation in user mode. We are aware of the performance degradation but this can be improved using the Hypervisor bus as communication medium between the the co-hosted domains.

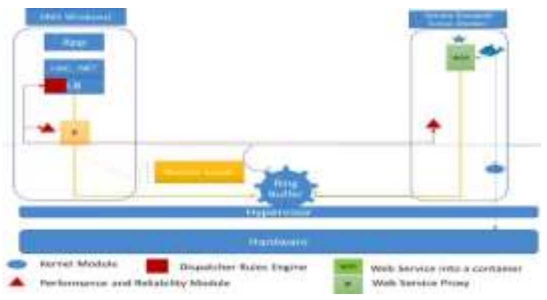

Fig. 3 Syscalls processing via the hypervisor bus channel.

2) Scenario2: discovered remote domains in the network This scenario is based on the architecture of WSNFS. The processing of the syscalls is externalized via the remote Web Services executed into containers or hosted directely by processes.

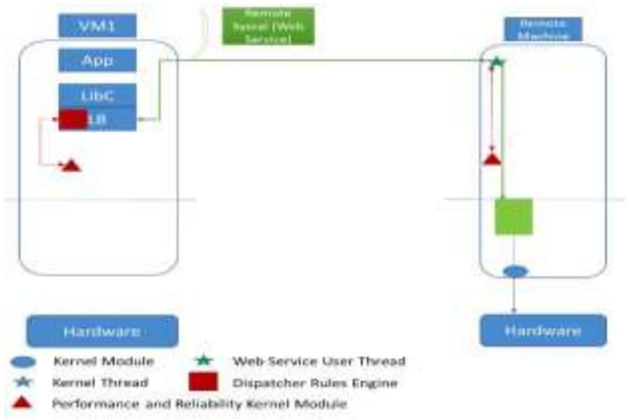

Fig. 4. Syscalls processing

C. The intelligent API

The frontend API is so crucial in the process of syscalls invocation. It implements a business rules engine that helps in selecting the best available service domain for the current therad and transforming if needed the generated syscall routine to be supported by the selected domain. To simplify the processing of future syscalls for a given thread, an affinity is maintained through a table that associates a thread to its corresponding service domains.

Three domains detinations types are defined:

- Local handler
- Remote handler on a virtual machine via the shared bus.
- Remote handler on a virtual machine via the network.

Selecting the most suitable service domain is based on the information collected by the performance module. This latter maintains with other modules a data structure containing entries corresponding to service domains. Each entry includes the domain, the corresponding performance measure done on a subset of syscalls, the corresponding SLA that measures the availability importance of the domain and the corresponding cost that measures the cost of invoking the domain. All these data is exchanged between all performance modules to have an accurate and unique view of the state of domains. The performance measure is obtained by generating a subset of syscalls that are not resource consuming like open(2) and close(2) on a sample file for the storage domains for example.

The following formula (to be defined) attributes a score to each service domain :

Domaine X Score = Formula (Perf, SLA, Cost).

The following figure defines the syscalls routines dispatching system based on the performance data structure (Performance Mapping Table) for the send() syscall routine in order to select the best netword domain .

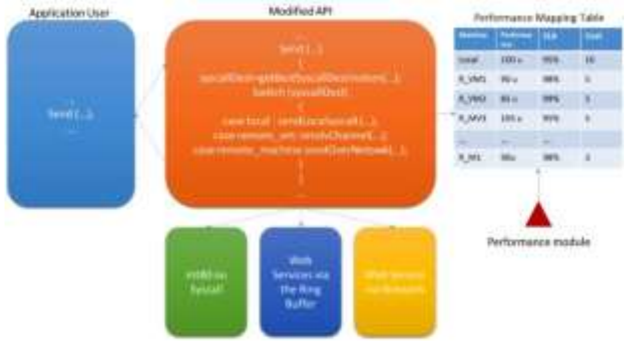

Fig. 5. Syscalls routines dispatching system based on business rules

D. Towards a more intelligent Operating System

The meta-OS is able to perform the following tasks in order to offer better quality of services by exploiting the Performance Mapping Table:

- Collect and interpret vulnerabilities reports of various services on the Internet. This will assign lower marks and disposal decisions for kernels that may be hazardous in the future. This technique improves the proactive nature of the actions.
- Stop the domain service that cause vulnerabilities or technical problems. This will effectively isolate these domains and release the resources consumed.
- Update the configuration of the requested domain services by increasing corresponding resources (RAM, CPU, disk IOs, etc.).
- Clone the service domains with excellent scores especially in the presence of resources.

Using advanced checkpoint/restore technologies, it is possible to proactively migrate microsevices from a suspect kernel to another version of the kernel without any interruption of running applications.

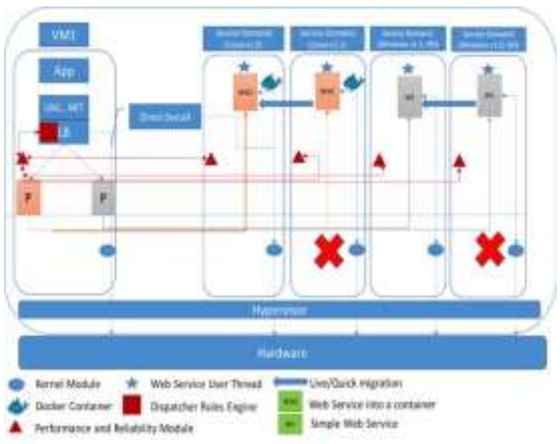

Fig. 6. Microsoervices migration principle

3. RESULTS

In what follows, we present some results related to tests on the architecture of microservices APIs on heterogeneous domains. We developed a POC based on a frontend API and related backend APIs dealing with the storage domain. The backend APIs are developed as WCF (Windows Communication Foundation) Web Services that run either in processes or in Docker containers on Windows and Linux. Our tests were performed on virtual machines hosted on the Azure Cloud platform of Microsoft.

Here are the configurations of test machines:

- *Azure Virtual Machines (Cloud) :*
- *Windows Server 2016 CTP 4 Core with Docker, Type: Standard_D1 : RAM: 3.5 GB, 2vCPU (Physical CPU: Intel Xeon® E5-2673 v3 (Haswell) processor, 2.4 GHz)*
- *Linux Ubuntu Desktop 15.04 with Docker, Type : Standard_D1 : RAM: 3.5 GB, 2vCPU (Physical CPU: Intel Xeon® E5-2673 v3 (Haswell) processor, 2.4 GHz)*

Tools and Frameworks:

- *.NET 4.5 to develop frontend and backend storage APIs on Windows*
- *Mono 3.2.8 et Mono 4.2.1 5 to develop frontend and backend storage APIs on Linux (NET Core is a Microsoft initiative for a cross-platform .NET framework. As this framework presents some limitations likein WCF, it was replaced by Mono).*
- *Docker version 1.10.0-dev, build 18c9fe0 on Windows*
- *Docker version 1.10.0-dev, build e39c811,*
- *experimental on Linux (https://github.com/boucher/docker)*

Our POC was limited to the sequential read and write operations by blocks or characters on different sizes of files and using vried cache sizes. The different service domains (virtual machines) access to shared files stored on a CIFS share. The sizes of files vary generally as follows: 1MB, 10MB, 50 MB and 100MB. The cache sizes are: 512B, 1KB, 4KB. We measure in each case the Throughput (MB / s), the channel communication time (ms) and the service side (backend API) in (ms). The total running application time is the sum of the channel communication time and the service side time. The measures are averages of 10 tests on the same case. In order to validate these averages, we calculate the standard deviation. The results are homogeneous in general with a standard deviation not exceeding 3 units.

In this paper we present results realted to two scenarios:

1) Scenario1: Backend APIs on service domains (virtual machines) on Azure

In this scenario we compare performance data between backend APIs running in different environments:

- Web Services on Windows Server 2016 CTP4 based on .NET 4.5
- Web Services on Windows Server 2016 CTP4 containers based on .NET 4.5
- Web Services on Linux Ubuntu 15.04 based on Mono 3.2.8
- Web Services on Linux Ubuntu 15.10 based on Mono 4.2.1
- Web Services on Linux containers based on Mono 4.2.1

Interpretation of Results and Comments We can raise the following points:

- Backend APIs on Linux APIs recorde better rates especially in the case of operations by character.
- The containers based microservices recorde a minimal cost compared to Web services that run directly on virtual machines.

2) Scenario 2: Migration of microservices

To ensure high availability of microservices that run as Web services directly in the service domains or in containers, we used the CRIU tool for capturing and restoring microservices from a Ubuntu 15.04 service domain to a Ubuntu 15.10 service domain. To estimate the necessary times to the abovementioned operations, we measured the duration of capture and recovery of microservices dealing with files cerated sequentially by character with various sizes. The tests were carried out on a local host and an Azure host (Cloud). Three measurements are obtained:

- User Time: time required to capture or restore in user mode.
- Kernel Time: time required to capture or restore in Kernel Mode.
- Real Time: the whole time required to capture or restore taking into account interruptions.

4. RELATED WORK

The design of reliable operating systems has been the subject of long debates. Systems based on microkernels as L4 have proven a high reliability level by isolating many OS services effectively in user processes but require a large IPC communications optimization work to improve performance. Some improvements using advanced techniques like Aspect oriented programming [9] were applied. Tesselation [10] and fos [11] are solutions that use microkernels techniques where OS serveices are factored and implemented in User space.

Operating System decomposition has also knew different solutions [12] in addition to microkernels.

System calls can cause serious performance problems. FlexSC [1] is an excellent solution to invoke system calls asynchronously providing more performance.

On the other hand, code reuse has been the subject of several research projects. Levasseur et al. [13] proposed a drivers virtualization solution where drivers run inside processes (virtual machines) in the user space. The virtual machines run on a L4 micro kernel.

Operating systems based on several kernel instances have also emerged to exploit the physical resources. Barrelfish [14] for example is an OS that aims to run multiple instances of the same kernel on multiple types of processors (CPU, GPU, DSP, etc.).

5. CONCLUSION

In this paper we tried to present the architecture of a Meta-Operating System based on heteregenous domain services discovered in the network or running on the same host.

The aim of this Meta-Operating System is to reap the benefits of all the discovered service domains by developing a technique of dispatching syscalls routines towards the most adequate service domain. This technique is leveraged by decomposing the APIs into frontend APIs running in the original applications domain and backend APIs that expose syscall routines services.

To underpin our approach, we developed a prototype by extending the classic API file processing. Backend APIs as microservices run directly in storage domain services or in containers.

We measured performance data in different situations and obtained very promising results that could be optimized using advanced techniques like GPGPU [15] to accelerate messages processing, Exceptioneless syscalls to eliminate the switching mode overhead, etc.

References

[1] Soares, L., & Stumm, M. (2011, June). Exception-Less System Calls for Event-Driven Servers. In USENIX Annual Technical Conference (Vol. 10).

[2] Nikolaev, R., & Back, G. (2013, November). VirtuOS: an operating system with kernel virtualization. In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (pp. 116-132). ACM.

[3] Rutkowska, J., & Wojtczuk, R. (2010). Qubes OS architecture. Invisible Things Lab Tech Rep, 54.

[4] Merkel, D. (2014). Docker: lightweight linux containers for consistent development and deployment. Linux Journal, 2014(239), 2.

[5] Martin L. A. & Michael T. F. (2015). Art of Scalability, The: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise, 2nd Edition. “Addison-Wesley Professional, Inc.”.

[6] Newman, S. (2015). Building Microservices. " O'Reilly Media, Inc.".

[7] Abraham S., Peter B.G. & Greg G.(2013). Operating System Concepts, Ninth Edition. “John Wiley & Sons, Inc.”.

[8] Hwang, G. H., Yu, C. H., Sy, C. C., & Chang, C. Y. (2008). WSNFS: A Web-Services-Based Network File System. J. Inf. Sci. Eng., 24(3), 933-947.

[9] Borchert, C., & Spinczyk, O. (2016). Hardening an L4 microkernel against soft errors by aspect-oriented programming and whole-program analysis. ACM SIGOPS Operating Systems Review, 49(2), 37-43.

[10] Colmenares, J. A., Eads, G., Hofmeyr, S., Bird, S., Moretó, M., Chou, D., ... & Asanović, K. (2013, May). Tessellation: refactoring the OS around explicit resource containers with continuous adaptation. In Proceedings of the 50th Annual Design Automation Conference (p. 76). ACM..

[11] Wentzlaff, D., & Agarwal, A. (2009). Factored operating systems (fos): the case for a scalable operating system for multicores. ACM SIGOPS Operating Systems Review, 43(2), 76-85.

[12] Jacobsen, C., Khole, M., Spall, S., Bauer, S., & Burtsev, A. (2016). Lightweight capability domains: towards decomposing the Linux kernel. ACM SIGOPS Operating Systems Review, 49(2), 44-50.

[13] LeVasseur, J., Uhlig, V., Stoess, J., & Götz, S. (2004, December). Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In OSDI (Vol. 4, No. 19, pp. 17-30).

[14] Schüpbach, A., Peter, S., Baumann, A., Roscoe, T., Barham, P., Harris, T., & Isaacs, R. (2008, June). Embracing diversity in the Barrelfish manycore operating system. In Proceedings of the Workshop on Managed Many-Core Systems (p. 27).

[15] Jordan, V. (2010). XML query processing using GPGPU (Doctoral dissertation, Master’s thesis, University of Tsukuba & Université de technologie de Belfort-Montbéliard).