Facultad de Ciencias

ANÁLISIS DE RENDIMIENTO DE UNIKERNELS EN ENTORNOS DE COMPUTACIÓN DISTRIBUIDA EN EL ÁMBITO CIENTÍFICO
(Performance analysis of unikernels in distributed computing environment for the scientific field)

Trabajo de Fin de Grado para acceder al
GRADO EN INGENIERÍA INFORMÁTICA

Autora: Carmen Marcos Sánchez de la Blanca
Director: Enrique Vallejo Gutiérrez
Co-Director: Álvaro López García
Junio - 2022

Índice general

1. Introducción **1**
- 1.1. Motivación del trabajo . . . 1
- 1.2. Objetivo . . . 2
- 1.3. Estructura del documento . . . 2

2. Conocimientos previos y definiciones **3**
- 2.1. Virtualización . . . 3
  - 2.1.1. Definición . . . 3
- 2.2. Máquina Virtual . . . 3
  - 2.2.1. Definición . . . 3
  - 2.2.2. Origen y motivación de uso . . . 3
  - 2.2.3. Tipos de MV y su arquitectura . . . 4
  - 2.2.4. Uso actual en entornos de computación científica . . . 4
- 2.3. Hipervisor . . . 5
  - 2.3.1. Definición . . . 5
  - 2.3.2. Origen y motivación de uso . . . 5
  - 2.3.3. Tipos de hipervisores y su arquitectura . . . 6
  - 2.3.4. Uso actual en entornos de computación científica . . . 6
- 2.4. Contenedor . . . 8
  - 2.4.1. Definición . . . 8
  - 2.4.2. Arquitectura . . . 8
  - 2.4.3. Origen y motivación de uso . . . 9
  - 2.4.4. Uso actual en entornos de computación científica . . . 9
- 2.5. Unikernel . . . 10
  - 2.5.1. Definición . . . 10
  - 2.5.2. Origen y motivación de uso . . . 10
  - 2.5.3. Tipos de Unikernels y su arquitectura . . . 11
  - 2.5.4. Uso actual en entornos de computación científica . . . 12
  - 2.5.5. Diferencias entre unikernel, microkernel y exokernel . . . 13
- 2.6. Comparación entre MVs, Contenedores y Unikernels . . . 14
- 2.7. OpenStack . . . 15
  - 2.7.1. Definición y concepto . . . 15
  - 2.7.2. Arquitectura y componentes . . . 15
  - 2.7.3. Uso actual en entornos de computación científica . . . 16

3. Análisis de frameworks de unikernels **17**
- 3.1. Enumeración de unikernels existentes hoy en día . . . 17
  - 3.1.1. MirageOS . . . 17
  - 3.1.2. HermitCore . . . 17
  - 3.1.3. IncludeOS . . . 18
  - 3.1.4. ToroKernel . . . 18

3.1.5. OSv . . . 19
3.1.6. OPS/Nanos . . . 19
3.2. Comparativa de los unikernels existentes . . . 20
3.3. Análisis . . . 20
3.3.1. OSv vs OPS/Nanos . . . 21
3.4. Unikernels seleccionados . . . 22
3.4.1. MirageOS: Especificaciones técnicas y modo de creación . . . 23
3.4.2. Ops/Nanos: . . . 25
3.4.3. Creación de unikernel OPS/Nanos de prueba en local . . . 26
3.4.4. Implementación de unikernels de prueba en el CPD del IFCA . . 27

4. Implementación unikernel especializado (CPD IFCA) 31
4.1. Elección de aplicación científica: OpenQCD . . . 31
4.1.1. Elección del benchmark . . . 32
4.1.2. Creación del unikernel con aplicación en un entorno de computación científica e implementación en el cloud . . . 32
4.1.3. Problemas encontrados en la ejecución . . . 33
4.2. Elección de aplicación científica: Benchmark 435.Gromacs . . . 33
4.2.1. Elección del benchmark . . . 34
4.2.2. Creación del unikernel con aplicación en un entorno de computación científica e implementación en el cloud . . . 34
4.3. Creación de los contenedores Docker con las aplicaciones científicas . . . 34

5. Resultados obtenidos y análisis de rendimiento 35
5.1. Aplicación científica OpenQCD . . . 35
5.1.1. Ejecución con tamaños de matriz original, 64 elementos . . . 35
5.1.2. Ejecución con tamaños de matriz de 4096 elementos . . . 37
5.1.3. Comparación de las ejecuciones con los dos tamaños seleccionados 39
5.2. Aplicación científica Gromacs Benchmark . . . 39
5.2.1. Ejecución del benchmark . . . 39

6. Conclusiones obtenidas 41
6.1. Respuesta a las preguntas de investigación . . . 41
6.2. Conclusiones . . . 42
6.3. Posibles futuras líneas de trabajo e investigación . . . 42

A. Anexos I
A.1. Glosario . . . I
A.2. MirageOS y OCaml . . . II
A.2.1. Comandos de instalación y configuración de Opam y MirageOS . II
A.2.2. Librería Lwt threads . . . III
A.3. OPS y Nanos . . . IV
A.3.1. Comandos de instalación y configuración de OPS . . . IV
A.4. Dockerfile para la creación de la imagen Docker . . . V

Índice de figuras

2.1. Figura adaptada de [1]. Arquitectura típica de una Máquina Virtual. . . . 4
2.2. Figura adaptada de [2]. Arquitectura del hipervisor Xen. 7
2.3. Arquitectura del hipervisor KVM. Figura adaptada de [3]. 8
2.4. Figura adaptada de [4]. Arquitectura de un unikernel en comparación con una máquina virtual. 12
2.5. Estructura simplificada de un Microkernel [5] 13
2.6. Estructura simplificada de un Exokernel [6] 14

3.1. Arquitectura simplificada del unikernel HermitCore [7]. 18
3.2. Imagen de prueba OPS subida al cloud 28
3.3. Creación de la instancia de prueba en el cloud 28
3.4. Creación de la instancia de prueba en el cloud 29

5.1. Tabla con los tiempos de ejecución, en ns, obtenidos con las matrices originales de 64 elementos. 35
5.2. Gráfica tiempos de ejecución de operaciones de time3 con tamaño de matriz de 64 elementos 36
5.3. Tabla con los tiempos de ejecución, en ns, obtenidos con matrices de 4096 elementos 37
5.4. Gráfica tiempos de ejecución con tamaño de matriz de 4096 elementos . . 37
5.5. Gráfica tiempos de ejecución de operaciones de time3 con tamaño de matriz 4096 38
5.6. Gráfica tiempos de ejecución benchmark GROMACS 40

Resumen

En este Trabajo Fin de Grado se ha realizado el análisis de rendimiento de unikernels con distintas funcionalidades de aplicaciones científicas. Para la mejor comprensión del documento, se inicia realizando una introducción a los conceptos claves, así como la motivación de uso de distintas tecnologías involucradas y su uso actual en los entornos de computación científica.

También se realiza un análisis de los frameworks de unikernels más utilizados hoy en día, con el objetivo de elegir el más adecuado para el objetivo de este Trabajo Fin de Grado. Con el framework seleccionado, se realizan tests iniciales de implementación para comprobar la viabilidad del posterior desarrollo de los unikernels a evaluar.

Tras la implementación de dos aplicaciones científicas en el framework seleccionado, se realiza un análisis de rendimiento. Este desempeño es comparado con el conseguido mediante la implementación de las mismas aplicaciones en otros dos entornos: máquinas virtuales y contenedores Docker. El análisis se realiza con el fin de evaluar si es viable la implementación de unikernels en entornos de HPC y computación científica, como en el caso del IFCA (Instituto de Física de Cantabria).

Finalmente, tras el análisis se ha concluido que la implementación de dichas aplicaciones científicas en unikernels, no supone una mejora de rendimiento frente a la ejecución de las mismas aplicaciones sobre contenedores y máquinas virtuales. Sin embargo, a día de hoy, si que es una alternativa factible a dichas tecnologías en caso de que no se necesite hacer uso de MPI en dichas aplicaciones. Esto se debe a que aportan una mayor facilidad de uso, seguridad y ahorro de recursos en las infraestructuras cloud.

Palabras clave: Sistemas operativos, entornos de computación científica, virtualización, unikernel, contenedores, evaluación de rendimiento.

Abstract

In this Final Degree Project, the performance analysis of unikernels with different functionalities of scientific applications has been carried out. For a better understanding of the document, we begin with an introduction to the key concepts, as well as the motivation for the use of different technologies involved and their current use in scientific computing environments.

An analysis of the most used unikernel frameworks nowadays is also carried out, with the aim of choosing the most suitable one for the objective of this Final Degree Project. With the selected framework, initial implementation tests are carried out to check the viability of the subsequent development of the unikernels to be evaluated.

After the implementation of two scientific applications in the selected framework, a performance analysis is performed. This performance will be compared with that achieved by implementing the same applications in two other environments: virtual machines and Docker containers. The analysis is performed in order to evaluate whether it is feasible to implement unikernels in HPC and scientific computing environments, as in the case of IFCA (Institute of Physics of Cantabria).

Finally, after the analysis it has been concluded that the implementation of these scientific applications in unikernels, does not represent a performance improvement compared to the execution of the same applications on containers and virtual machines. However, today, it is a feasible alternative to these technologies if MPI it's not needed to run the applications. This is because they provide greater ease of use, security and resource savings in cloud infrastructures.

Keywords: Operative systems, scientific computing environments, virtualization, unikernel, containers, performance benchmark.

Capítulo 1

Introducción

1.1. Motivación del trabajo

Actualmente, cuando se habla de virtualización en entornos de computación científica, los términos que asociamos a este ámbito de la computación, son: máquina virtual, contenedores, e incluso, Docker. Sin embargo, la vuelta al paradigma de los kernels con un único espacio de direcciones puede cambiar esta realidad en los próximos años, añadiendo en nuevo término a nuestro mapa mental: Unikernel.

Como es ampliamente sabido en el ámbito de la informática y la ciencia de datos, los contenedores es una tecnología ampliamente usada en los CPD's para la virtualización de aplicaciones y aislamiento de las mismas. Estos datos se ven reflejados en una encuesta realizada por una de las multinacionales proveedoras de código abierto más importante del mundo: Red Hat. Dicha encuesta [8], realizada en Junio del 2021 con el fin de evaluar y comprender como el mercado responde a los contenedores, arrojó sorprendentes resultados: el 91 % de las empresas encuestadas (empresas clientes de Red Hat) afirmaron que actualmente estaban desarrollando e integrando aplicaciones basadas en contenedores. Con estos datos, podemos llegar a comprender la buena respuesta que el mercado ha dado a esta tecnología desde su aparición en el 2013 [8].

Sin embargo, durante los próximos años, los contenedores podrían ver reducido su uso en algunos ámbitos debido a la aparición de los unikernels. Lo que hoy conocemos como unikernel proviene de un concepto acuñado en los años sesenta cuando empezaron a aparecer los primeros sistemas operativos: Sistemas Operativos de Biblioteca [9]. Estos primeros sistemas operativos tenían un único espacio de direcciones, correspondiente a la memoria física del computador, donde se cargaba el programa que se tenía que ejecutar con las librerías necesarias para su ejecución. No obstante, no era una implementación muy eficaz, pues mucho espacio de memoria se quedaba inutilizado y no se podían ejecutar procesos de manera concurrente, solo el que estaba cargado en memoria física. Este tipo de sistemas operativos fueron rápidamente remplazados por sistemas operativos capaces de tener cargados en memoria más de un proceso y realizar compartición de tiempo entre ellos para mejorar la eficiencia y la utilización efectiva de la CPU. En 2013, Anil Madhavapeddy y David J. Scott publicaron un artículo científico [10] en el que proponen traer de vuelta este tipo de sistema operativo pero con un propósito bastante concreto: implementarlo en el mundo del cloud computing y la virtualización, dándole el nombre de Unikernel.

Pese a que, los contenedores ofrecen grandes ventajas en el mundo de la virtualización, también tienen sus inconvenientes. Aunque son mucho más ligeros que las máquinas vir-

tuales, presentan un mayor overhead que el de los unikernels debido a la arquitectura de los mismos. Mientras que los unikernels solo necesitan tres capas (hardware, hipervisor y el unikernel), los contenedores tienen una arquitectura más "compleja": hardware, sistema operativo host (linux o macOS), hipervisor y un guest Linux dónde se alojan los contenedores. Estos últimos, incorporan los programas, librerías y el software que se desea ejecutar.

1.2. Objetivo

El objetivo de este Trabajo Fin de Grado es dar respuesta a la siguiente pregunta de investigación:

¿Es posible, a corto o medio plazo, reemplazar los contenedores por los unikernels en un entorno de computación distribuida en un entorno científico?

Para lograr este objetivo se intentará responder a las siguientes subpreguntas mediante un análisis de rendimiento de varios frameworks de unikernels de aplicación en el ámbito científico, en comparación con los resultados obtenidos mediante la tecnología de contenedores Docker :

- ¿Es más fácil desplegar una aplicación científica en un CPD mediante un unikernel frente a un contenedor como pueden ser los desarrollados por Docker?
- ¿Mediante el uso de los unikernels se consigue alguna ganancia a nivel de rendimiento (tiempo de boot, tiempo de ejecución, uso de memoria, etc.)?

1.3. Estructura del documento

En este documento se hace una breve introducción a los términos necesarios para entender el TFG. Asimismo, se hace una comparación y análisis de los frameworks de unikernels más empleados hoy en día con el fin de elegir los más apropiados para implementarlos en este trabajo. Para ello se evalúan dichos frameworks en base a distintos parámetros como es el tipo de unikernel, los hipervisores sobre los que es capaz de ejecutar, entorno de uso, lenguaje en que se puede desarrollar y facilidad de uso.

Tras la comparación teórica, se realiza una prueba de los frameworks elegidos para evaluar la viabilidad de implementación y facilidad de uso de estos. A continuación, se implementan los unikernels con una aplicación propia de computación distribuida en el ámbito científico sobre los que se ejecutarán unos benchmarks. Además, se realiza el mismo benchmark sobre dicha aplicación ejecutada en contenedores. Finalmente, se procede al análisis correspondiente en función de los resultados obtenidos y se recogen las conclusiones finales.

Capítulo 2

Conocimientos previos y definiciones

A lo largo de este trabajo se hará referencia a estos términos de un modo bastante exhaustivo y con terminología propia, por lo que he realizado esta sección para introducir los conceptos básicos para la correcta comprensión de este Trabajo de Fin de Grado. Estos conceptos son: **Virtualización, máquina virtual, hipervisor, contenedor y unikernel**.

2.1. Virtualización

2.1.1. Definición

La virtualización es el mecanismo mediante el que se permite a múltiples máquinas virtuales ejecutar en un cluster de máquinas físicas compartido. Cada máquina virtual tiene un computador, que arranca como un kernel del sistema operativo normal [11].

2.2. Máquina Virtual

2.2.1. Definición

Una máquina virtual es un entorno de computación completo con sus recursos totalmente aislados del resto de máquinas virtuales y sistemas que comparten el mismo dispositivo o sistema físico. A efectos prácticos podemos denominar una máquina virtual como una capa de software que nos permite ejecutar otro sistema operativo sobre el SO del sistema nativo [12].

2.2.2. Origen y motivación de uso

Antes de que naciese la virtualización, los científicos tenían que acceder a los servidores físicos para poder lanzar sus datos a computar. Esto suponía que solo un usuario podía usar ese servidor físico durante un determinado tiempo, resultando en una gran pérdida de rendimiento. En los últimos años previos al nacimiento de la virtualización, la mayoría de los recursos estaban inutilizados, pues gran parte de los jobs no eran altamente demandantes de CPU o memoria [13].

La primera tecnología creada que se asemeja al concepto de máquina virtual, tal y como lo conocemos hoy en día, se remonta al principio de la decada de los 60. En el proyecto M44/44X, IBM desarrolló un método para particionar una máquina IBM, en concreto una IBM 7044, en submáquinas. Estas submáquinas eran imágenes de la máquina 7044 pero con menos memoria. En este proyecto se realizó la primera referencia al término máquina virtual para referirse a las imágenes de la máquina 7044. A estas máquinas virtuales se las denominó las 44X [14].

Más adelante, el equipo de IBM que desarrolló el CP-40 reusó el término de máquina virtual para referirse a lo que previamente se llamaba pseudo-máquina Con el paso del tiempo, las tecnologías han seguido evolucionando, más allá del CP-40, y con ellas la acepción que se le da al término de máquina virtual, hasta llegar al concepto actual.

2.2.3. Tipos de MV y su arquitectura

Las máquinas virtuales se pueden clasificar en 2 grupos, en función del nivel de abstracción que proporcione: **Máquinas virtuales de sistema y las máquinas virtuales de proceso**. Estas últimas también se las denomina máquinas virtuales ligeras.

- **Máquinas virtuales de sistema:** El entorno de computación generado emula por completo a un sistema completo, aislado del resto de máquinas virtuales que puedan estar alojadas en el sistema físico nativo. Cada máquina virtual de sistema ejecuta su propia instancia del sistema operativo de modo independiente.
- **Máquinas virtuales de proceso:** La virtualización en este tipo de MV solo emula parte de un computador. Puede ser un proceso, una aplicación o un computador con el conjunto de instrucciones y servicios limitados que el usuario requiera.

Como se muestra en la figura 2.1, la máquina virtual necesita de una capa hardware por debajo, así como un sistema operativo anfitrión. Es decir, un sistema operativo base. Además, otra pieza software llamada hipervisor será la encargada de asignar recursos y gestionarlos para asegurar el correcto funcionamiento de la máquina virtual.

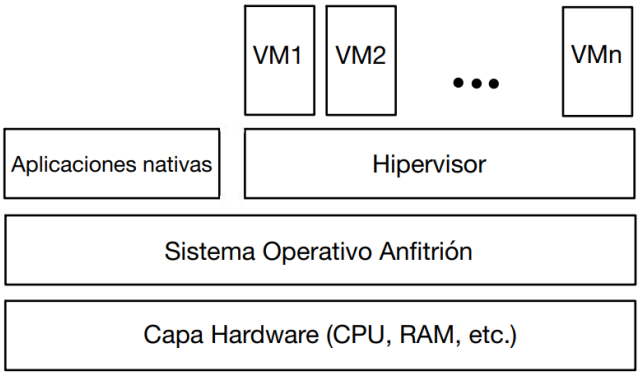

Figura 2.1: Figura adaptada de [1]. Arquitectura típica de una Máquina Virtual.

2.2.4. Uso actual en entornos de computación científica

Los beneficios que nos aporta esta tecnología han hecho que en los entornos de computación científica y en los CPD's se utilicen máquinas virtuales a diario, siendo la tecnología que proporciona los pilares para la computación en cloud. El aislamiento entre MV, la seguridad y flexibilidad que proporciona, y la portabilidad son solo algunos de los factores por los que su uso está tan extendido en la computación científica. En su día era uno de los pocos modos de aprovechar todos los recursos que nos proporciona una máquina física de

altas prestaciones, como las que se encuentran en los centros de procesado de datos.

Sin embargo, actualmente las MV, debido a su arquitectura, no aprovechan los recursos al completo. Las capas y complejidades técnicas provocan que parte del rendimiento se consuma en la gestión de la máquina virtual, en vez de aprovecharse para el cómputo que dicha MV tiene que realizar. Como se introducirá más adelante, en esta última década han ido surgiendo nuevas tecnologías que están remplazando a las máquinas virtuales en ciertos ámbitos de la computación científica.

2.3. Hipervisor

2.3.1. Definición

Acorde a la definición que se da en el libro "*Hardware and Software Support for Virtualization*" escrito por Edouard Bugnion, Jason Nieh, y Dan Tsafrir [15], un hipervisor es una pieza software del sistema operativo especializada en la gestión y ejecución de máquinas virtuales para minimizar el "overhead". Esta pieza software permite que un computador o sistema físico pueda hospedar más de una máquina virtual invitada, a la vez, mediante la compartición de recursos de la máquina física.

Esta pieza de software tiene 2 piezas principales: **"Virtual Machine Monitor (VMM)" y "Device Model"**

- **Virtual Machine Monitor:** Es la pieza encargada de emular y gestionar el conjunto de instrucciones que se ejecutan en modo kernel o supervisor. Es decir, las instrucciones que solo el kernel del sistema operativo puede ejecutar. Acorde a la definición dada por Popek y Goldberg en 1973 [16], todo VMM ha de satisfacer las siguientes tres propiedades fundamentales:
  - **Aislamiento:** Las distintas máquinas virtuales que se ejecuten sobre el mismo sistema físico tienen que estar completamente aisladas entre sí.
  - **Equivalencia:** El sistema tiene que comportarse igual con y sin virtualización. A efectos prácticos se traduce en que, si tenemos 2 máquinas idénticas, una de ellas en una máquina física y otra en una máquina virtual, la mayor parte del ISA tiene que ser igual.
  - **Rendimiento:** El rendimiento no debe verse afectado al virtualizar. Pese a que es imposible que no haya cierto "overhead", este ha de ser el menor posible.
- **Device Model:** Es el controlador de dispositivos. Es la pieza encargada de virtualizar los dispositivos de entrada y salida. Se realiza emulando la trap correspondiente, y devolviendo el control del sistema, tras la interrupción, a la máquina virtual.

2.3.2. Origen y motivación de uso

Su origen se remonta a los finales de los 60, e inicios de los 70. Por aquel entonces, el acceso a un mismo computador, para ejecutar los jobs, por parte de más de un usuario

era muy ineficiente y complicado. Los ingenieros eran conscientes de estas limitaciones e intentaron mejorar la interacción de los usuarios con el sistema para facilitar la labor de lanzar jobs. Sin embargo, estas supuestas mejoras no hacían más que añadir complejidad a los sistemas operativos. Fue entonces cuando ingenieros de IBM crearon lo que hoy en día podemos denominar como máquina virtual. De este modo, cada usuario podía tener su propia máquina virtual aislada de los jobs del resto de usuarios. Además, el sistema operativo de cada máquina virtual no era tan complejo, ya que solo tenía que soportar un único usuario por máquina virtual. A medida que el tiempo pasaba, IBM seguía sacando sistemas operativos con el hipervisor integrado, no estaba pensado como producto, sino como una pieza software para dar soporte a los sistemas operativos como CP-40 o CP-67 que integraban el novedoso concepto de máquina virtual para mejorar la complejidad de los sistemas operativos anteriores [17].
El hipervisor comenzó a ser integrado en los sistemas a partir de 1972, cuando la tecnología de las máquinas virtuales se popularizo. Sin embargo, ya era fundamental cinco años antes en los entornos científicos, y la computación interactiva.

2.3.3. Tipos de hipervisores y su arquitectura

Los hipervisores se dividen en 2 tipos acorde a su arquitectura: Hipervisores de tipo 1 e hipervisores de tipo 2. Esta terminología para diferenciar a los hipervisores en función de su arquitectura fue introducida por Robert Goldberg [16].

- **Hipervisores tipo 1:** Se ejecutan sobre una máquina desnuda, sin sistema operativo. Esta configuración de máquinas sin sistema operativo también es conocida como "bare metal".

- **Hipervisores tipo 2:** Se ejecutan sobre un host ya existente, ya bien sea como parte del sistema operativo ya existente, o sobre él.

En ambos tipos de hipervisores el VMM (Virtual Machine Monitor) crea la máquina virtual. Sin embargo, en el caso de los hipervisores de tipo 1, el VMM también tiene que realizar la planificación y la asignación de recursos en tiempo real, mientras que en los hipervisores de tipo 2 estas tareas las realiza parcialmente el sistema operativo. Cabe destacar, que en la terminología introducida por Goldberg, se da por hecho que todos los hipervisores, sin importar el tipo, se ejecutan en modo privilegiado.

Actualmente los dos tipos de supervisores coexisten en la industria y en los entornos de computación científica. Esto se puede ilustrar mediante la clasificación de los hipervisores más populares y utilizados hoy en día. Por ejemplo, Xen, KVM y VMware son considerados de tipo 1, mientras que Oracle y VirtualBox son hipervisores de tipo 2.

2.3.4. Uso actual en entornos de computación científica

Actualmente, todos los centros de procesamientos de datos hacen uso de la virtualización, por lo que también hacen uso de hipervisores. Los más utilizados en el ámbito científico son Xen y KVM. Ambos son Open-Source.

- **Xen:** Es un hipervisor de tipo 1, desarrollado en el ámbito de los "Linux Foundation Projects". Proporciona servicios que permiten a varios sistemas operativos ejecutar en el mismo hardware de modo concurrente. Xen tiene un modo virtual simplificado, sin controlador de dispositivo. De este modo se asegura de que cada usuario virtual es independiente entre sí. Además de proporcionar este aislamiento entre cada usuario virtual, también nos permite migrar máquinas en caliente, y nos brinda garantías de calidad de servicio.

  Por otro lado, este hipervisor open-source se basa en los dominios de servicio para completar algunas características propias de los hipervisores: acceso al hardware físico, tareas de gestión y administración, etc.

  Xen tiene tantos dominios como máquinas virtuales más el dominio 0, normalmente denominado "**dom0 o Linux Server Domain**". Dom0 es una máquina virtual con privilegios especiales que se arranca siempre tras el arranque del sistema. Es el encargado de administrar el hipervisor Xen y proporcionar los recursos al resto de dominios invitados sin privilegios. Es decir, al resto de las máquinas virtuales creadas por los usuarios [18]. La estructura general de la arquitectura de un hipervisor Xen se puede ver en la figura 2.2.

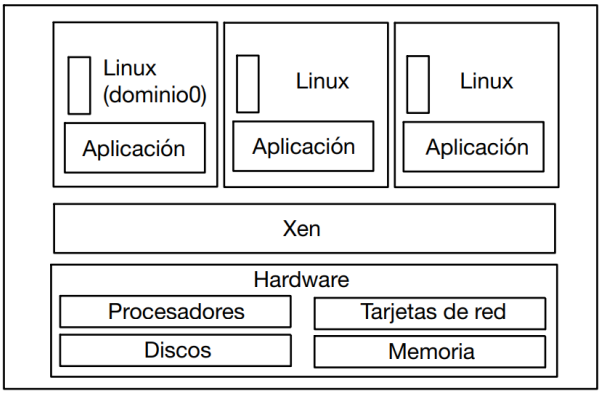

Figura 2.2: Figura adaptada de [2]. Arquitectura del hipervisor Xen.

- **KVM:** Kernel Virtual Machine es un hipervisor de tipo 1 capaz de implementar una virtualización completa con Linux, sin tener que modificar las imágenes de disco. En la figura 2.3 se puede observar cómo KVM consiste en un módulo cargable del kernel de Linux. Este proporciona la infraestructura necesaria para llevar a cabo la virtualización. De este modo, cada máquina virtual que se ejecuta sobre el hipervisor es un proceso regular de Linux.

  Es compatible para procesadores Intel y AMD, proporcionando un módulo distinto para cada procesador correspondiente (kvm-intel.ko / kvm-amd.ko). Asimismo, tiene herramientas en el espacio de usuario para la gestión de las máquinas virtuales. Cada máquina virtual que se ejecuta sobre este hipervisor es un proceso de Linux. KVM utiliza una versión modificada de QEMU como frontend [19].

Puesto que en el cloud del IFCA el hipervisor utilizado es KVM, los unikernels que se desarrollarán estarán orientados a esta tecnología.

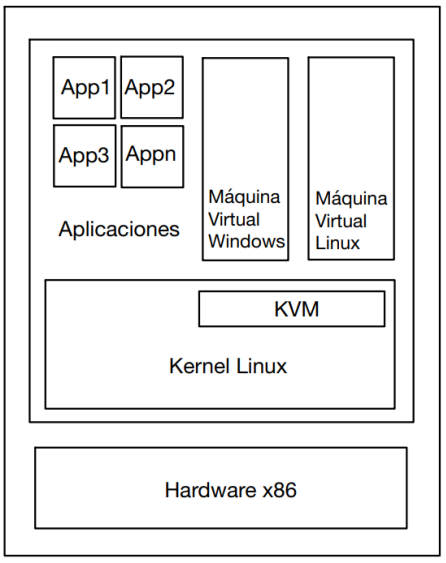

Figura 2.3: Arquitectura del hipervisor KVM. Figura adaptada de [3].

2.4. Contenedor

2.4.1. Definición

La contenerización es un modo de virtualización donde la aplicación se ejecuta aislada en lo que se denomina contenedor sobre el mismo sistema operativo sobre el que se ejecuta la máquina anfitriona. Los contenedores nos ofrecen aislamiento, portabilidad, menor overhead y un alto rendimiento frente a los métodos previamente usados para la virtualización, y en concreto para la virtualización del sistema operativo. El contenedor está compuesto por el código de la aplicación, las bibliotecas necesarias junto a las dependencias necesarias.

El framework más utilizado en la actualidad es Docker. Este framework open-source nacido en 2013, automatiza la creación de contenedores en un entorno de computación determinado. De este modo, las aplicaciones que se ejecutan en cada contenedor son más fáciles de desplegar y ponerlas en marcha, que si lo hiciésemos sobre una máquina virtual, o creásemos el contenedor sin esta automatización. Del mismo modo, Docker ofrece prestaciones que otros sistemas de containerización no nos ofrecen:

- Aislamiento de recursos del kernel de Linux.
- Uso de “spacenames”.
- Capas de control de las imágenes que están corriendo en los contenedores.

2.4.2. Arquitectura

Los contenedores se ejecutan sobre la siguiente “pila hardware”: en lo más bajo de la pila tenemos la infraestructura hardware, sobre ella tenemos el Sistema Operativo base de la

máquina. Sobre este sistema operativo se encuentra el denominado "container engine". Esta última capa es una pieza software que se encarga de gestionar la creación de los contenedores y sus imágenes. Finalmente, en lo más alto de la pila tenemos los contenedores que estemos ejecutando. Estos están compuestos por las librerías, los datos y la aplicación que se desea ejecutar en ese contenedor.

2.4.3. Origen y motivación de uso

Antes de la implantación de los contenedores, día a día se tenía que hacer frente a muchos problemas de portabilidad al mover una aplicación de un entorno a otro. Esto resultaba en problemas de compatibilidad, llegando al punto de no poder ejecutar la aplicación en el nuevo entorno. La primera propuesta similar a lo que hoy es conocido como un contenedor se remonta al año 2000, apareciendo bajo el nombre de *jails*. La versión 4 de FreeBSD fue una implementación de virtualización del sistema operativo que permitía particionar un computador FreeBSD en distintos subsistemas independientes. La principal diferencia que presentaba a las primeras máquinas virtuales era que estaba implementado mediante una llamada de sistema *(syscall)*. Se puede decir que el propósito de los *jails* era compartimentar las máquinas y las máquinas virtuales, de modo que, dentro de una misma máquina los recursos y los archivos estuviesen en distintos compartimentos en función del usuario que accediese a la máquina [20].

En esta primera aproximación a lo que hoy se conoce como contenedores, se fueron asentando algunos de los conceptos de la virtualización a nivel de sistema operativo:

- Los contenedores y jails han de ofrecer una vista límitada de la tabla de procesos, señales entre procesos, interfaces de red, dispositivos de entrada, salida, cuotas e información general del sistema. Solo se ofrecerá vista de los procesos del "compartimento" o "compartimentos" que competen al usuario activo de la máquina.
- Se han de respetar las asignaciones de recursos del resto de contenedores.
- Cuando los servicios de un contenedor no están siendo utilizados, estos se duermen para reducir el consumo de estos lo máximo posible.

Con el paso de los años, las tecnologías de virtualización a nivel de sistema operativo han ido evolucionando hasta los contenedores. Entre todos los frameworks disponibles, el más utilizado es Docker. Sin embargo, podemos encontrar muchos más como RedHat OpenShift, Rancher, Amazon ECS, Google GKE, Microsoft ACS, etc.

Por otro lado, tenemos también los orquestadores de contenedores. Es la capa encargada de gestionar los contenedores y su red. Entre los orquestadores más populares tenemos Kubernetes, Docker Swarm y Apache Mesos.

2.4.4. Uso actual en entornos de computación científica

Los contenedores son una de las tecnologías más utilizadas en entornos de computación científica para la virtualización y ejecución de aplicaciones científicas. Este dato se puede ver reflejado en las estadísticas recogidas en la encuesta anual que realiza *"The Cloud*

Native Computing Fundation" [21].

En la encuesta realizada en 2021, publicada en Febrero de 2022 [22], se muestra la gran adopción de los contenedores entre el personal IT de todo el mundo. Teniendo una muestra de 2302 encuestados, y siendo 1596 de ellos usuarios habituales del cloud público o mediante un enfoque híbrido (cloud público y cloud privado). El 58 % del total de los encuestados afirman haber usado contenedores en los últimos 12 meses, mientras que tan solo un 5 % desconocen que es o nunca han oído hablar de ello. Teniendo en cuenta que la muestra es de 2302 personas, tan solo 115 personas no conocen esta tecnología.
Los datos detallados, junto a las preguntas realizadas en la encuesta se pueden encontrar en el siguiente repositorio de Github [23].

2.5. Unikernel

2.5.1. Definición

Un unikernel, también denominado como "single-purpose OS", es una imagen ejecutable que combina un sistema operativo especializado, denominado biblioteca, con una aplicación, ejecutándose todo ello en un único espacio de direcciones. Esto hace que las métricas de rendimiento energético, consumo de memoria y uso de CPU puedan verse mejoradas frente al uso de contenedores en el ámbito de la virtualización, al reducir lo máximo posible las funcionalidades y los recursos necesarios frente al desarrollo de una misma aplicación en una máquina virtual o un contenedor.

Es importante destacar que a partir de ahora nos vamos a referir a los distintos frameworks existentes para la creación de estas imágenes como "unikernels".

2.5.2. Origen y motivación de uso

Tal y como se introduce en la sección 1.1, el concepto que reside bajo el término de unikernel, proviene de los primeros sistemas operativos, en los años 60, los cuáles solo tenían un único espacio de direcciones. Todos los procesos que se quisieran ejecutar en este tipo de sistemas operativos tenían que compartir un único espacio de direcciones: la memoria física del computador. En ella se cargaba el programa que se tenía que ejecutar con las librerías necesarias para su ejecución. No obstante, no era una implementación muy eficaz, pues mucho espacio de memoria se quedaba inutilizado y no se podían ejecutar procesos de manera concurrente, solo el que estaba cargado en memoria física.

El mundo de la computación siguió evolucionando y, para poder ejecutar más de un proceso se implementó la multiprogramación. Además de tener unas direcciones del espacio de memoria reservadas para el sistema operativo, se asignaba una región de memoria a cada proceso, dividiendo la memoria física en regiones. Es necesario mencionar que se tuvieron que añadir mecanismos para proteger a los jobs del resto, ya que un proceso malicioso podía escribir en direcciones de memoria asignadas a otro proceso. De este modo, cada proceso estaba aislado del resto. Mientras un proceso se estaba ejecutando en la CPU, otro proceso cargado en memoria podía estar realizando operaciones de entrada

y salida. Este mecanismo se pudo ver implementado en máquinas como la IBM 7094 [24].

Con el paso de los años este tipo de gestión de memoria en los sistemas operativos fue desapareciendo de los computadores más utilizados por aquel entonces, debido a los problemas de protección que presentaba. Fue entonces cuando apareció la virtualización de memoria, cambiando por completo este paradigma y alejándonos del concepto de sistema operativo de biblioteca [25].

Sin embargo, en 2016, Anil Madhavapeddy y David J. Scott publicaron un artículo científico [10] en el que se propone traer de vuelta este tipo de gestión de memoria mediante lo que denominaron como unikernel. Un único espacio de direcciones, pero solo para una aplicación, con un propósito bastante concreto: implementarlo en el mundo de la computación en la nube, los microservicios y la virtualización para mejorar los tiempos de respuesta y el rendimiento.

2.5.3. Tipos de Unikernels y su arquitectura

Arquitectura

La arquitectura de los unikernels está basada en una arquitectura surgida en los 90, llamada libOS [25], en la que se encapsulan las funcionalidades que necesitamos del Sistema Operativo en una librería. En esta arquitectura los límites de protección están implementados en las capas hardware, mediante la implementación de librerías y políticas de control. Esta implementación varía en función del framework de unikernel que se emplee.

Esta arquitectura tiene grandes ventajas para sistemas que lo implementan, como NUSE [26], ya que se permite a la aplicación el acceso a los recursos hardware directamente, sin tener que adquirir privilegios y cambiar de espacio de direcciones, de userspace a kernelspace. De este modo se evita el overhead que ello conlleva. Por otro lado, libOS no tiene un mecanismo para priorizar flujo a la hora de recibir paquetes de datos por la red, como los mecanismos de “Calidad de Servicio”. En su lugar, las aplicaciones que implementan esta arquitectura tienen que emplear colas separadas puesto que no tienen un servicio de red central [10].

Otra característica relevante es que se necesitan crear librerías de protocolo para remplazar los servicios de un sistema operativo tradicional, así como drivers. Si no fuese por el uso de los hipervisores, esto último supondría una desventaja frente a otras arquitecturas, puesto que los drivers de los dispositivos deberían ser adaptados o reescritos.

Como se muestra en la figura 2.4, en comparación con la arquitectura de las máquinas virtuales, la arquitectura de los unikernels (basada en libOS), es mucho más sencilla. El unikernel está compuesto tan solo por las librerías del Sistema Operativo que necesitan para la ejecución de la aplicación elegida. Una vez se genera el ejecutable, este puede ser ejecutado sobre bare metal o sobre un hipervisor. En este TFG, todos los despliegues que vamos a realizar se van a ejecutar en el servicio cloud del IFCA. Por lo tanto, se ejecutarán sobre el supervisor implementado, que en este caso es KVM.

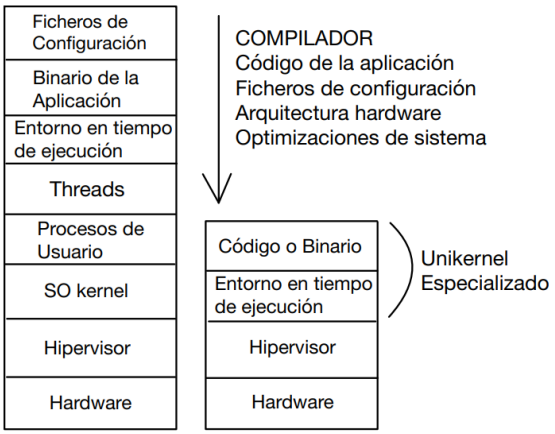

Figura 2.4: Figura adaptada de [4]. Arquitectura de un unikernel en comparación con una máquina virtual.

Tipos de unikernels

Actualmente, los unikernels se puede clasificar en dos grupos principales en función del propósito de uso: **Clean State** y **Legacy** [27].

- **Clean State**: Este tipo de unikernels han de ser creados desde cero. El código de la aplicación ha de escribirse en el lenguaje especificado por el framework correspondiente. De este modo, se asegura un alto rendimiento con las librerías de sistema diseñadas para dicho framework y lenguaje de programación.
- **Legacy**: El objetivo de este tipo de unikernels es que se pueda adaptar una aplicación ya existente sin realizar modificaciones o con cambios menores. De este modo, no habría que rehacer aplicaciones enteras para poder convertirla en un unikernel.

Debido a que el propósito de este Trabajo Fin de Grado es evaluar, mediante un análisis de rendimiento, la posible implementación de los unikernels en entornos de computación científica, y el remplazo de los contenedores, buscaremos un framework de unikernels de tipo Legacy. Esto se debe a que no es viable rehacer el código de la mayoría de aplicaciones científicas que se ejecutan en un Centro de Procesado de Datos como el IFCA.

Además, la inmensa mayoría de estas aplicaciones están escritas en más de un lenguaje de programación. Lo que dificulta su adaptación a algunos frameworks de unikernels, como veremos en el posterior análisis de los frameworks más populares hoy en día.

2.5.4. Uso actual en entornos de computación científica

Los unikernels hoy en día siguen en desarrollo, por lo que la utilización de estos en entornos de computación es aún incipiente. Además de ser relativamente novedoso, ya que es una tecnología surgida a partir del 2015, presenta un problema que otras tecnologías como los contenedores no presentan: la dificultad técnica. Para poder crear un unikernel, sobre todo si es “clean state”, se necesita tener amplios conocimientos de un lenguaje

de programación en concreto. A esto se le suma, saber elegir correctamente las librerías necesarias para el correcto funcionamiento de la aplicación a la hora de la creación del unikernel, y la configuración y adaptaciones necesarias para que sea ejecutable sobre el supervisor que se tenga.

Sin embargo, sí es cierto que, debido a que ciertas empresas desarrolladoras de unikernels están trabajando en la creación de herramientas de gestión, creación y orquestación de unikernels, esta tarea cada vez es más sencilla. Algunas de estas herramientas permiten la creación de unikernels con un gran nivel de abstracción y totalmente transparente para el programador.

2.5.5. Diferencias entre unikernel, microkernel y exokernel

Al igual que los unikernels, los microkernels y exokernels también son propuestas de actualidad para mejorar ciertos aspectos de la computación, relacionados con los cores de los sistemas operativos, virtualización y su gestión. en relación con Sin embargo, presentan ciertas diferencias notables entre ellos que pueden dar lugar a equívoco, empleando los términos de manera errónea. A continuación paso a definir cada uno de los términos:

- **Microkernel:** Un microkernel es, como su propio nombre indica, un kernel de un sistema operativo con las funcionalidades y características mínimas necesarias para poder dar servicio. Los servicios orientados a la gestión de dispositivos de entrada y salida, o los relacionados con comunicación entre procesos, se delegan a los servicios y procesos de usuario [28].

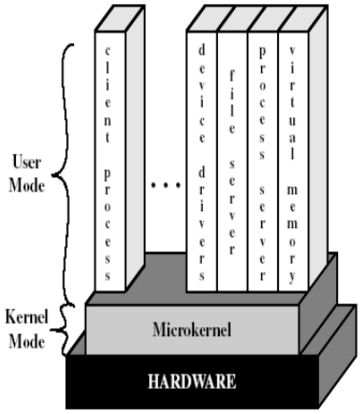

Figura 2.5: Estructura simplificada de un Microkernel [5]

La principal diferencia entre los microkernels y los unikernels está en el propósito de la reducción del SO. El microkernel pretende minimizarlo hasta conseguir las funcionalidades básicas de un sistema operativo convencional. En contraposición el unikernel lo hace para tener la funcionalidad mínima que satisfaga las necesidades de mi aplicación.

- **Exokernel:** Un exokernel es un tipo de sistema operativo, propuesto por el MIT (Massachusetts Institute of Technology), con el que pretenden proporcionar funcionalidades propias del hardware, como puede ser la gestión de recursos, a nivel aplicación. El kernel se sitúa sobre el software y las librerías, estableciendo comunicación solo con estas últimas. En fin de este cambio de paradigma es proporcionar un mayor control a los programadores, usando el menor número de capas de abstracción. Todo ello teniendo en cuenta la protección de los recursos. [6]

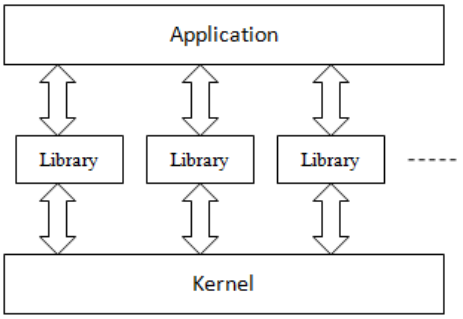

Figura 2.6: Estructura simplificada de un Exokernel [6]

Por lo tanto la diferencia de un exokernel respecto a un unikernel reduce al modo en el que se reduce el sistema operativo, así como el propósito de su uso y creación. Mientras que un exokernel reduce sus sistema operativo para migrar la gran parte de su código a una capa superior a la aplicación, un unikernel minimiza el SO mediante la eliminación de parte de código innecesaria para la aplicación que se desea ejecutar.

2.6. Comparación entre MVs, Contenedores y Unikernels

Teniendo en cuenta todos conceptos que se han introducido hasta ahora, podemos realizar algunas afirmaciones que nos posicionan en el punto de querer evaluar el rendimiento de los unikernels frente al resto de tecnologías introducidas anteriormente:

- Pese a que las máquinas virtuales nos brindan aislamiento completo del host y hay muchas plataformas de orquestación, se requiere una infraestructura desproporcionada debido a que cada instancia tiene que cargar un sistema operativo completo.

- Los contenedores, debido a su arquitectura, son más ligeros que las máquinas virtuales, lo que resulta en un mejor rendimiento y menores tiempos de boot. Sin embargo, el aislamiento entre el host y la aplicación conteinerizada da lugar a problemas de flexibilidad y problemas de seguridad.

- Los unikernels son más ligeros debido a la especialización al utilizar solo las librerías necesarias. Además, ofrece aislamiento completo del host, ya que no comparte kernel del SO con el host. La parte negativa del uso de los unikernels es la poca repercusión que tienen hoy en día y la uniformidad de todos los frameworks para estandarizar la tecnología. Por otro lado, la poca repercusión y uso que hay de los

unikernels, hace que no se hayan desarrollado muchas herramientas que faciliten el uso y despliegue de los unikernels.

2.7. OpenStack

2.7.1. Definición y concepto

OpenStack [29] es una plataforma de cloud computing open-source, escrita en Python, para virtualizar, organizar y gestionar los recursos hardware del cloud público o privado. El despliegue que normalmente se hace de esta plataforma es el propio de una IaaS. Actualmente esta gestionada por la OpenStackFundation.

2.7.2. Arquitectura y componentes

La plataforma está compuesta por una serie de componentes que se comunican entre sí para proporcionar los servicios. Cada uno de los componentes proporciona un servicio distinto. A continuación se enumeran los más importantes [30]:

- **Horizon:** Es el componente encargado del dashboard con el que interacciona el usuario y el administrador.
- **Keystone:** Es el componente responsable de la autenticación y la seguridad. Se debe a que, para utilizar cualquier otro componente o servicio, hay que autenticarse, y posteriormente, verificar si el usuario tiene autorización para hacer uso o acceder a ciertos recursos.
- **Nova:** Es el componente encargado del dominio de cómputo.
- **Glance:** Es el componente responsable del servicio que ejecuta las instancias de máquinas virtuales.
- **Swift:** Es el componente encargado del servicio de almacenamiento de recursos y objetos en OpenStack.
- **Neutron:** Es el componente que se encarga del servicio de red y comunicación entre las distintas máquinas.
- **Cinder:** Es el componente responsable del servicio de “block storage”.
- **Heat:** Es el componente que se encarga de la orquestación de los servicios.
- **Ceilometer:** Es el componente encargado de las métricas de consumo de las instancias para realizar la facturación al usuario.

2.7.3. Uso actual en entornos de computación científica

OpenStack es una de las IaaS más utilizadas en entornos cloud. Puesto que fue inicialmente desarrollada por la NASA y Rackspace en 2005, actualmente, pese a no ser propietarios, hacen uso de ella en para sus servicios cloud. Otras grandes empresas como son RedHat, actual desarrollador de la plataforma, CISCO, IBM y HP hacen uso de esta tecnología. En el IFCA, es la plataforma desplegada para acceder a la parte cloud del Centro de Procesado de Datos y mediante la que desplegaremos las máquinas virtuales y los unikernels para realizar el análisis.

Capítulo 3

Análisis de frameworks de unikernels

A continuación, se enumeran y describen los frameworks de unikernels más utilizados hoy en día. Para la elección de estos seis unikernels, he buscado los proyectos más relevantes que actualmente hacen uso de unikernels open-source, así como la relevancia del equipo que los ha desarrollado y el estado de mantenimiento de sus repositorios.

3.1. Enumeración de unikernels existentes hoy en día

Se pasan a enumerar los unikernels seleccionados junto a una breve descripción:

3.1.1. MirageOS

Framework inicialmente desarrollado por la Universidad de Cambridge en 2008. Se presenta como una implementación de los sistema operativo de biblioteca (Library Operating System) que nos brinda la posibilidad de crear unikernels de tipo clean state. Para ello tenemos que desarrollar el código con la funcionalidad deseada, y posteriormente compilarla junto a las librerías OCaml que nos proporcionan el almacenamiento, funcionamiento de red y el soporte de concurrencia. De este modo obtendremos el unikernel capaz de correr en cualquier máquina sin necesidad de depender de ninguna pieza externa de software. El código en el que se ha de programar la aplicación es OCaml.

Este unikernel está orientado para la construcción de aplicaciones seguras de alto rendimiento para su despliegue en diversas aplicaciones de cloud computing basadas en Xen (principalmente testado en Amazon EC2). En caso de querer realizar el despliegue sobre otros hipervisores, MirageOS proporciona un middleware y la documentación necesaria para realizarlo [31].

3.1.2. HermitCore

HermitCore es un framework relativamente nuevo (2016) para unikernels enfocado a entornos cloud y de HPC desarrollado por la Universidad de Aechen, en Alemania (RWTH Aachen University). Actualmente, pese a no estar en producción, soporta código en C, C++, Fortran y GO. Además, soporta Pthreads, OpenMP, y MPI e iRCCE para el paso de mensajes.
HermitCore puede ser usado como un unikernel clean state (funcionalidad y propósito principal), o con un enfoque multikernel. Es decir, está optimizado para correr múltiples kernels simultáneamente en cada nodo de cómputo, con los beneficios y características de los unikernels. HermitCore es una combinación de unikernel con multikernel, no unikernel puro [32].
Está diseñado para KVM o computadores con sistemas operativos Unix-like. Sin embargo, también soporta entornos "bare metal" sobre máquinas x86 de 64 bits. En el caso de bare metal, proporciona una mejor programabilidad y escalabilidad [7].

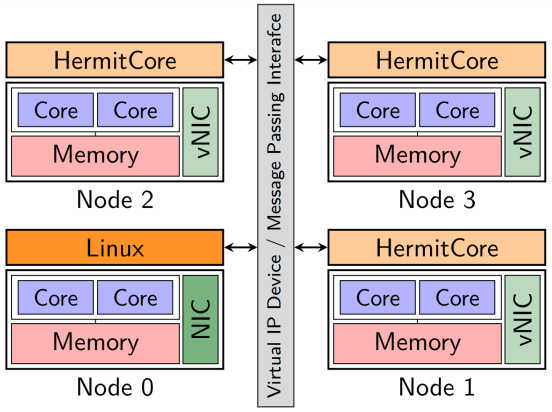

Figura 3.1: Arquitectura simplificada del unikernel HermitCore [7].

3.1.3. IncludeOS

Framework de unikernels desarrollado por IncludeOS A.S., una empresa noruega fundada en 2014 y dirigida por Alfred Bretterud. Permite el desarrollo de unikernels de tipo clean state, con el propósito de que se ejecuten en entornos cloud con hipervisores KVM [33].

El lenguaje de programación en el que se tienen que escribir las aplicaciones es C++, ya que permite la compilación de modo nativo de este lenguaje. Además, aseguran que IncludeOS mantiene compatibilidad de código con Linux. Esto hace que todas las librerías que se pueden compilar en Linux, también puedan en IncludeOS gracias al uso de la librería Musl [34].

IncludeOS es "event-driven", es decir, sigue una arquitectura dirigida por eventos. El sistema operativo emula las llamadas al sistema en función del framework de los eventos. El modelo de eventos que emplea IncludeOS está basado en Node.js [35].

3.1.4. ToroKernel

Framework de unikernels desarrollado en Pascal que crea unikernels capaces de ejecutar sobre bare metal (máquinas x86), los hipervisores KVM, Xen, VirtualBox e HyperV. El propósito de los unikernels creados por este framework es desarrollar microservicios. De hecho, la implementación de Torokernel, propone dos tipos de sockets, o modos de acceso, para construirlos: sockets bloqueantes y sockets no bloqueantes. Esto permite que haya conexiones concurrentes reduciendo el número de cambios de contexto. Esta característica, acorde a las documentaciones que aportan los desarrolladores de los frameworks, es únicamente de Torokernel, ya que el resto no especifíca nada al respecto.

Los unikernels clean-state que se desarrollen mediante este framework tienen que ser desarrolladas en FreePascal. Torokernel está compuesta por una serie de componentes básicos escritos en FreePascal: La memoria, el sistema de ficheros, los procesos, la red y los dispositivos. Para la gestión y desarrollo de esta tecnología se ha de utilizar una API desarrollada explícitamente para ello [36].

- **Memoria**: Proporciona la API para gestión de memoria.
- **Sistema de ficheros**: Proporciona la API para permitir el acceso a los ficheros.
- **Procesos**: Según la documentación oficial, proporciona la API para gestionar los threads.
- **Red**: Incorpora lo hablado previamente, los sockets bloqueantes y los no bloqueantes.
- **Dispositivos**: Proporciona gestión de entrada y salida para las microservicios que lo requieren.

3.1.5. OSv

Es un framework de unikernel de tipo legacy desarrollado inicialmente por Cloudius Systems, ahora conocido como ScyllaDB. Como el resto de frameworks, asegura poder ejecutar y desplegar microservicios, servicios cloud y aplicaciones "serverless" con un mayor rendimiento que otras tecnologías como los contenedores. Una característica que se destaca en su página oficial [37] es la facilidad de despliegue, pues pretende que la tarea de creación de estos unikernels legacy sea cada vez más sencilla para los desarrolladores.

En lo referente a la arquitectura que emplea, es bastante similar al resto de unikernels estándar, pues utiliza la arquitectura principal denominada "Simplified cloud stack" en la que se prentende tener una infraestructura mucho más ligera, precisando solo del hardware y del hipervisor. Sobre este stack tan solo haría falta las librerías de OSv, o como ellos lo denominan, OSv VM, y la aplicación en sí.

Cabe destacar que OSv soporta ciertos "language runtime" como JVM, Pyhton, Python3, Node.JS y Ruby, además de lenguajes que se compilan directamente a código de máquina de modo nativo como Rust y Golang [37].

3.1.6. OPS/Nanos

OPS y Nanos conforman la tecnología para crear unikernels desarrollada por NanoVMs, una empresa dirigida por Ian Eyberg. OPS es la herramienta utilizada para crear, compilar y ejecutar el unikernel Nanos. Nanos es un unikernel de tipo legacy. Desde sus inicios, se diseñó para poder ejecutar una aplicación ya existente que se encuentre en un binario ELF. Pese a ser diseñado originalmente para poder ejecutar aplicaciones ya existentes, también permite la creación de aplicaciones clean-state en los siguientes lenguajes de programación: C, Golang, PHP, NodeJS, Perl, Python, Ruby, Rust y Java. Puede ejecutarse sobre máquinas desnudas y distintos hipervisores: Xen y KVM [38].

Sus desarrolladores aseguran que la ganancia que se obtiene mediante el uso de sus unikernels frente a los contenedores se debe al uso de discos, red y virtualización nativa, además de eliminar los cambios de contexto entre procesos.

Comparativa Unikernels				
Nombre Unikernel	Tipo	Hypervisor	Lenguaje	Entorno
HermitCore	Clean State	KVM/Linux (x86-based)	C/C++, Fortran, Go	Cloud, HPC
IncludeOS	Clean State	KVM, VirtualBox, ESXi, Google Cloud, OpenStack	C++	Cloud y PC's
MirageOS	Clean State	KVM, Xen, RTOS/MCU	OCaml	Cloud y HPC
OPS/Nano	Legacy	QEMU/KVM, XEN, ESXi, Amazon EC2, Google Cloud, HyperV	C, Golang, PHP, NodeJS, Perl,Python, Ruby, Rust, Java	Cloud, empresa
OSv	Legacy	VirtualBox, ESXi, KVM, Amazon EC2, Google Cloud	Java, C, C++, Node, Ruby	Cloud, IOT
ToroKernel	Clean State	VirtualBox, KVM, XEN, HyperV	FreePascal	Microservicios

Cuadro 3.1: Tabla resumen de los seis unikernels seleccionados.

3.2. Comparativa de los unikernels existentes

En la siguiente tabla comparativa se presenta un resumen de los seis unikernels que vamos a someter a estudio, junto a algunas características importantes para su evaluación. Estas características son: el tipo de unikernel, los hipervisores sobre los que se puede ejecutar, los lenguajes de programación sobre los que se pueden programar y en el que están desarrolladas sus librerías, y, finalmente, el entorno para el que están desarrollados.

Todos los datos mostrados en esta tabla han sido obtenidos de las páginas oficiales de cada proyecto y sus respectivos repositorios oficiales donde se desarrollan [7] [35] [31] [38] [37] [36].

3.3. Análisis

Teniendo en cuenta los datos mostrados en la tabla comparativa, y la previa enumeración de los unikernels a estudio, se ha de elegir un unikernel que pueda ejecutar aplicaciones cloud que actualmente se estén ejecutando en un entorno HPC, y que sea capaz de desplegarse sobre el CPD del IFCA.

Por lo tanto, el unikernel ha de ser de tipo legacy y ejecutable sobre un hipervisor KVM. Esto nos lleva a reducir el espectro a tan solo 2 unikernels: OSv y OPS/Nanos. Por lo tanto, pasamos a analizar las diferencias entre OSv y OPS/Nanos.

3.3.1. OSv vs OPS/Nanos

Puesto que los mostrados en la tabla comparativa no hace inclinar la balanza hacía ninguno de los 2 unikernels, ya que son muy similares en las características expuestas en la tabla, haremos un análisis más en profundidad sobre otros aspectos, también importantes, para seleccionar la herramienta con la que trabajar:

- **Propósito:** El propósito de ambos es poder ejecutar una aplicación en una imagen pequeña que permita un rápido arranque, y que pueda ejecutar en cualquier plataforma, sobre cualquier hipervisor. A su vez, ambos intentan dotar a la herramienta de una facilidad de uso, de este modo no solo expertos pueden desarrollar un unikernel, sino que cualquier usuario que tenga unos mínimos conocimientos sobre el despliegue de aplicaciones de propósito específico puedan hacer uso de ella.

- **Modo de desarrollo del unikernel:** Ambos frameworks nos ofrecen una herramienta para simplificar la tarea de creación. En el caso de OSv es *Capstan* [39] y *OPS* [38] en el caso de Nanos. Sí es cierto que OPS además de ser la herramienta mediante la que se crea y se compila el unikernel de un modo más sencillo, también sirve para la orquestación. Por otro lado, OSv precisa de una librería adicional en caso de estar intentando compilar en un unikernel aplicaciones Linux. En contraposición, OPS no distingue en función de la plataforma en la que se diseño originalmente. Este aspecto es una ventaja para los usuarios que hagan uso de OPS, pues pueden desarrollar las mismas funcionalidades que con Capstain, pero sin necesidad de tener que emplear la librería adicional [39] [38].

- **Diseño e implementación:** Como todos los unikernels menos HermitCore, ambos utilizan la arquitectura de la gran mayoría de frameworks de unikernels:“Simplified cloud stack”. Así es como se consigue eliminar costes, a nivel de rendimiento y utilización de los recursos, en lo que respecta al aislamiento de las aplicaciones. Ambos ejecutan una única aplicación en la que se ejecuta un solo proceso mediante múltiples threads. De este modo, no necesitamos protección entre el espacio de usuario y el kernel.

  Las llamadas al sistema, en ambos casos, son llamadas a funciones de las librerías esenciales que nos proporciona cada desarrollador de unikernels. De este modo, se evita overhead por las llamadas al las “system calls” inexistentes en estos dos unikernels [39] [38].

- **Dificultad de uso acorde a documentación:** La documentación de ambas tecnologías afirma que desplegar sus unikernels es muy simple. En ambos casos, he encontrado congresos y seminarios con ejemplos de puesta en marcha de los unikernels

que facilitan el despliegue de los mismos por parte de los usuarios.

- **Nivel de abstracción:** El nivel de abstracción que aporta OPS es mucho mayor que el que aporta Capstan, haciendo más fácil el desarrollo del unikernel por parte de gente que no tiene conocimientos técnicos sobre virtualización. El mayor nivel de abstracción se debe a la gran facilidad de uso de la herramienta, pues mediante directivas como *run*, *build* y *list*, puedes crear los unikernels. Este punto es muy importante, ya que no solo ingenieros informáticos o trabajadores con roles similares hacen uso de las infraestructuras cloud o HPC, sino que también otros profesionales como Físicos o Matemáticos, entre otros, hacen uso de la tecnología para poder lanzar sus jobs. Estos profesionales no tiene porque tener un amplio conocimiento de la tecnología, más allá del uso de la herramienta, pues no es su objetivo final. El objetivo final de este grupo de especialistas es poder ejecutar sus aplicaciones para obtener los datos salientes, para posteriormente analizarlos.

  Por otro lado, la documentación de OPS/Nanos es de mayor calidad, pues presenta ejemplos de uso junto a tutoriales de uso en el repositorio oficial de GitHub [40] para aprender desde cero a usar OPS y comprender los fundamentos y aspectos generales de la tecnología.

3.4. Unikernels seleccionados

Tras el análisis realizado en el apartado anterior, elegí OPS/Nanos como unikernel en el que desarrollar el despliegue de la aplicación científica para analizar el rendimiento de esta. Ambos unikernels presentan una arquitectura muy similar, y el propósito es el mismo en ambos casos. Tomé la decisión en base a la gran calidad de la documentación de OPS/Nanos, el nivel de abstracción que ofrece la herramienta OPS, el servicio de atención en directo y la dificultad de uso que tiene en estos momentos NanosVM.

Sin embargo, si en algún punto los unikernels se adoptan como tecnología a usar en el ámbito científico, puede resultar interesante realizar el desarrollo de las nuevas y futuras aplicaciones directamente sobre esta tecnología de modo nativo. Por lo tanto, pese a que se queda al margen del propósito final de este Trabajo Fin de Grado, veo conveniente una pequeña comparación y análisis de la facilidad de uso de un unikernel de tipo clean-state. Dentro de los unikernels clean-state que se han tenido en cuenta en el análisis, se elige MirageOS para realizar esta implementación. La elección se ha tomado en base a que es el unikernel de tipo clean-state más desarrollado hasta la fecha y el más utilizado.

Por lo tanto, a continuación se detallan las especificaciones técnicas y el modo de creación de unikernels de prueba en local. Además, se evalúa, para el análisis de rendimiento, el unikernel de tipo legacy: OPS/Nanos.

3.4.1. MirageOS: Especificaciones técnicas y modo de creación

Especificaciones técnicas

Como hemos introducido en el apartado 3.1.1, MirageOS es un framework de unikernels, cuya creación data de 2008, basado en una arquitectura de sistema operativos de los años 90, denominada library Operative System, conocido como libOS.

En este tipo de arquitectura, los límites de protección a nivel de seguridad y aislamiento llegan hasta las capas hardware más bajas, dando lugar a un conjunto de librerías especializadas que implementan mecanismos específicos que permiten la comunicación con el hardware y hablar los protocolos de red, entre otros propósitos. Estas librerías también nos permiten establecer un conjunto de políticas que refuerzan el acceso y control en la capa de aplicación.

Asimismo, el uso de un unikernel clean-state hace que la optimización sea máxima, pues la aplicación está diseñada ad-hoc en el lenguaje de programación correspondiente, y se tiene un gran control sobre las dependencias de la aplicación.

Para la creación de los unikernels utiliza Opam [41] como herramienta principal. Opam es un gestor de paquetes para OCaml, el lenguaje en el que se ha de programar la funcionalidad de la aplicación. Se puede decir que es análogo a apt en Ubuntu. Las directivas utilizadas para su uso se asemejan mucho a la de otros gestores. Por ejemplo:

```
$ opam list -a       #Lista paquetes OCaml disponibles
$ opam install lwt   #Instala el paquete lwt
$ opam update        #Actualiza paquetes OCaml disponibles
$ opam upgrade       #Upgrade paquetes disponibles
```

Creación de un unikernel de prueba en local

Mediante los comandos mostrados previamente, se consigue instalar el gestor de paquetes para OCaml (Opam), y su inicialización. Opam necesita inicializar su directorio de trabajo (/.opam). Además, el comando init también elige, de modo automático, un compilador a instalar, al no ser que se indique explícitamente mediante la flag "–bare".

Durante la configuración, se nos pedirá que configuremos las variables de entorno que necesita Opam para hacer a la shell consciente de estos nuevos cambios. Simplemente se han de aceptar las sugerencias que aparecerán en la terminal para una correcta configuración del entorno. Igualmente, se ha de instalar el paquete de OCaml, Mirage, mediante Opam. Este es el paquete que nos facilitará todas las directivas de MirageOS para crear el unikernel. En el Anexo podremos encontrar las directivas necesarias para realizar estas tareas A.2.1.

Una vez instalado el gestor de paquetes de OCaml y el paquete necesario para crear el unikernel, necesitamos crear el unikernel. El unikernel creado por MirageOS necesita 2 archivos: una configuración y la aplicación, ambos archivos han de estar en formato *.ml*.

Estos 2 archivos podemos obtenerlos del repositorio oficial de github de mirageOS [31], donde tenemos ciertos proyectos de ejemplo. Para esta prueba utilizaremos el ejemplo del Hello World.

```
$ cat config.ml

    open Mirage

    let main = main ~packages:[ package "duration" ]
    "Unikernel.Hello" (time @-> job)
    let () = register "hello" [ main $ default_time ]

$ cat unikernel.ml

    open Lwt.Infix

    module Hello (Time : Mirage_time.S) = struct
        let start _time =
            let rec loop = function
            | 0 -> Lwt.return_unit
            | n ->
                Logs.info (fun f -> f "hello");
                Time.sleep_ns (Duration.of_sec 1)
                >>= fun () -> loop (n - 1)
            in
            loop 4
        end
```

El fichero unikernel.ml es la funcionalidad de aplicación que quiero que tenga mi unikernel. Al estar escrita en OCaml, hace uso de una librería propia, que imprescindible para crear cualquier unikernel de MirageOS: *"Lwt threads"*. Para más información sobre esta librería y la descripción línea a línea del código OCaml, ver en el AnexoA.2.2.

Finalmente, se ha de evaluar la configuración del unikernel, instalar las dependencias necesarias y compilar el unikernel. En este punto, la ejecución del unikernel se realiza del mismo modo que se ejecuta cualquier otra aplicación tras realizar la compilación y obtener el ejecutable.

Reflexiones MirageOS: Facilidad de uso

Tras estudiar la documentación ofrecida por MirageOS, e intentar realizar un desarrollo de un unikernel de prueba en local con código de prueba ya aportado por los desarrolladores, he realizado las siguientes reflexiones en lo referente al posible desarrollo de unikernels clean-state para aplicaciones científicas:

- Hoy en día OCaml no es un lenguaje de programación ampliamente usado en el ámbito de aplicaciones científicas. Los lenguajes de programación que más predominan en este ámbito son Python, C/C++ y R [42]. Por lo tanto, es complicado que

los desarrolladores de las aplicaciones aprendan y cambien de lenguaje de programación teniendo en cuenta la curva de aprendizaje que supone, no solo aprender un nuevo lenguaje de programación, sino dominarlo.

- En caso de querer desarrollar desde cero una aplicación científica, tenemos que escribir toda su funcionalidad en OCaml, así como la configuración. Esto limita en gran medida la compatibilidad con librerías que hayan podido ser desarrolladas en proyectos pasados y que sean necesarias para estos nuevos proyectos, suponiendo una gran desventaja a la hora de utilizar unikernels clean-state para el propósito a estudio en este trabajo.

- En el caso de querer desplegarlo posteriormente sobre una infraestructura cloud, dependerá del hipervisor empleado la facilidad de esta portabilidad. Este unikernel es parte del *"Xen Project"*, lo que indica que está optimizado para la ejecución sobre el supervisor Xen. En caso de querer desplegarlo sobre un supervisor KVM se ha de utilizar un middleware para hacerlo posible: Solo5 [43].

3.4.2. Ops/Nanos:

Especificaciones técnicas

Este framework de unikernels está compuesto por el unikernel en sí: Nanos, y su herramienta de gestión y orquestación: OPS.

Nanos es un unikernel desarrollado para arquitecturas x86-64 y ARM. Este unikernel, a diferencia de otros que pueden ser desarrollados sobre bare metal, siempre ha de ser desplegado sobre el hipervisor correspondiente.

A nivel de arquitectura, este unikernel usa un espacio de direcciones propio para almacenamiento, en vez de duplicar el almacenamiento (ficheros y directorios) como hacen los contenedores. Además, a nivel de red tan solo se apoya en las capas de red que nos proporciona el hipervisor.

El sistema de fichero que utiliza es el conocido como TFS (Temporary File System). Este tipo de sistema de ficheros utiliza la memoria local para las lecturas y escrituras que necesite hacer. Esto hace que la velocidad de escritura y lectura alcanzada sea mucho mayor que la de un sistema de ficheros convencional. Es cierto, que un unikernel puede permitirse montar este tipo de FS debido a que no tiene sistema operativo al uso, como ya hemos explicado anteriormente. Recalcar que, aún así, NanosVm admite el uso de controladores de almacenamiento como virtio_blk (QEMU-KVM) y Storvsc (Hyper-v/Azure).

Respecto a la seguridad, los unikernels se caracterizan por solventar gran parte de los problemas que presentan los contenedores, y Nanos no se queda atrás. Puesto que es un sistema de un único proceso (aunque multi-threaded), significa que no tiene soporte para SSH, shells o programas que ejecuten en segundo plano o en modo interactivo. Esto evita muchos tipos de ataques asociados a las conexiones SSH, o mediante inyecciones al proceso que se está ejecutando, pues no se puede crear otro proceso mientras está corriendo

el correspondiente a la aplicación. Adicionalmente, al ser un único proceso sin sistema operativo al uso, la superficie de ataque se ve reducida en gran modo. Otro aspecto a mencionar, es que el kernel del unikernel está en una partición distinta a la partición visible por el usuario, por lo que también aplica otra capa de seguridad al realizar está partición.

Por ello, resaltar que el hecho de solo poder ejecutar un proceso a la vez, el unikernel no soporta paralelismo a nivel de proceso. Por lo que todas las aplicaciones ya desarrolladas que quieran hacer uso de estás funcionalidades, como son las aplicaciones con MPI, verían limitaciones en su implementación. MPI se podría utilizar para comunicar diferentes procesos en diferentes unikernels, pero no procesos MPI en un mismo nodo.

Para finalizar, mencionar que actualmente Nanos puede desplegarse en varios proveedores de cloud públicos como Google Cloud, Amazon Web Server, Microsoft Azure, Oracle Cloud y Openstack.

El funcionamiento de la herramienta de orquestación OPS se verá en la creación del unikernel de prueba.

3.4.3. Creación de unikernel OPS/Nanos de prueba en local

Para la creación del unikernel Nanos hay que tener instalado el emulador QEMU en el PC. Una vez está instalado se puede pasar a descargar la aplicación de OPS del repositorio oficial [38].

A continuación, si la instalación se ha realizado correctamente, tan solo se tiene que crear el unikernel y ejecutarlo. Para probarlo, se utiliza uno de los ejemplos disponibles en el repositorio oficial: Un servidor web escrito en JavaScript que muestra un “Hello World”. El código a escribir es el siguiente:

```
$ touch hi.js
```

```
var http = require('http');
http.createServer(function (req, res) {
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.end('Hello World\n');
}).listen(8083, "0.0.0.0");
console.log('Server running at http://127.0.0.1:8083/');
```

Una vez se tiene el código escrito, se puede ejecutar el unikernel. Es de resaltar el hecho de que no necesitamos descargar e instalar las dependencias de javaScript (Node.js), ya que, mediante el comando de ejecución podemos indicar que dependencias tiene y se descargará un paquete que contiene dichas dependencias necesarias.

```
$ ops pkg load eyberg/node:v14.2.0 -p 8083 -f -n -a hi.js
```

Tras esta ejecución ya se tendrá el unikernel corriendo en el puerto 8083 de nuestra máquina. Si se desean más detalles de como ha sido desarrollado el unikernel paso a paso

con los comandos empleados ver el Anexo A.3.

Hasta ahora, se ha podido comprobar lo simple que es crear un unikernel en nuestro computador local y ejecutarlo. Realmente, para poder realizar la ejecución en local del unikernel Nanos estamos haciendo uso del emulador QEMU, debido a que estamos creando y ejecutando una instancia, de una imagen de disco tipo RAW.

El siguiente paso es implementar estos unikernels de prueba en el CPD del IFCA para evaluar si es posible la futura puesta en marcha de un unikernel más complejo, con la funcionalidad de una aplicación científica en el CPD del IFCA.

3.4.4. Implementación de unikernels de prueba en el CPD del IFCA

Tras la ejecución en local del unikernel de prueba, paso a intentar ejecutarlo en el CPD del IFCA para comprobar su compatibilidad en el cloud, en concreto con la plataforma OpenStack sobre un hipervisor KVM.

Además, como hemos mencionado en algunas ocasiones con anterioridad, no buscamos solo un aumento o un rendimiento similar al de las tecnologías actuales, sino que también sea un proceso sencillo para los usuarios del cloud. Por este motivo, veo conveniente detallar los pasos que se deben realizar para tener una instancia del unikernel ejecutándose sobre OpenStack.

Obtención de los ejecutables

Obtener la imagen que se ejecutará en el cloud es muy sencillo con OPS/Nanos. Al haber hecho previamente el comando que nos ha creado el unikernel con la funcionalidad del servidor web, la imagen ya está creada. Una de las ventajas que presenta este unikernel, es que la misma imagen que se genera para la ejecución local es válida para el despliegue en el cloud. Tan solo deberemos ir al directorio donde OPS almacena las imágenes:

```
$ cd ~/.ops/images
$ file node
    node: DOS/MBR boot sector; partition 1 : ID=0x83,
    active, start-CHS (0x0,1,61), end-CHS (0x1,137,3),
    startsector 123, 24576 sectors; partition 2 : ID=0x83,
    active, start-CHS (0x1,137,4), end-CHS (0xb,10,43),
    startsector 24699, 152689 sectors
```

Como se observa mediante la ejecución del último comando, ya tenemos una imagen lista para subirla al cloud y crear la instancia.

Configuración del entorno en OpenStack

Para poder crear la instancia en OpenStack, lo primero de todo es agregar la imagen al pool de imágenes disponibles del proyecto en el que nos encontramos. En mi caso, esta imagen es privada, por lo que solo es visible para los usuarios del proyecto.

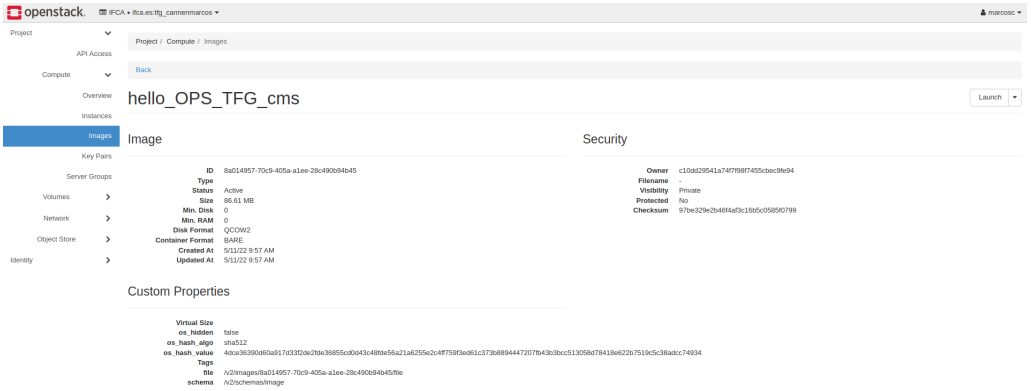

Figura 3.2: Imagen de prueba OPS subida al cloud

Como se puede observar en la figura superior 3.2, la imagen tan solo tiene un tamaño de 86.61 MB. Una vez la imagen está en el cloud, podemos pasar a crear una instancia para ejecutar el unikernel en los servicios cloud del IFCA.

En la imagen inferior 3.3 vemos cómo la instancia no está haciendo uso de ningún sistema operativo como Ubuntu, Debian o CentOS. La imagen base de la instancia es el propio unikernel.

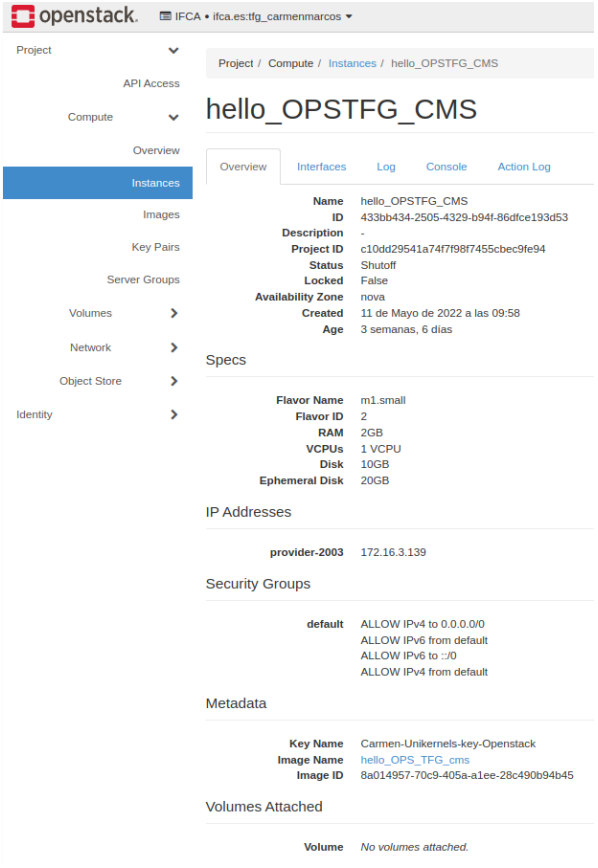

Figura 3.3: Creación de la instancia de prueba en el cloud

Si procedemos a ejecutar la imagen y accedemos a la consola, como en la figura 3.4, podemos observar los logs en los que se indica que el servidor está corriendo en el puerto 8083. Cabe destacar que los logs que aparecen por consola son visibles ya que javascript lo permite mediante la directiva

```
console.log("Mensaje a mostrar");
```

Figura 3.4: Creación de la instancia de prueba en el cloud

Antes de pasar a la creación del unikernel con la aplicación científica, procedo a crear una simple aplicación en local para comprobar que las aplicaciones básicas que pueda realizar un usuario pueden compilarse en un unikernel.

Creando un ‘hello world” en el lenguaje de programación C, compruebo que es posible poner en funcionamiento un unikernel a partir de un ejecutable elaborado por un usuario. Después de compilar el programa, procedo a crear el unikernel a partir del ejecutable obtenido mediante el comando *“ops build helloWorld”*. Se repite el procedimiento previo en OpenStack como se muestras en las figuras 3.2 y 3.3 obteniendo el resultado esperado. La instancia se ejecuta correctamente.

Capítulo 4

Implementación unikernel especializado (CPD IFCA)

En este capítulo se elegirán dos aplicaciones científicas a implementar en un unikernel, así como los benchmarks y se explicará la creación de los tres entornos que serán evaluados por cada aplicación.

4.1. Elección de aplicación científica: OpenQCD

Una de las aplicaciones científicas elegida para ejecutar en el unikernel y realizar el análisis de rendimiento es OpenQCD [44]. Esta aplicación desarrollada por Martin Lüscher, Stefan Schaefer, John Bulava, Isabel Campos y Antonio Rago incluye distintos programas de simulación basados en el algoritmo SMD (Stochastic Molecular Dynamics) para realizar el cálculo numérico del modelo que recrea las interacciones fuertes entre los quarks dentro de los protones y neutrones. Esta teoría es conocida como **Lattice QCD Theory**.

La idea clave es substituir el continuo espacio-tiempo cuadridimensional infinito por un retículo hipercúbico finito. Los campos de quark se definen en los nodos de la red mientras que los campos de gluón viven en los eslabones y la teoría de campos cuánticos se hace corresponder con un sistema estadístico clásico. En efecto, Wilson tomó prestado de la mecánica estadística el desarrollo en una constante de acoplamiento fuerte (equivalente a una expansión para temperatura alta) para demostrar el confinamiento al orden más bajo en 1/g2 [45].

En la aplicación, el retículo hipercúbico finito está modelizado por una matriz de 4 dimensiones de longitudes pares (N0 x N1 x N2 x N3). Cada una de estas componentes son: longitud, altura, profundidad y tiempo.

Por lo tanto, los cálculos que se realizan en la aplicación se basan en multiplicaciones de matrices de cuatro dimensiones, siendo cálculos de uso intensivo de CPU. Estas simulaciones consumen una gran cantidad de recursos de computación, requieren de un desarrollo continuo de los algoritmos y de optimización hardware. Esto se debe a que son cálculos que requieren una paralelización masiva. En caso de que se deseé saber más acerca de la aplicación y el algoritmo que se emplea en las simulaciones, visitar la página web donde, además de obtener más detalles, se puede descargar la aplicación [44].

Para crear el unikernel nos quedaremos con los programas de la aplicación que no requieren de MPI para ejecutar más de un proceso en el mismo nodo, pues, como he explicado con anterioridad, por ahora, los unikernels que hay en producción no permiten esta paralelización a nivel de proceso. Aunque si soportan la ejecución de múltiples unikernels comunicándose entre sí mediante MPI, para aumentar la aumentando así la escalabilidad.

Dentro de estos programas, nos centraremos en los siguientes 3:

- **time1:** Realiza los cálculos correspondientes a la multiplicación de vectores SU(3) con simple precisión.
- **time2:** Realiza los cálculos correspondientes a la multiplicación de vectores SU(3) con doble precisión.
- **time3:** Realiza los cálculos correspondientes a distintas operaciones vectoriales: su3dagxsu3, su3xsu3dag, su3dagxsu3dag, su3xu3alg, su3dagxu3alg, u3algxsu3 y u3algxsu3dag

4.1.1. Elección del benchmark

Para realizar el análisis de rendimiento se ha de medir el tiempo transcurrido en ejecutar los cálculos del programa en las tres implementaciones que se van a realizar (aplicación sobre máquina virtual, aplicación en contenedor y aplicación en unikernel). Para obtener estos tiempos, el propio código de la aplicación ya implementaba el benchmarking. La aplicación nos proporciona el tiempo transcurrido y los Mflops mediante la llamada a la función clock() de la librería time.h, propia de C. La función clock() devuelve el número de ticks de reloj que transcurren entre 2 momentos de la ejecución del programa.

Es de resaltar que los tiempos e instrucciones por segundo medidas son correspondientes al cálculo intensivo de CPU. Previamente, se cargan todos los elementos de las matrices con las que opera, y demás datos necesarios. Se leen los datos de disco a memoria y se cachea. El tiempo que consume en traer los datos a memoria en primera instancia no está está computado dentro de los tiempos que se proporcionan para el análisis de rendimiento.

4.1.2. Creación del unikernel con aplicación en un entorno de computación científica e implementación en el cloud

Para crear el unikernel de la aplicación científica mencionada anteriormente, partimos del código fuente de la aplicación. Para obtener los programas de la aplicación que vamos a probar tenemos que ir al directorio de las integraciones no MPI. Una vez situados en el directorio correcto, tan solo tenemos que Makefile para generar los ejecutables. A continuación se ha de crear el unikernel del mismo modo que hicimos con la aplicación de prueba:

```
$ ops build time3
```

Tras la ejecución de este último comando por terminal ya tenemos la imagen unikernel creada en el directorio correspondiente /.ops/images. Para implementarlo en el cloud tenemos que seguir el mismo procedimiento que se realizó en el apartado 3.4.4.

4.1.3. Problemas encontrados en la ejecución

Durante la ejecución de las aplicaciones en los tres modos seleccionados (máquina virtual, contenedor y unikernel) me encontré con un problema que ralentizo mi trabajo hasta que pude encontrar la solución. Cuando ejecutaba el unikernel con la funcionalidad seleccionada en mi computador para comprobar que la imagen del unikernel se había creado correctamente no se observó problema alguno. Sin embargo, al ejecutarlo sobre OpenStack la instancia se iniciaba correctamente hasta que, transcurridos escasos segundos, se apagaba sin reportar los datos de rendimiento por consola serie, ni en los logs.

En un principio pensé que se podía deber a que, pese a que se podía estar ejecutando el unikernel correctamente, los resultados no se mostraban ya que la instancia se apagaba inmediatamente. Por lo tanto decidí realizar modificaciones en el código fuente, pese a que este no era el objetivo. Dichas modificaciones pasaron por crear sockets dentro de la aplicación para recogerlo desde otra máquina que estuviese en la misma red, ejecutar la instrucción *fflush()* para forzar al programa a que mostrase la salida*stdout* por consola, ejecutar *sleeps()*, mandarme los resultados por correos desde la propia aplicación, etc.

Además de asistir a distintos workshops y seminarios de la herramienta OPS pensando que el problema radicaba en la herramienta,me puse en contacto con el CEO de la empresa que desarrolla el unikernel empleado, OPS/Nanos. Tras varios mensajes me confirmó que ese problema no se debía al unikernel en sí, sino a alguna limitación de la aplicación.

Finalmente, revisando el Makefile de la aplicación reparé en que se estaba haciendo uso de un módulo de operaciones vectoriales (AVX: Extensiones Vectoriales Avanzadas) para optimizar algunos de los cálculos. Este tipo de operaciones, pese a que la arquitectura de los servidores físicos las soportan, están deshabilitadas en el cloud. Esto se debe a que cada servidor físico puede estar albergando más de una máquina virtual, pero solo 1 puede hacer uso del módulo AVX, por lo que se deshabilita su uso en el cloud.

Por lo tanto, cuando iniciaba la instancia, se realizaban los cálculos pertinentes, pero al llegar a las operaciones vectoriales paraba su ejecución. La solución al problema ha sido inhabilitar el uso de estas extensiones vectoriales en el Makefile de la aplicación. De este modo, ya se puede ejecutar el unikernel de manera correcta.

Cabe destacar que todas las ejecuciones, tanto las ejecutadas sobre máquina virtual, contenedor o unikernel se han realizado sin hacer uso del módulo de extensión vectorial.

4.2. Elección de aplicación científica: Benchmark 435.Gromacs

El benchmark 435.gromacs es creado a partir de GROMACS, una serie de programas que realizan operaciones de dinámica molecular como puede ser la simulación de las ecuaciones Newtonianas del movimiento para sistemas con cientos de millones de partículas [46].

El uso que se le da a GROMACS por parte de la comunidad científica varía en función del objetivo que se persiga. Muchos grupos de investigación usan la aplicación para la investigación en sistemas no biológicos, pues es una herramienta muy rápida calculando interacciones no ligadas. Sin embargo, GROMACS fue diseñado para las simulaciones de las interacciones de moléculas bioquímicas como las proteínas y los lípidos.

Este benchmark realiza una simulación de la proteína Lisozima en una solución de agua e iones. Mediante la simulación del movimiento de los átomos de estas estructuras se puede llegar a comprender la dinámica de las proteínas, e incluso poder predecir estructuras de nuevas proteínas [46]. A nivel de programación, la simulación se basa en bucles dónde se calculan las cierto tipo de interacciones entre átomos que están a menos de 1.4 nm.

4.2.1. Elección del benchmark

Puesto que la propia aplicación es un benchmark, ciertas métricas del sistema, como la energía potencial media calculada, la temperatura media del sistema y el número de operaciones de punto flotante ejecutadas son mostradas al finalizar la ejecución. Sin embargo, puesto que no muestra el tiempo que tarda en ejecutarse, se ha de recabar dicha métrica.

En el caso de la ejecución sobre una máquina virtual y sobre el contenedor se hace uso del comando time. Este comando nos muestra el tiempo que tarda en ejecutar el programa una vez haya finalizado el comando de ejecución. Por ejemplo:

```
$ time ./gromacs_base.x64_gcc_cern -deffnm gromacs -nice 0
```

En el caso del unikernel, acudimos a la plataforma OpenStack una vez finaliza la ejecución de la instancia para ver el tiempo que ha estado ejecutando.

4.2.2. Creación del unikernel con aplicación en un entorno de computación científica e implementación en el cloud

Para crear el unikernel, se ha tenido que crear un fichero de configuración dónde se ha indicado los argumentos de entrada y archivos necesarios para la ejecución. El resto del procedimiento es el mismo que para los unikernels creados anteriormente.

```
"Files": ["gromacs.tpr"],
"Args": ["gromacs", "-deffnm","gromacs","-nice","0","2"]
```

4.3. Creación de los contenedores Docker con las aplicaciones científicas

Uno de los modos de ejecución con el que se comparan los tiempos obtenidos por el unikernel, es mediante la ejecución de la aplicación en un contenedor. Para poder ejecutar las dos aplicaciones en un contenedor he creado una imagen docker para cada aplicación, para ejecutarla sobre la máquina virtual. El dockerfile mediante el que he creado la imagen openQCD está disponible en el Anexo **??**.

Capítulo 5

Resultados obtenidos y análisis de rendimiento

A continuación se muestran los resultados obtenidos tras ejecutar la aplicación científica OpenQCD y el benchmark de GROMACS en 3 entornos distintos: Ejecución en máquina virtual, ejecución en un contenedor Docker dentro de una máquina virtual y ejecución en un unikernel directamente sobre el cloud, sin necesidad de máquina virtual o sistema operativo anfitrión.

Para obtener los resultados se ha realizado la media de cinco ejecuciones por cada tiempo mostrado en la tabla en caso de OpenQCD y 4 ejecuciones en el caso del benchmark de GROMACS. Previo a cada medida, se han realizado 5 ejecuciones de calentamiento. La frecuencia de las CPU se fija a 2300MHz. Sin embargo, monitorizando las ejecuciones de prueba se observa que no se fija a la frecuencia especificada, sino que toma dicha cifra de manera orientativa.

Para realizar las gráficas se han calculado los errores absolutos de cada una de las medidas para poder discriminar los resultado que se salgan de la tendencia. En caso de que haya alguna medida que tenga mucha variabilidad se tendrá en cuenta en el análisis dicha variabilidad.

5.1. Aplicación científica OpenQCD

5.1.1. Ejecución con tamaños de matriz original, 64 elementos

En la figura 5.1 se muestran los datos obtenidos tras realizar la media de las cinco ejecuciones en cada caso, siendo sucesos independientes entre sí.

	TAMAÑO MATRIZ ORIGINAL			
	Tiempo transcurrido en ns			
		VM	container	unikernel
	time1	11514	10868	15392
	time2	11882	11296	16411
time3	su3dagxsu3	43566	43095	42571
	su3xsu3dag	44079	46566	46720
	su3dagxsu3dag	44524	46382	45711
	su3xu3alg	43893	44611	44496
	su3dagxu3alg	44247	44663	43971
	u3algxsu3	45775	47440	44853
	u3algxsu3dag	45534	47716	45228

Figura 5.1: Tabla con los tiempos de ejecución, en ns, obtenidos con las matrices originales de 64 elementos.

Se puede observar cómo en las ejecuciones con tamaño de matriz 64 los tiempos que se obtienen en los tres entornos son comparables en la mayoría de casos. En las ejecuciones de *time1* y *time2* se observa un menor rendimiento en la ejecución del unikernel, aumentando el tiempo de ejecución en el peor de los casos un 38 % frente a la ejecución sobre la máquina virtual.

Para analizar las ejecuciones de time3, ha sido elaborada una gráfica más detallada 5.2, en base a los datos recogidos en la tabla5.1.

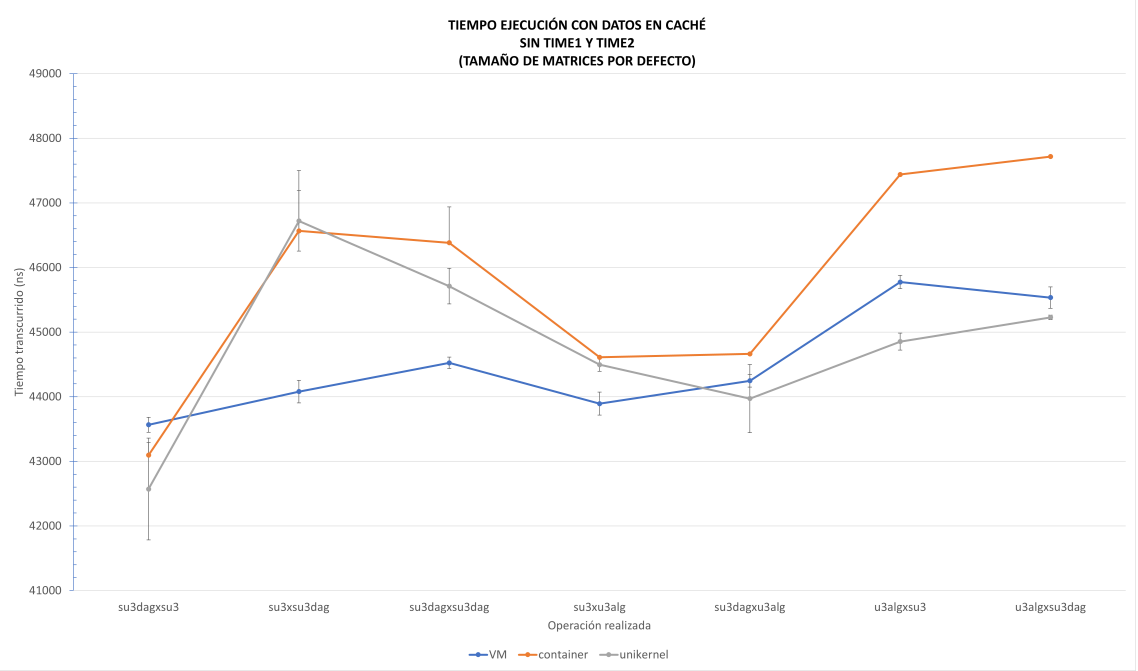

Figura 5.2: Gráfica tiempos de ejecución de operaciones de time3 con tamaño de matriz de 64 elementos

Como se muestra en la gráfica 5.2, las ejecuciones mediante los tres modos arrojan tiempos comparables. Asimismo, se puede afirmar que las ejecuciones sobre la máquina virtual proporcionan unos tiempos de ejecución ligeramente menores frente a las ejecuciones sobre el contenedor y el unikernel. Por otro lado, el unikernel se comporta mejor en los cálculos *u3algxsu3* y *u3algxsu3dag* con errores casi despreciables.

Cabe destacar la alta variabilidad que presentan ciertas ejecuciones de time3 en el unikernel. Esto demuestra que los tiempos obtenidos en dichas ejecuciones han sido más dispares e irregulares, haciendo que no sean resultados tan certeros.

5.1.2. Ejecución con tamaños de matriz de 4096 elementos

	TAMAÑO MATRIZ 4096 ELEMENTOS			
	Tiempo transcurrido en ns			
	nsec	VM	container	unikernel
	time1	11533	10876	11409
	time2	11933	11257	13172
time3	su3dagxsu3	43769	43154	47478
	su3xsu3dag	44200	46647	54933
	su3dagxsu3dag	44537	46858	51930
	su3xu3alg	44043	44957	50415
	su3dagxu3alg	44255	44681	50134
	u3algxsu3	45956	48125	50818
	u3algxsu3dag	45580	47484	52060

Figura 5.3: Tabla con los tiempos de ejecución, en ns, obtenidos con matrices de 4096 elementos

En la figura 5.4, realizada en base a los resultados obtenidos, mostrados en la tabla 5.3, se observan los tiempos de ejecución correspondiente a las operaciones t1, t2 y t3 con tamaños de 4096 elementos por matriz.

En los tiempos obtenidos en time1 y time 2 se mantiene la tendencia a obtener resultados similares en los tres entornos, siendo ligeramente menor el rendimiento de los unikernels. Aunque la diferencia es casi despreciable. La mayor diferencia en t1 y t2 se observar en la ejecución de time2 en la máquina virtual respecto al unikernel, con un rendimiento inferior por parte del unikernel de un 10,4 %.

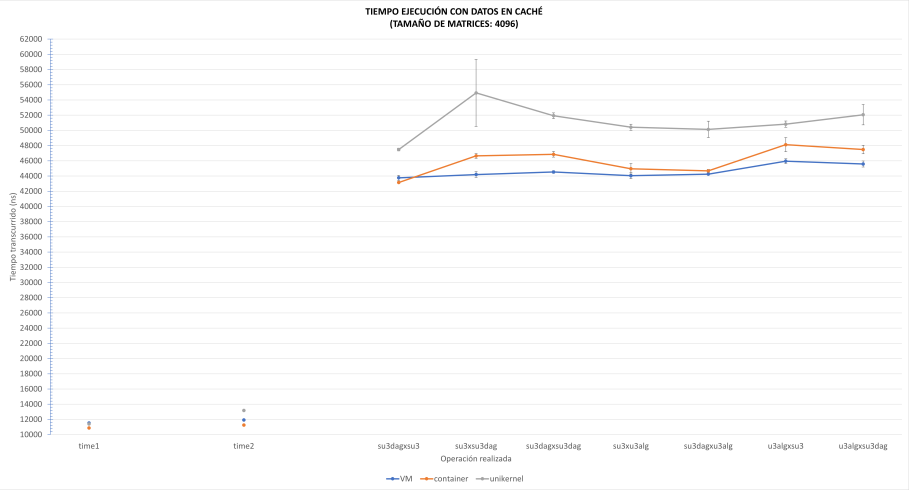

Figura 5.4: Gráfica tiempos de ejecución con tamaño de matriz de 4096 elementos

De modo análogo al análisis de la ejecución con tamaños de matriz de 64 elementos, se procede a realizar el análisis de las operaciones de t3 mediante una gráfica más detallada de estas ejecuciones.

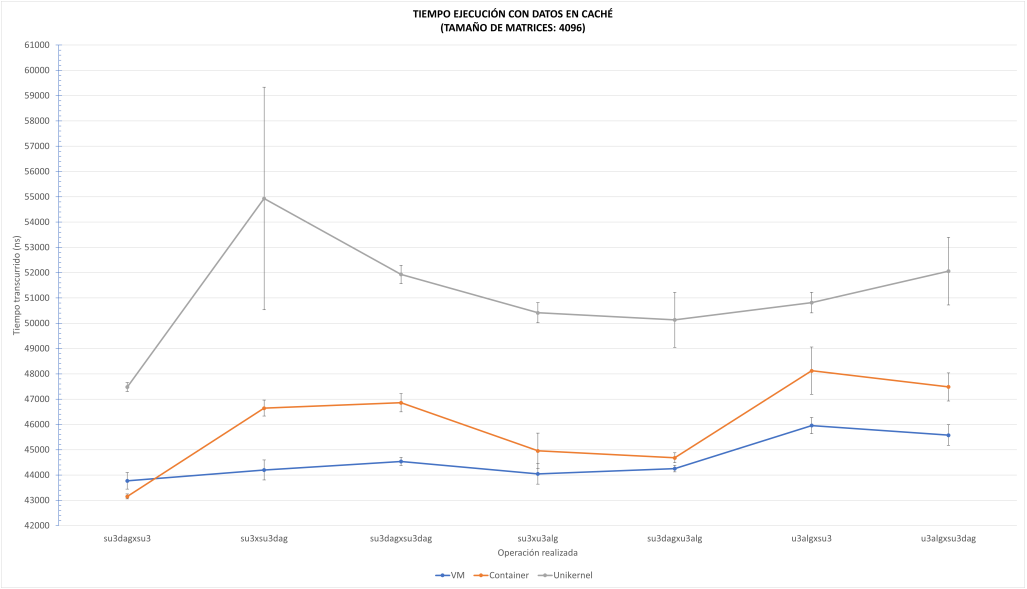

Figura 5.5: Gráfica tiempos de ejecución de operaciones de time3 con tamaño de matriz 4096

Acorde a los resultados que muestra la gráfica 5.5, los tiempos de ejecución de la aplicación científica en el unikernel incrementan a medida que se incrementa el tamaño de la matriz. Por otro lado, el error relativo en dichas ejecuciones es bastante alto. Sin embargo, aunque tomemos como certero el valor más favorable dentro del error, sigue siendo un incremento en el tiempo de ejecución bastante significativo.

Por otro lado, seguimos observando cómo los tiempos obtenidos en la máquina virtual son mejores frente al contenedor, continuando con lo observado en los datos obtenidos con matrices más pequeñas.

El hecho de que los unikernels consigan menos rendimiento en esta ejecución se puede deber al acceso a la caché para traer los datos con los que realizar las operaciones. Otro posible motivo es el uso de un unikernel de tipo legacy en lugar de uno clean-state. Los unikernels de tipo legacy intentan emular el funcionamiento de la aplicación sobre un sistema operativo. Para ello, implementan un grupo reducido de interfaces POSIX para asegurar que se pueda ejecutar el código sin modificar. Mientras tanto, un unikernel de tipo clean-state no intenta emular un sistema operativo clásico. Por lo tanto, el uso de las interfaces POSIX puede ser una de las causas del overhead que presentan las ejecuciones de los unikernels.

Resaltar que, hasta que no se haga un profiling no se va a poder determinar la causa del aumento de tiempo de ejecución. En cuanto a las condiciones de compilación de cada carga, todas las ejecuciones han sido realizada con las mismas condiciones de compilación.

5.1.3. Comparación de las ejecuciones con los dos tamaños seleccionados

Llama la atención la comparación de los datos obtenidos en las ejecuciones con 4096 elementos por dimensión de cada arista del hipercubo mostrados en la figura 5.3, frente a los correspondientes resultados de la ejecución de matrices con 64 elementos. No hay una diferencia significativa de rendimiento en proporción al número de cálculos extra que se han de realizar al aumentar sesenta y cuatro veces su tamaño, lo que podría indicar que hay mucho overhead de inicialización comparado con el cómputo. No obstante, tras haber analizado el código fuente de la aplicación, he descartado esta posibilidad. Fue descartada debido a que en dicho código se realiza la medición justo antes de realizar las operaciones de multiplicación correspondientes, y tras finalizarlas. Por lo tanto, solo se está midiendo el tiempo de la región crítica.

Posteriormente, contacté con una de las personas que han desarrollado código, por lo que he podido dar con el motivo de estos resultados tan llamativos. Los datos correspondientes a los hipercubos con los que se realizan los cálculos son cargados en caché antes de realizar los cálculos. En caso de que los hipercubos con las dimensiones seleccionadas no entren en caché, el programa ignora dichos datos que se han quedado sin espacio en caché. Por lo tanto, cabe la posibilidad de que 4096 elementos por arista sea un número extremadamente alto, propiciando que no entren todos los datos en caché. Debido esta limitación, el hecho de probar a realizar modificaciones en el número de elementos por arista puede dejar de tener interés en este caso de estudio. Mediante la ejecución de esta aplicación lo que se está midiendo es la capacidad de cómputo de la CPU para obtener un indicador de un rendimiento máximo.

5.2. Aplicación científica Gromacs Benchmark

A continuación, se presentan los resultados obtenidos tras la implementación del Benchmark de Gromacs en los tres distintos modos de ejecución junto al análisis correspondiente.

5.2.1. Ejecución del benchmark

Los tiempos de ejecución obtenidos mediante este benchmark, reflejados en la figura 5.6, muestran que el unikernel sigue siendo el modo de ejecución que más tarde en ejecutar, y por tanto el menos eficiente de todos. Puesto que los tiempos de ejecución en los tres casos superan los siete minutos, la diferencia de tiempos no se puede achacar a los tiempos de inicialización, como podía ser en el caso de la ejecución de OpenQCD. Por lo que podemos afirmar que los datos obtenidos son fiables.

Por otro lado, la gráfica 5.6, además de ilustrar los resultados obtenidos, podemos apreciar que la variabilidad de los tiempos obtenidos son prácticamente despreciables. También podemos afirmar que las diferencias de tiempo entre las ejecuciones en la máquina virtual y el contenedor son muy bajas, demostrando porque el contenedor es una de las tecnologías

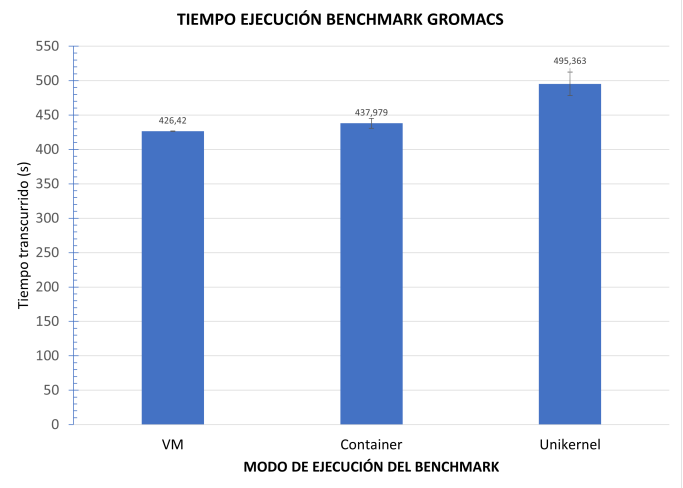

Figura 5.6: Gráfica tiempos de ejecución benchmark GROMACS

más utilizadas para la ejecución de aplicaciones científicas. Se consigue un rendimiento muy similar al de la máquina virtual, además de las ventajas que presenta en portabilidad.

Capítulo 6

Conclusiones obtenidas

Es este capítulo se dará respuesta a las preguntas planteadas en la introducción, se explicarán las conclusiones obtenidas y se propondrán futuros trabajos para seguir explorando el rendimiento y la usabilidad de los unikernels en el ámbito científico.

6.1. Respuesta a las preguntas de investigación

En la introducción de este trabajo fin de grado se proponen una serie de preguntas de investigación, que a continuación paso a responder en vista a los resultados obtenidos y análisis realizados.

- **¿Es más fácil desplegar una aplicación científica en un CPD mediante un unikernel frente a un contenedor como pueden ser los desarrollados por Docker?**

  En base al trabajo realizado en este TFG, es más sencillo desplegar un unikernel que un contenedor en un CPD. Se puede crear en un cualquier computador con tan solo un binario, y subir la imagen a la plataforma cloud correspondiente. Mientras, el contenedor necesita la creación previa de la máquina virtual, la creación del dockerfile (siendo más compleja que la creación del unikernel), y crear la imagen.

- **¿Mediante el uso de los unikernels se consigue alguna ganancia a nivel de rendimiento?**

  En base al análisis de rendimiento realizado, si estamos haciendo uso de una aplicación que demanda mucho uso de CPU y accesos continuos a memoria caché, no se consiguen ganancias de rendimiento en lo que a tiempo de ejecución se refiere. Si hay ganancias a nivel de recursos en la infraestructura cloud, ya que cada instancia e imagen necesita menos recursos físicos.

- **¿Es posible, a corto o medio plazo, reemplazar los contenedores por los unikernels en un entorno de computación distribuida en un entorno científico?**

  En base al trabajo realizado a lo largo de TFG, es posible que en un futuro sea una alterativa factible a los contenedores y máquinas virtuales, pero no a corto plazo.

  Sin embargo, el hecho de que los unikernels que están en producción a día de hoy no estén diseñados para trabajar con paralelismo a nivel de proceso, limita esta evolución. A día de hoy, HermitCore es el framework que en un futuro nos puede llegar a acercar a este paradigma de programación dentro de los unikernels, ya que, pese a no estar en producción actualmente, se está trabajando para que permita el uso de MPI.

Finalmente, se tendrá que evaluar, si no mejora el rendimiento de los unikernels en este tipo de aplicaciones, que es lo que más compensa. Si un menor rendimiento a cambio de mayor portabilidad y menor uso de recursos físicos del cloud, o lo contrario.

6.2. Conclusiones

A la vista de los resultados obtenidos y el análisis realizado, los unikernels no proporcionan un mejor rendimiento en aplicaciones científicas del tipo que hemos empleado. Realmente, su implementación supone un rendimiento menor, pero hay que evaluar las contrapartidas que nos puede ofrecer:

- La facilidad de uso y creación que brindan los unikernels es bastante mayor que la de un contenedor. En el framework de unikernels que hemos usado para el análisis, OPS/Nanos, podemos crear el unikernel en local en base a un binario ejecutable con tan solo un comando. Además, este se puede ejecutar en cualquier plataforma o dispositivo sin necesidad de software adicional.

- El unikernel, al igual que los contenedores como Docker, nos proporcionan una portabilidad de que una máquina virtual no es capaz de ofrecer. Por otro lado, los unikernels nos permiten ejecutar una imagen en una cluster de Slurm sin necesidad de software adicional, como es el caso de los contenedores. Estos últimos precisan de tecnología como UDocker para poder ejecutar sobre un cluster de Slurm.

- El uso del unikernel supone una reducción de tamaño en la imagen y la instancia a ejecutar, por lo que disminuye la memoria física que se necesita en el cloud para almacenar la imagen, así como en su ejecución. De este modo, las infraestructuras físicas cloud pueden albergar más instancias e imágenes por servidor físico, maximizando así los recursos del cloud.

6.3. Posibles futuras líneas de trabajo e investigación

Tras finalizar este trabajo fin de grado, queda pendiente evaluar como la orquestación de más de un unikernel puede suplir la falta de soporte de MPI.

Otro aspecto interesante es determinar el origen del aumento de tiempo de ejecución en el caso de uso de unikernels, el cuello de botella. Esto se puede realizar tratando de crear unikernels que solo hagan uso exhaustivo de un elemento del computador: Aplicaciones que solo hagan uso de CPU, otras que solo accedan a memoria o programas que accedan a caché continuamente sin uso intensivo de CPU u otros elementos del computador.
Finalmente, aunque se sale del ámbito que hemos analizado, una posible continuación podría ser un análisis de seguridad de los unikernels frente a los contenedores, ya que al

reducir la superficie del programa, por consiguiente también se reduce la superficie de ataque.

Bibliografía

[1] S. S. Tadesse, C. F. Chiasserini, and F. Malandrino, "Characterizing the power cost of virtualization environments," *Transactions on Emerging Telecommunications Technologies*, vol. 29, no. 8, p. e3462, 2018.

[2] B. P. Tholeti, "Dive into the Xen hypervisor - IBM." `https://developer.ibm.com/articles/cl-hypervisorcompare-xen/`, Sep 2011.

[3] B. P. Tholeti, "Dive into the KVM hypervisor - IBM." `https://developer.ibm.com/articles/cl-hypervisorcompare-kvm/`, Sep 2011.

[4] C. Koppelt, "Technical background of mirageos." `https://mirage.io/docs/technical-background`, Nov 2013.

[5] D. Duggan, "Operating system structure."

[6] R. Matarneh, "Multi microkernel operating systems for multi-core processors," *Journal of Computer Science*, vol. 5, 07 2009.

[7] S. Lankes, S. Pickartz, and J. Breitbart, Jun 2016.

[8] B. Rotibi, C. Howard, and M. Bell, "Containers and kubernetes market dynamics report 2021 - red hat." `https://www.redhat.com/rhdc/managed-files/cl-state-kubernetes-security-report-ebook-f29117-202106-en_0.pdf`, Jun 2021.

[9] H. Tazaki, R. Nakamura, and Y. Sekiya, "Library operating system with mainline linux network stack," *Proceedings of netdev*, 2015.

[10] A. Madhavapeddy and D. J. Scott, "Unikernels: Rise of the virtual library operating system," *ACMQueue*, vol. 11, no. 11, p. 30–44, 2013.

[11] I. C. Education, "Virtualization complete guide."

[12] S. Crosby and D. Brown, "The virtualization reality," *ACMQueue*, vol. 4, no. 10, p. 34–41, 2006.

[13] P. Denning, "Anecdotes [virtual machines]," *IEEE Annals of the History of Computing*, vol. 23, no. 3, p. 73, 2001.

[14] R. A. Nelson and D. Sayre, "Virtual machines, operating systems accomplishment." `https://researcher.watson.ibm.com/researcher/view_group.php?id=7032`, Jul 2016.

[15] S. M. Jain, *Linux containers and virtualization: A kernel perspective*. APRESS, 2020.

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

[17] B. Bitner and S. Greenlee, "Z vm - a brief review of its 40 year history." https://www.vm.ibm.com/vm40hist.pdf.

[18] Xen.org, "Xen overview/es." https://wiki.xenproject.org/wiki/Xen_Overview/es.

[19] R. Hat, "Kernel virtual machine." https://www.linux-kvm.org/page/Main_Page.

[20] G. Wolf, "Virtualización: Contenedores, o virtualización a nivel de sistema operativo." http://sistop.gwolf.org/html/A2_virtualizacion.html.

[21] "Cloud Native Computing Foundation." https://www.cncf.io/, May 2022.

[22] "Annual survey 2021 - cloud native computing foundation." https://www.cncf.io/wp-content/uploads/2022/02/CNCF-AR_FINAL-edits-15.2.21.pdf, Feb 2022.

[23] C. Abraham. https://github.com/cncf/surveys/blob/main/cloudnative/Cloud_Native_Survey_2021-Part_1.xlsx.

[24] "7094 data processing system."

[25] H. Tazaki, "An introduction of library operating system for linux (libos)." https://lwn.net/Articles/637658/, Mar 2015.

[26] H. Tazaki, "Libos-nuse/net-next-nuse: Network stack in userspace." https://github.com/libos-nuse/net-next-nuse.

[27] J. Talbot, P. Pikula, C. Sweetmore, S. Rowe, H. Hindy, C. Tachtatzis, R. Atkinson, and X. Bellekens, "A security perspective on unikernels," in *2020 International Conference on Cyber Security and Protection of Digital Services (Cyber Security)*, pp. 1–7, 2020.

[28] D. B. Golub, D. P. Julin, R. F. Rashid, R. P. Draves, R. W. Dean, A. Forin, J. Barrera, H. Tokuda, G. Malan, and D. Bohman, "Microkernel operating system architecture and mach," in *In Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pp. 11–30, 1992.

[29] "Open source cloud computing infrastructure." https://www.openstack.org/.

[30] "Open source cloud computing platform software: Components." https://www.openstack.org/software/project-navigator/openstack-components#openstack-services.

[31] "Mirageos: Documentation." https://mirage.io/docs.

[32] B. Gerofi, Y. Ishikawa, R. Riesen, R. W. Wisniewski, Y. Park, and B. Rosenburg, "A multi-kernel survey for high-performance computing," *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers*, 2016.

[33] IncludeOS, "IncludeOS Technology." `https://www.includeos.org/technology.html`, 2022.

[34] "MUSL LIBC." `https://www.musl-libc.org/`.

[35] "Documentation for includeos.readthedocs.io." `https://includeos.readthedocs.io/_/downloads/en/latest/pdf/`.

[36] "Toro kernel." `https://torokernel.io/`.

[37] "Osv: The operating system designed for the cloud." `https://osv.io/`.

[38] I. Eyberg, "Easily build and run unikernels," *OPS*, 2022.

[39] Cloudius-Systems, "Cloudius-systems/capstan: Capstan, a tool for packaging and running your application on osv. http://osv.io/capstan." `https://github.com/cloudius-systems/capstan/`.

[40] Nanovms, "Nanovms/ops-examples: A repository of basic and advanced examples using ops." `https://github.com/nanovms/ops-examples`.

[41] OCamlPro, "Ocaml package manager." `https://opam.ocaml.org/`.

[42] "Top programming languages for data scientists in 2022." `https://www.datacamp.com/blog/top-programming-languages-for-data-scientists-in-2022`, Jan 2022.

[43] Solo5, "Solo5/solo5: A sandboxed execution environment for unikernels." `https://github.com/Solo5/solo5`.

[44] M. Lüscher, S. Schaefer, J. Bulava, I. Campos, and A. Rago, "openqcd - simulation programs for lattice qcd." `https://luscher.web.cern.ch/luscher/openQCD/index.html`, May 2022.

[45] "Lattice qcd, el estudio numérico de la interacción fuerte," Jun 2016.

[46] E. Lindahl and D. van der Spoel, "435.gromacs spec cpu2006 benchmark description," Apr 2008.