UC | Facultad de Ciencias

Análisis de la viabilidad de un puesto de laboratorio docente RISC-V basado en hipervisores

Analysis of the feasibility of a hypervisor-based RISC-V teaching laboratory setup

Trabajo de Fin de Máster
para acceder al

MÁSTER EN INGENIERÍA INFORMÁTICA

Autor: Saúl Fernández Tobías

Directores: Pablo Fuentes, Cristóbal Camarero

Julio - 2024

Resumen.

En las asignaturas del área de *Estructura y Organización de Computadores* del Grado en Ingeniería Informática de la Universidad de Cantabria se enseña el funcionamiento y composición de un computador con un enfoque eminentemente práctico, mediante el desarrollo de programas sencillos y drivers de dispositivos de Entrada/Salida en la arquitectura ARM.

Los alumnos utilizan para esta tarea placas RaspberryPi, que resultan accesibles, en conjunto con el sistema operativo RISC OS, el cual presenta una interfaz amigable. Sin embargo, en busca de fomentar la adopción de hardware libre en el que haya más transparencia y menos dependencias con proveedores hace que resulte interesante realizar una migración del laboratorio a la arquitectura RISC-V, de código abierto y modular. Como RISC OS no está disponible para la arquitectura RISC-V, y dado que no se espera su llegada dentro de un futuro previsible, será necesario idear soluciones alternativas que mantengan los beneficios de RISC OS: una interfaz amigable,sumado a las bajas restricciones para acceder a regiones de memoria privilegiadas.

En trabajos anteriores se ha observado que el ecosistema hardware de RISC-V está aún inmaduro, con baja disponibilidad de placas y de bajo coste y alto grado de documentación. Sin embargo consideramos que, dada la creciente comunidad que presenta la arquitectura junto con el atractivo comercial que tiene al no requerir de licencias, fomentará su adopción en el mercado, lo que conllevará una mayor accesibilidad a dichos recursos.

En este trabajo se hace un análisis del estado del arte de las herramientas y sistemas operativos disponibles para la arquitectura, analizando de qué modo pueden ser utilizados para crear un laboratorio que permita la ejecución de código que acceda a elementos hardware sin restricciones, a la vez que se presente una interfaz amigable para el desarrollo de las prácticas por parte del alumnado.

Para conseguir estos objetivos, se plantea un enfoque del laboratorio basado en el uso de hipervisores, de modo que se permita la coexistencia de un sistema operativo moderno, que facilite el entorno de desarrollo del código, con un sistema operativo más simple que permita la ejecución de código con bajas restricciones, simplificando el proceso de depuración.

Con el fin de no limitar el desarrollo del análisis a un hardware en específico y dada la baja disponibilidad y poca accesibilidad de las placas en la actualidad, durante el desarrollo de este trabajo se emplea emulación. De este modo se podrá recrear un entorno de desarrollo genérico que no esté atado a las limitaciones de un hardware específico. Esto a su vez facilitará la transición a futuras implementaciones de la arquitectura, ya que se evitarán las dependencias con las placas actuales.

Durante el desarrollo del proyecto se han definido además dos posibles aproximaciones a la hora de establecer las tareas que deberá realizar cada uno de los sistemas operativos involucrados en el despliegue. Por un lado, se plantea la posibilidad de que un sistema operativo sirva únicamente de interfaz mientras que el otro sistema se encarga de contener al núcleo del depurador, controlando la ejecución del código del alumno. Por otro lado, se plantea otra posible aproximación en la que el sistema menos restrictivo simplemente sirva como entorno de ejecución para el código del alumno, mientras que el sistema que sirve de interfaz, a su vez, se encargue de depurar la ejecución del sistema operativo ligero y, a su vez, el código del alumno.

Por último, se hará una valoración de los hipervisores y sistemas operativos analizados, escogiendo de entre ellos como posibles candidatos aquellos que mejor se adapten a los requisitos de este proyecto. Durante esta fase del proyecto se han encontrado opciones prometedoras, entre las que destacan los hipervisores BAO y KVM junto con el sistema operativo Mini-riscv-os. Además, las pruebas desarrolladas con estos candidatos evidencian de forma preliminar la viabilidad de la migración mencionada.

Palabras clave.

RISC-V, Hipervisores, Docencia, Depurador, Estado del arte.

Abstract.

In the courses of the *Structure and Organization of Computers* area from the Computer Engineering degree at the University of Cantabria, the operation and composition of a computer are taught with a primarily practical approach. This is done through the development of simple programs and Input/Output device drivers in the ARM architecture.

For this task, students use Raspberry Pi boards, which are accessible, along with the RISC OS operating system, which features a user-friendly interface. However, in an effort to promote the adoption of open hardware with more transparency and fewer dependencies on vendors, it becomes interesting to migrate the laboratory to the open-source and modular RISC-V architecture. Since RISC OS is not available for the RISC-V architecture and its arrival is not expected in the foreseeable future, it will be necessary to devise alternative solutions that maintain the benefits of RISC OS: a user-friendly interface combined with low restrictions on accessing privileged memory regions.

In previous works, it has been observed that the RISC-V hardware ecosystem is still immature, with low availability of boards and at an inaccessible price. However, we believe that given the growing community around the architecture and its commercial appeal due to not requiring licenses, its adoption in the market will be encouraged, leading to greater accessibility to these resources.

This work provides an analysis of the state of the art of tools and operating systems available for the architecture, examining how they can be used to create a laboratory that allows code execution to access hardware elements without restrictions, while also presenting a user-friendly interface for students to develop their practical exercises.

To achieve these objectives, a laboratory approach based on the use of hypervisors is proposed, allowing the coexistence of a modern operating system, which facilitates the code development environment, with a simpler operating system that permits code execution with low restrictions, thus simplifying the debugging process.

In order not to limit the analysis to specific hardware and given the low availability and limited accessibility of the boards currently, emulation is used during the development of this project. This way, a generic development environment can be recreated that is not tied to the limitations of specific hardware. This will also facilitate the transition to future implementations of the architecture, as dependencies on current boards will be avoided.

During the development of the project, two possible approaches have been defined for establishing the tasks that each of the operating systems involved in the deployment will perform. On the one hand, there is the possibility that one operating system serves solely as an interface while the other system contains the debugger's core, controlling the execution of the student's code. On the other hand, the less restrictive system simply serves as an execution environment for the student's code, while the system serving as the interface also handles debugging the execution of the lightweight operating system and, in turn, the student's code.

Finally, an evaluation of the analyzed hypervisors and operating systems will be conducted, selecting as possible candidates those that best meet the requirements of this project. During this phase of the project, promising options have been found, notably the BAO and KVM hypervisors along with the Miniriscv-os operating system. Additionally, preliminary tests developed with these candidates demonstrate the feasibility of the aforementioned migration.

Key words.

RISC-V, Hypervisors, Teaching, Debugger, State of the Art.

Índice

1. Introducción **9**
1.1. Motivación . . . 9
1.2. Objetivos . . . 10
1.2.1. Desarrollo del trabajo . . . 11

2. Planteamiento **13**
2.1. Entorno de desarrollo . . . 13
2.1.1. Toolchains . . . 15
2.2. Estrategias de trabajo . . . 16
2.2.1. Interfaz y núcleo . . . 17
2.2.2. Depuración completa del SO . . . 18

3. Hipervisores **20**
3.1. XtratuM . . . 20
3.2. Xen . . . 21
3.3. Jailhouse . . . 21
3.4. Diosix . . . 22
3.5. Xvisor . . . 24
3.6. RVirt . . . 25
3.7. KVM . . . 26
3.8. Bao . . . 27

4. Sistemas Operativos Ligeros **29**
4.1. Egos2000 . . . 29
4.2. FreeRTOS . . . 30
4.3. Xv6 . . . 30
4.4. Mini RISC-V OS . . . 30

5. Pruebas desarrolladas **33**
5.1. Prueba de uso con KVM . . . 33
5.2. Prueba de uso con Bao . . . 37

6. Conclusiones **45**
6.1. Trabajo futuro . . . 46

Bibliografía **48**

A. Mensajes de error **51**

B. Instalación de Qemu 4.X **52**

C. Instalación del nuevo kernel de Linux **53**

Índice de figuras

1. Diagrama de Gantt del proyecto. 12
2. Desarrollo con hipervisores tipo 1. 17
3. Desarrollo con hipervisores tipo 2. 18
4. Depurado básico con GDB. 34
5. Primera prueba de comunicación entre sistemas operativos guest en BAO. 41
6. Segunda prueba de comunicación entre sistemas operativos guest en BAO. 41
7. Depurado básico con GDB. 53
8. Vista principal del menú de configuración del kernel de Linux. 54
9. Activación del módulo de KVM en el kernel de Linux. 54

1. Introducción

En este capítulo se van a exponer las causas que motivan la realización de este proyecto. Se detalla el modelo de trabajo que se sigue en la parte práctica de las asignaturas pertenecientes a la mención de Ingeniería de Computadores, especialmente aquellas en las que se ejercita la programación en ensamblador a través de dispositivos basados en la arquitectura ARM. Asimismo, se muestran los beneficios que tiene una adaptación del entorno actual a una arquitectura *open source* y se detallarán los objetivos planteados en este proyecto.

1.1. Motivación

Durante su paso por la *Universidad de Cantabria*, los estudiantes de Ingeniería Informática deben adquirir conocimientos relativos al diseño y organización de los computadores. Estos conocimientos se imparten en asignaturas pertenecientes a la rama de Ingeniería de Computadores, tales como *Introducción a los Computadores* y *Estructura de Computadores*. Dichas asignaturas son consideradas básicas en la titulación, y todo estudiante debe cursarlas para poder graduarse. En estas asignaturas se explican aspectos tales como los diferentes niveles de los lenguajes de programación o los *sets* de instrucciones de una arquitectura. Estas asignaturas toman como referente la arquitectura ARM [29]. Esta arquitectura sigue la filosofía RISC (*Reduced Instruction Set Computing*), presentando un número reducido de formatos en las instrucciones y teniendo estas siempre un tamaño constante. Además, una de las motivaciones por las que se escogió esta arquitectura es su amplia presencia en el mercado [30][4].

Uno de los aspectos más destacados de las asignaturas ya mencionadas es su alto componente práctico [38] [37]. El laboratorio se organiza mediante puestos basados en Raspberry Pi, ya que estos dispositivos presentan un bajo coste y una amplia disponibilidad. Estas cualidades son necesarias para que el alumnado tenga la capacidad de trabajar de forma autónoma. Como parte de las prácticas se centran en el uso de dispositivos de entrada/salida, en trabajos previos [2] [48] se decidió utilizar el sistema operativo RISC OS [31]. Este sistema operativo fue desarrollado inicialmente para la arquitectura ARM por parte del equipo que diseñó la propia arquitectura. Este sistema operativo nace influenciado por el *ACORN MOS*, otro sistema operativo lanzado originalmente para la *BBC Micro*, una plataforma desarrollada por *Acorn Computers* como parte del proyecto *Computers Literacy* de la BBC, orientado al ámbito de la docencia en Reino Unido.

Se escogió este sistema operativo principalmente por dos motivos. Por una parte, RISC OS dispone de una interfaz gráfica que simplifica trabajar con dicho sistema. En segundo lugar, RISC OS permite manejar los dispositivos conectados a muy bajo nivel, pudiendo controlar la entrada o salida de cada uno de los pines de las Rasbberry Pi a los que se conecta el periférico sin apenas intervención por parte del sistema operativo.

Los alumnos tienen la posibilidad tanto de ejecutar directamente su código como de depurarlo haciendo uso del depurador UCDebug [16]. Este depurador, desarrollado íntegramente desde la Universidad de Cantabria, nace a causa de la inexistencia en RISC OS de una herramienta de depuración gratuita que disponga de interfaz gráfica y cuyo uso sea relativamente sencillo. UCDebug permite comprobar el funcionamiento de los programas desarrollados por los alumnos, mostrando los cambios sobre los elementos del computador (CPU, memoria) que genera durante la ejecución. El uso de un depurador nos ayuda a determinar la causa de posibles fallos de ejecución y nos permite observar si el código se ajusta a la funcionalidad pedida, como parte del proceso de desarrollo del código [36].

En este trabajo se pretende estudiar la viabilidad de una sustitución de las placas actuales basadas en la arquitectura ARM a un nuevo hardware basado en RISC-V [24] [1]. Esta migración viene motivada por los beneficios que trae consigo esta arquitectura, además del hecho de que esta sigue una filosofía que se adecua más con los intereses de la universidad así como su presencia en diversos proyectos europeos.

RISC-V es una arquitectura abierta desarrollada inicialmente por la Universidad de California. A diferencia de ARM, RISC-V permite a desarrolladores y empresas diseñar sus propios procesadores sin la

necesidad de tener que abonar licencias asociadas a la arquitectura. Esto ha hecho que en los últimos años RISC-V esté ganando popularidad, creando una comunidad abierta que fomenta un entorno colaborativo. En este trabajo se estudia el estado del arte de esta tecnología, valorando que posibles aproximaciones se pueden tomar de cara a la creación de un laboratorio de computación basado en RISC-V.

El nuevo entorno planteado deberá cumplir con los mismos requisitos que se establecieron para el laboratorio basado en ARM. En primer lugar, el hardware debe ser accesible para el alumno, algo que se consiguió mediante el empleo de RaspberryPi. En el momento de la redacción de esta memoria existe una disponibilidad limitada de placas basadas en RISC-V, lo que hace que el precio de éstas pueda no resultar accesible de cara a su compra por parte del alumnado, ya que en función del modelo puede ser necesario incluso solicitar la producción de los dispositivos bajo demanda. Sin embargo, esta renovación se plantea de cara al medio o largo plazo, y dado el atractivo comercial de esta arquitectura, creemos que esta continuará ganando presencia en el mercado, con lo que los costes de las placas se reducirán.

Un detalle adicional es que no existe en RISC-V el paradigma de Raspberry Pi, en el que el objetivo no es necesariamente producir placas de bajo coste con las mejores prestaciones, sino mantener la producción y soporte de placas ya antiguas, facilitando la formación de una comunidad de desarrollo y favoreciendo un enfoque académico y de iniciación, lo que redunda en una mayor documentación, una mayor facilidad para universidad y alumnos para adquirir las mismas placas, y un mayor atractivo para el alumno para emplear las placas más allá del uso previsto en las asignaturas. En un trabajo aún en curso pero iniciado de forma previa a este TFM, se observaron las limitaciones de desarrollar un entorno de laboratorio basado en placas RISC-V, y la falta de continuidad en la producción de las mismas placas, lo que dificulta en la actualidad la implementación de un laboratorio basado en RISC-V.

Debido al estado prematuro de esta tecnología y a la baja disponibilidad de placas que esto conlleva, durante el desarrollo de este proyecto se ha trabajado con el emulador QEMU [11]. Esta herramienta nos ha permitido realizar diferentes estudios sin depender de un modelo concreto de placa, aportando mayor flexibilidad de cara a la elección del hardware que se emplee en una futura implementación del entorno de laboratorio.

El segundo requisito del nuevo entorno se centra en las capacidades del sistema operativo que se utilice sobre el hardware RISC-V. Como ya se ha comentado, RISC OS presentaba en un mismo sistema una interfaz amigable y una baja intervención a la hora de acceder a los recursos hardware de la placa, como los pines de entrada y salida. RISC OS es un sistema operativo antiguo, nacido para la divulgación de las primeras iteraciones de la arquitectura ARM y prácticamente obsoleto para el uso cotidiano. Al tratarse de un SO mantenido por una comunidad de aficionados mediante donaciones, no dispone de versión para la arquitectura RISC-V ni medios para adoptarla en un futuro, lo que dificulta la migración a RISC-V del entorno de trabajo actual en el laboratorio. No se han encontrado sistemas similares disponibles para la arquitectura RISC-V, por lo que se han ideado dos posibles aproximaciones basadas en el uso de hipervisores que pueden ser tomadas en consideración. La idea básica de estas aproximaciones es la coexistencia en la placa de dos sistemas operativos, uno que pueda aportar la interfaz y las comodidades necesarias de cara al usuario, y otro que permita la ejecución de código con bajas restricciones.

Por este motivo, una de las tareas realizadas durante el transcurso de este proyecto ha sido el análisis de los diferentes sistemas operativos e hipervisores que se encuentran actualmente disponibles para la arquitectura RISC-V. Para cada uno de ellos se pretende analizar el estado en el que se encuentra y cómo se adapta a las necesidades del proyecto.

1.2. Objetivos

El objetivo principal de este proyecto reside en un estudio de viabilidad del concepto de entorno con dos sistemas operativos y un hipervisor, mediante el análisis del estado del arte de las herramientas y sistemas operativos desarrollados para la arquitectura RISC-V. Se pretende analizar las capacidades que presentan y de qué modo se pueden utilizar dichas capacidades para desarrollar un nuevo laboratorio basado en la arquitectura RISC-V. Se estudiará la viabilidad de un sistema en el que, por medio de un

hipervisor, coexistan dos sistemas operativos ejecutándose sobre una placa RISC-V, de modo que uno de ellos sirva como interfaz gráfica y entorno de desarrollo para el alumno, mientras que el otro presente bajas restricciones a la hora de interactuar con el hardware, permitiendo de este modo la ejecución del código desarrollado por el alumno.

Como parte fundamental de esta tarea, se pretenden idear diferentes estrategias de desarrollo basadas en el modo en el que interactúan ambos sistemas operativos entre sí. Por un lado se plantea la posibilidad de que un sistema operativo sirva únicamente de interfaz mientras que el otro sistema se encargue de contener al núcleo del depurador, controlando la ejecución del código del alumno. Alternativamente, se plantea que el sistema menos restrictivo simplemente sirva como entorno de ejecución para el código del alumno, mientras que el sistema que sirve de interfaz, a su vez, se encargue de depurar la ejecución del sistema operativo ligero y, a su vez, el código del alumno.

Gran parte de los hipervisores encontrados se ejecutan directamente sobre el hardware, sin necesidad de un sistema operativo por debajo. Por este motivo es necesario emplear un conjunto de herramientas denominado toolchain que permitan la compilación cruzada del sofware que debe correr directamente sobre el hardware RISC-V. Una toolchain es una colección de herramientas que permite desarrollar un conjunto de tareas, como puede ser compilar un fichero, depurar un ejecutable, modificar ficheros binarios, etc. Será necesario por lo tanto comprobar qué toolchains son necesarias para cada caso, puesto que en función del sistema operativo o hipervisor que se quiera compilar se encuentran unas limitaciones u otras.

Será necesario también comprender qué mecanismos se pueden utilizar a la hora de comunicar ambos sistemas operativos, ya sea a través del hipervisor que los virtualiza o a través de otros mecanismos, de modo que se pueda controlar el núcleo de depuración desde la interfaz gráfica y mostrar la información relativa al depurado en esta.

Por último, se tratará de desarrollar una prueba de concepto de alguna de las estrategias presentadas que ilustre la viabilidad de los objetivos planteados en este proyecto. Esta prueba consistirá en una demostración mínima de la depuración de código en RISC-V, evidenciando la viabilidad del proyecto. A lo largo de esta memoria se podrá comprobar cómo los diferentes objetivos han sido cumplidos de forma satisfactoria, demostrando la viabilidad del laboratorio planteado.

1.2.1. Desarrollo del trabajo

Dada la naturaleza del proyecto, las diferentes tareas que se han llevado a cabo se han ido superponiendo entre sí. Durante la prueba de un hipervisor, por ejemplo, ha sido necesario revisar las toolchains disponibles, realizar modificaciones en el entorno de desarrollo, profundizar en las capacidades y opciones de la arquitectura, etc.

Aunque la idea de realizar este proyecto se formalizó en julio de 2023, no fue hasta septiembre de ese año cuando se comenzó a estudiar y revisar la documentación y estado de la arquitectura RISC-V. Este proceso comprende en primer lugar el estudio de la arquitectura en sí, es decir, en qué estado está, qué extensiones define y cuáles son potencialmente necesarias para los propósitos planteados (como la extensión para hipervisores), así como tratar de idear las posibles estrategias que permitan lograr los objetivos planteados en este capítulo. Por otro lado, se ha hecho un estudio de las herramientas disponibles para esta plataforma que puedan ser utilizadas para los fines mencionados, es decir, recopilar el conjunto de sistemas operativos e hipervisores que puedan ser útiles para la realización de este proyecto.

Estas tareas se han realizado tanto en una etapa inicial del proyecto, sirviendo como punto de partida, como en la etapa final, en la que se escribe esta memoria, para proponer con precisión las tareas de trabajo futuro que se pueden realizar, al disponerse de una visión de conjunto. En total se estima haber dedicado aproximadamente 90h de trabajo en este bloque de tareas.

El siguiente objetivo dentro del proyecto ha consistido en el estudio del entorno de trabajo sobre el que

realizar las diferentes pruebas y simulaciones descritas en esta memoria. Durante este proceso se ha tratado de emular un sistema operativo que sirva como base para las futuras pruebas, idealmente con una interfaz gráfica; por limitaciones de los entornos empleados, no ha sido posible el uso de la interfaz gráfica para las pruebas de viabilidad, pero se han elegido solo herramientas que permitan disponer de dicho entorno. En total a esta tarea se le han destinado 50 horas de trabajo aproximadamente.

Otro bloque de tareas se ha basado en la prueba y análisis concreto de los diferentes sistemas operativos e hipervisores, probando a instalar y ejecutar cada uno de ellos, analizando qué capacidades tienen y valorando de qué modo estas se adaptan a los objetivos del proyecto, escogiendo por último los que han resultado más prometedores de cara a los intereses del proyecto. Se han invertido en torno a 90 horas de trabajo al desarrollo de estas tareas.

La instalación de los sistemas operativos e hipervisores ha requerido en varios casos de la instalación de diferentes toolchains. Este proceso no ha resultado trivial, y ha precisado en ocasiones de procesos de prueba y error para instalar según qué software. En total, a la instalación y prueba de toolchains, se le han dedicado 30 horas de trabajo aproximadamente.

Con los sistemas operativos e hipervisores más prometedores se han realizado pruebas específicas para analizar su viabilidad en los escenarios específicos del proyecto. Como se verá más adelante, por un lado se ha analizado la implementación de los mecanismos de memoria compartida que presenta el hipervisor BAO. Por otro lado se ha estudiado el uso de KVM para depurar un sistema operativo completo junto con su código en RISC-V, lo que ha obligado a recompilar múltiples veces (una vez se corrompió el sistema operativo emulado, otra se rompió el disco duro del equipo) el kernel del sistema operativo principal dentro de una máquina emulada, lo cual resulta en un proceso lento que requiere de atención ya que durante el proceso se debían instalar algunos paquetes en el sistema. A este proceso de pruebas específicas se le han dedicado 120 horas de trabajo aproximadamente.

Por último ha sido necesario redactar la memoria en la que se agrupan y exponen todas las tareas que se acaban de mencionar. Durante este proceso ha sido necesario repetir o realizar algunas pruebas para disponer de una documentación exhaustiva y con criterios unificados. Se estima que esta redacción ha ocupado aproximadamente 100h de trabajo.

En la *Figura 1* se puede ver el diagrama de Gantt con los grupos de tareas en los que se ha dividido el proyecto.

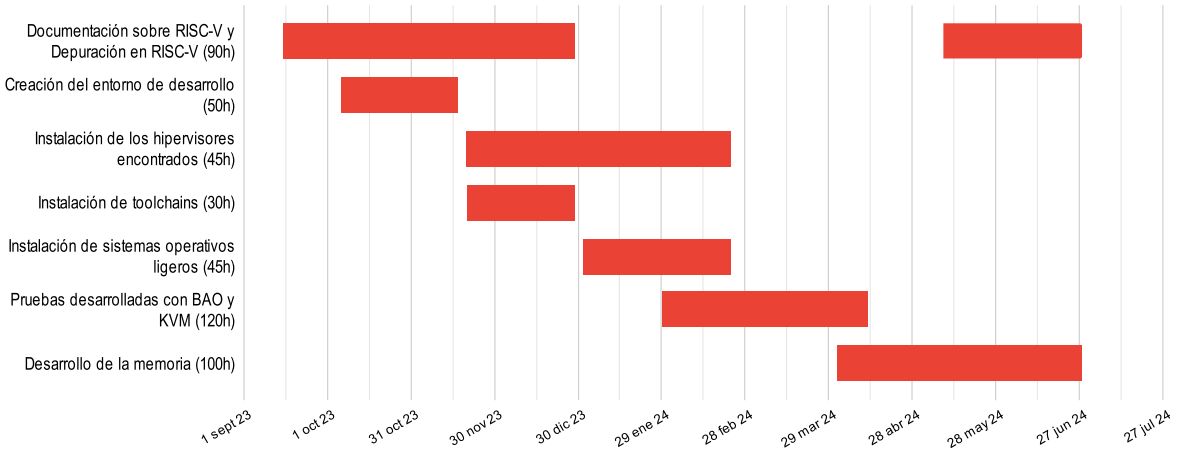

Figura 1: Diagrama de Gantt del proyecto.

2. Planteamiento

En este capítulo se va a describir cuál ha sido el entorno de trabajo utilizado para el desarrollo del proyecto, así como los principales enfoques o estrategias que se plantean para conseguir los objetivos propuestos.

Actualmente existen diferentes limitaciones a la hora de plantear el hecho de utilizar hardware real para las pruebas que se pretenden realizar en este proyecto. En primer lugar, la baja accesibilidad de las placas de RISC-V. Actualmente RISC-V es una arquitectura que no está implantada por muchos fabricantes, por lo que existe una oferta limitada de placas. Además, para varios modelos, es necesario solicitar la producción de las unidades que se quieren adquirir. Estos factores provocan que el precio de las mismas suponga una barrera a la hora de adquirir estos productos. En segundo lugar, muy pocas placas de las que actualmente se ofertan tienen el soporte hardware necesario para el desarrollo de las actividades que se plantean en este trabajo. Concretamente algunas de las estrategias planteadas en este trabajo requieren que estas placas tengan la extensión para hipervisores que se recoge dentro de la especificación de la arquitectura RISC-V.

2.1. Entorno de desarrollo

Debido a estas limitaciones, se ha decidido el uso de QEMU para el desarrollo de las pruebas de concepto y viabilidad de la propuesta, que se describen posteriormente en el *Capítulo 5*. QEMU es un software de código abierto que permite la emulación de hardware y la virtualización de máquinas, y está disponible para los sistemas operativos Linux OS, macOS, FreeBSD, NetBSD, OpenBSD y Windows en las arquitecturas Arm, MIPS, PPC, RISC-V, s390x, SPARC y x86. Este emulador nos permite ejecutar sistemas operativos destinados para una arquitectura de procesador diferente a la arquitectura sobre la que se ejecuta QEMU. Cabe destacar que QEMU es una herramienta madura, que permite actualmente emular x86, ARM, y arquitecturas de menor popularidad actual como SPARC, Alpha o PowerPC [12]. QEMU también nos permite emular la arquitectura RISC-V, dándonos la opción de escoger una placa existente concreta la cual emular, o permitiéndonos utilizar una máquina virtual genérica que no esté basada en ningún hardware real, pero incluyendo características que no tienen por qué estar aún implementadas en placas concretas del mercado, como el soporte para hipervisores. Este último aspecto es de particular relevancia para el objetivo del trabajo, ya que se pretende analizar un entorno de trabajo basado en el uso de hipervisores.

En este trabajo se pretende realizar un análisis de viabilidad de un puesto de laboratorio basado en la arquitectura RISC-V. Esto implica que no se parte de un hardware concreto que se tenga que utilizar, ni tan siquiera que las placas que posiblemente se acaben utilizando, si este proyecto se llega a implantar en la actividad docente, existan actualmente en el mercado. En este escenario, las posibilidades que proporciona QEMU tienen diversas ventajas.

Por un lado nos permite desarrollar sin tener en mente una placa concreta, gracias a la máquina genérica que se puede emular, de modo que se pueden valorar los diferentes planteamientos sin la limitación de que el hardware subyacente los pueda soportar. Esto, además, permite valorar qué características debe tener la placa que se quiera utilizar en la aplicación práctica de este proyecto. Por otra parte, si surgiese alguna placa que se pudiese considerar como candidata, QEMU nos permite emular el hardware concreto de la misma, de modo que se puedan hacer pruebas preliminares sobre hardware específico sin la necesidad de adquirir dicho hardware.

El sistema operativo que se ha utilizado como base para las diferentes pruebas es Ubuntu 22.04, de modo que este será el sistema operativo "principal"que se emule y que sirva de interfaz para el usuario. Ubuntu proporciona soporte para la arquitectura RISC-V desde la versión 20.04, recomendando a partir de la versión 22.04, el uso de la imagen precompilada de Ubuntu-server para la placa *SiFive HiFive Unmatched* [49] para su emulación con QEMU.

Otro factor que se tiene que tener en cuenta a la hora de realizar las diferentes pruebas es la versión de

QEMU que se está utilizando. A medida que se va actualizando la plataforma, se va añadiendo nuevo soporte a diferentes elementos emulables con QEMU. Esta información se puede obtener fácilmente consultando el *ChangeLog* [3] disponible en la página web de la herramienta. Como referencia se han tratado de utilizar versiones posteriores a la versión 7.0 de QEMU, ya que en esta versión se añaden características que resultan convenientes para las tareas que se quieren realizar. Por un lado, habilitan por defecto la extensión para hipervisores definida por la ISA de RISC-V, que estaba disponible en versiones anteriores de modo experimental. Por otro lado, añaden el soporte para el hipervisor KVM.

Pese a los beneficios que trae consigo el uso de versiones modernas de QEMU, no siempre ha sido posible utilizar las últimas versiones de la herramienta. En este trabajo se ha explorado el uso de diferentes hipervisores, algunos de los cuales no han recibido soporte en los últimos años y que no es posible ejecutarlos con las versiones más modernas del emulador. Por este motivo ha sido necesario disponer de diferentes instalaciones de QEMU conviviendo dentro del entorno de trabajo (principalmente las versiones 4.2, 5.0, 6.2, 7,2, 8.2), teniendo que hacer uso de diferentes ejecutables en función de las necesidades y requerimientos de la prueba concreta que se haya querido desarrollar.

Para configurar el emulador es necesario especificar diferentes parámetros que indiquen las características del entorno que se quiere emular. Los argumentos más comunes que se han utilizado en las diferentes ejecuciones de QEMU son los siguientes:

- *-machine*: Entorno físico que se quiere emular. Se pueden especificar tanto entornos hardware reales, como entornos genéricos virtuales. Por ejemplo, el argumento *-machine virt* sirve para especificar que se quiere emular un entorno genérico que no corresponde con ningún hardware específico. En todas las ejecuciones de QEMU detalladas en este trabajo se hace uso de esta opción, ya que, al no conocer el harware concreto en el que se ejecutarán las prácticas del laboratorio de computación, nos permite realizar las pruebas pertinentes sin afrontar problemas específicos de un dispositivo concreto.
- *-m*: Cantidad de memoria RAM para el entorno emulado. Con la opción *-m 4096* se estaría especificando la emulación de un sistema con 4GB de memoria principal. Por defecto se usará esta cantidad de memoria, salvo en casos concretos en los que las necesidades del sistema requieran un tamaño superior.
- *-smp*: Número de procesadores virtuales que se quieren emular. Por defecto se usará la opción *-smp 4* en las sucesivas invocaciones a QEMU, indicando que se requiere de 4 cores en la máquina emulada.
- *-nographic*: Parámetro que especifica que no se debe abrir una ventana gráfica para la salida de la máquina virtual. Esto permite continuar con la ejecución de la máquina desde la misma terminal desde la que se ejecuta el comando de QEMU.
- *-bios*: Ruta al firmware de inicio que se utilizará en la máquina virtual. En RISC-V, si se omite este atributo o se pone la opción *default*, se cargará el firmware de OpenSBI (una implementación open-source de la *Supervisor Binary Interface* de RISC-V) automáticamente. Este firmware se incluye con las instalaciones de QEMU. Otras opciones posibles son *none*, que indica que no se especifica ningun firmware para la emulación, o la ruta al fichero específico con el firmware.
- *-kernel*: Ruta al kernel que se debe cargar en la máquina virtual. En las ejecuciones típicas realizadas, el kernel que se quiere utilizar se encuentra en la imagen que se monta con el sistema operativo. A este parámetro por lo tanto se le pasa la ruta a un bootloader (ej: uboot) desde el que se arranca el sistema operativo con alguno de los kernels instalados.
- *-drive*: Disco virtual que se debe montar en la máquina virtual. En el caso más común corresponde con la ISO de Ubuntu que proporciona la web oficial. Acepta diferentes parámetros adicionales, como el archivo con la imagen de disco o la interfaz con la que montarlo.
- *-device*: Permite añadir el controlador de un dispositivo como, por ejemplo, el driver de una tarjeta de red.

Se ha utilizado el sistema operativo Ubuntu server 22.04 dado que este sistema operativo cuenta con una gran comunidad detrás y tiene soporte para la arquitectura RISC-V. Se utiliza la versión para servidores dado que esta es la única que ubuntu proporciona para la arquitectura RISC-V. Sobre esta imagen se han tratado de instalar los paquetes que posibilitan el uso de interfaz gráfica, pero se ha encontrado que estos no están disponibles en el respositorio oficial de ubuntu para la arquitectura RISC-V, por lo que se plantea esta tarea como parte del trabajo futuro del proyecto, a la espera de futuras actualizaciones del sistema operativo que habiliten esta posibilidad.

En las recomendaciones del desarrollador se indica el uso de un entorno virtual genérico *virt* con 4GB de memoria; 4 cores; el disco con la imagen de Ubuntu server precompilada para la placa *SiFive HiFive Unmatched,* que se carga desde el bootloader U-Boot [51], y un dispositivo de red (útil para instalar o actualizar software en el sistema operativo). Dicha configuración de entorno emulado se ejecuta mediante el comando del *Listado 1.*

```
./qemu-8.2.2/build/qemu-system-riscv64 -machine virt -m 4096 -smp 4 -nographic
-bios /usr/lib/riscv64-linux-gnu/opensbi/generic/fw_jump.bin
-kernel /usr/lib/u-boot/qemu-riscv64_smode/uboot.elf -drive file=ubuntu.img,
format=raw,if=virtio -netdev user,id=mynet0 -device virtio-net-pci,netdev=mynet0
```

Listado 1: Emulación de Ubuntu RISC-V con QEMU. La opción *-netdev* crea una red virtual entre host y guest, proporcionando conectividad. La opción *-device* permite añadir dispositivos a la máquina emulada, en este caso una interfaz de red.

2.1.1. Toolchains

Durante el desarrollo de este trabajo se ha necesitado compilar diferentes recursos, como hipervisores o sistemas operativos, para lo que ha sido necesario hacer uso de diferentes *toolchains*. Concretamente se han empleado variadas toolchains que permitan la compilación de software de forma cruzada. Es decir, se quiere compilar el software para que se ejecute en una arquitectura diferente a aquella en la que está ocurriendo el proceso de compilación.

Esto ha sido necesario ya que, por ejemplo, parte de los hipervisores que se han estudiado se ejecutan *bare metal*, es decir, directamente sobre el hardware sin un sistema operativo por debajo. Como ya se ha explicado, en ausencia de hardware real estos hipervisores serán ejecutados a través del emulador QEMU, pero requieren de un proceso previo de compilación. Al no haber tenido acceso a placas RISC-V durante el desarrollo de este proyecto ha sido imprescindible el uso de estas herramientas para realizar la compilación cruzada del código.

Principalmente se han utilizado dos toolchains diferentes durante la realización de este proyecto. Por un lado se ha hecho uso de la toolchain proporcionada por GNU [6]. Uno de los aspectos a los que prestar atención con esta herramienta es si la versión que se descarga incluye o no las librerías de C (*glibc*) en el caso de descargar los binarios precompilados, o si se configura correctamente para incluir dichas librerías en el caso de compilarla desde su código fuente.

La otra toolchain utilizada es la proporcionada por SiFive [47], desarrolladores de hardware RISC-V. Esta toolchain es la recomendada por algunos de los hipervisores que se han analizado, aunque no funciona en todos ellos, por lo que no se puede hacer un uso general de ella. La última release del software es de 2020, pero en el nombre de esta se hace referencia al sistema operativo Ubuntu 14.

Se ha tratado de instalar dicha toolchain a través de la compilación del código fuente proporcionado en su repositorio oficial, ya que de este modo se está utilizando el código más reciente de la misma. Además, usar una compilación propia en lugar de los binarios precompilados puede reducir posibles problemas de compatibilidad futuros, además de permitir la modificación de la toolchain en caso de ser necesario.

Sin embargo, se ha encontrado que este proceso no es trivial. Durante su instalación es posible escoger diferentes opciones a instalar en función del sufijo que se utilice a la hora de nombrar el paquete. Las opciones que se nombran son **-package*, **-native-package* y **-cross-package*. No se ha conseguido encontrar detallado el significado de estos sufijos, ni el uso de cada uno de los posibles paquetes, y la documentación de la toolchain [47] es insuficiente. Para conocer esta información ha sido necesario analizar parte de los scripts utilizados para construir la herramienta, de los que se ha concluido que tanto **-package* como **-native-package* hacen referencia a paquetes para Ubuntu de 64bits, mientras que **-cross-package* hace referencia a un paquete para Windows de 64 bits.

De este modo se ha visto cómo las opciones que deben ser utilizadas son o bien **-package* o bien **-native-package*. Sin embargo, instalando esta toolchain en Ubuntu 24.04, aparecen múltiples errores durante este proceso relacionados con algunas librerías, como se indica en el *Listado 22* del *Anexo A*. Solucionar estos problemas requeriría de un análisis en profundidad del código de la herramienta, lo que se sale del alcance planteado en este proyecto. Por este motivo se ha desistido de tratar de instalar la herramienta compilando el código fuente proporcionado, y se ha optado por utilizar los binarios precompilados proporcionados en el mismo repositorio.

2.2. Estrategias de trabajo

Como ya se ha explicado en la *Sección 1.1*, actualmente el alumnado desarrolla las prácticas en RISC OS, un sistema operativo que no restringe a los alumnos acciones como programar interrupciones o acceder a regiones de memoria privilegiadas. En ese capítulo también se mostró cómo *RISC OS* desde su comienzo es un sistema operativo pensado para fines académicos sobre la arquitectura ARM. Estas características han hecho que RISC OS haya sido una opción atractiva para el desarrollo del laboratorio de computación y que finalmente haya sido la elegida para el desarrollo de las prácticas sobre las RaspberryPi.

Sin embargo, este sistema operativo no está disponible para otras arquitecturas, y aunque RISC OS tenga una comunidad, que en el pasado ya ha realizado las adaptaciones necesarias para su ejecución sobre la RaspberryPi, las barreras sobre RISC-V ya mencionadas, sumado al hecho de que el público objetivo de cualquier adaptación de RISC OS a RISC-V sea muy específico, hace que una implementación de RISC OS para RISC-V sea particularmente improbable en el medio plazo.

Aunque esta implementación de RISC OS para la arquitectura RISC-V es posible, el esfuerzo necesario para realizar dicha tarea, así como la posterior adaptación del sistema operativo para la placa en concreto que finalmente se vaya a utilizar, excede de la labor que se puede desarrollar desde la Universidad de Cantabria. Además, este posible proyecto tendría poca usabilidad o atractivo, tanto para la comunidad de RISC OS como para la de RISC-V, debido al reducido número de personas que se podrían beneficiar de dicha aportación.

RISC OS proporciona, en un único sistema operativo, la libertad de modificar y utilizar elementos hardware de la placa sobre la que se ejecuta, así como una interfaz gráfica amigable que permite a los alumnos trabajar de forma cómoda. Aunque actualmente no exista un análogo de este sistema para la arquitectura RISC-V, este trabajo busca analizar la viabilidad de conseguir un entorno similar al mencionado, haciendo uso de múltiples sistemas operativos, manejados a través de hipervisores, de modo que cada uno de ellos aporte al conjunto las características deseadas.

De este modo, se propone el uso de un sistema operativo principal que presente una interfaz de usuario desarrollada, como Linux, que permita una interacción cómoda y fluida con el usuario, y otro sistema operativo que, pese a no tener todas las características principales de usabilidad presentes en los sistemas modernos, no limite las acciones a bajo nivel que los alumnos puedan realizar durante sus prácticas. De este modo obtendríamos un ecosistema en el que, aun con la funcionalidad de sistemas operativos complejos como Linux, no limite la ejecución de código ensamblador por parte del alumno.

Partiendo de esta base, existen existen dos paradigmas diferentes sobre los que se puede realizar la implementación. El primero de ellos plantea el uso de uno de los sistemas operativos como interfaz y otro como núcleo de depuración. El segundo plantea que, tanto la interfaz como el depurador, se sitúen en el sistema operativo principal, delegando únicamente la ejecución del código ensamblador al sistema operativo menos restrictivo.

2.2.1. Interfaz y núcleo

Una primera implementación plantea la existencia de un sistema operativo que sirva de interfaz, comunicado con otro sistema operativo que ejecute y depure el código del usuario. Esta aproximación, aún con diferencias, es similar al funcionamiento de UCDebug, el depurador que actualmente se usa en el laboratorio de computación. UCDebug está formado por una interfaz desarrollada en C, cuya función es ser el punto de interacción con el usuario. Esta interfaz se comunica con el núcleo del depurador, que es el encargado del control de la ejecución y manejo de los correspondientes errores del código del alumno, y que está desarrollado en lenguaje ensamblador ARM para facilitar la gestión de la ejecución y el manejo del hardware.

De esta aproximación surgen dos problemas que necesitan ser resueltos. El primero de ellos consiste en determinar qué sistema operativo se debe utilizar para alojar el núcleo del depurador, ya que este debe ser lo suficientemente complejo como para permitir desarrollar y ejecutar dicho núcleo, y que a su vez permita la ejecución de código de bajo nivel sin limitaciones. El segundo problema consiste en encontrar la forma de comunicar ambos sistemas operativos, de modo que el núcleo del depurador pueda ser controlado por la interfaz del usuario, y permitiendo a su vez que esta muestre información relativa a la ejecución del código.

La comunicación entre sistemas operativos va a depender de las capacidades y opciones que nos dé el hipervisor con el que se realice el despliegue. Algunas posibles vías pueden ser mecanismos de memoria compartida entre sistemas operativos, comunicación a través de dispositivos como la UART [22], o similares. En el caso de que los hipervisores disponibles no tengan estas capacidades mencionadas, se pueden explorar opciones basadas en la comunicación a través de la red, posiblemente añadiendo complejidad en el sistema operativo que constituye el núcleo del depurador, por lo que es una opción que debe ser evitada si es posible.

También es dependiente del hipervisor la disposición de los sistemas operativos. Los hipervisores se agrupan principalmente en dos categorías. Los hipervisores bare metal o de tipo 1 se caracterizan por ejecutarse directamente sobre el hardware, sin necesidad de un sistema operativo sobre el que correr. En este supuesto, los dos sistemas operativos mencionados se ejecutarían al mismo nivel, siendo ambos virtualizados por el hipervisor, como se muestra en la *Figura 2*.

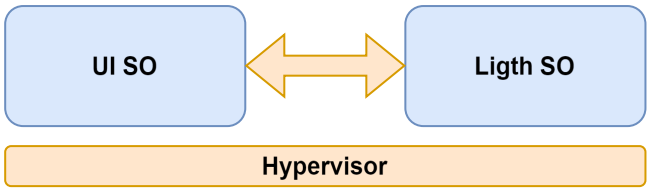

Figura 2: Desarrollo con hipervisores tipo 1.

Por otro lado, los hipervisores anidados o de tipo 2 necesitan un sistema operativo subyacente sobre el que ejecutarse. En este supuesto, sobre la placa se ejecutaría directamente el sistema operativo con la

interfaz gráfica, mientras que el sistema operativo con el depurador se ejecutaría sobre este, virtualizado por medio del hipervisor, como se muestra en la *Figura 3*.

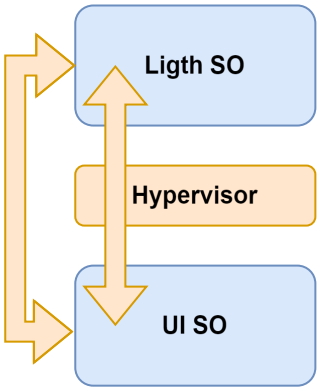

Figura 3: Desarrollo con hipervisores tipo 2.

Esta segunda aproximación se caracteriza por ser la más flexible de las dos planteadas. De cara a futuro, no hay en principio limitaciones que este planteamiento imponga sobre qué se puede hacer, por lo que posibles mejoras o añadidos sobre la funcionalidad original debieran ser realizables. Sin embargo, esta alternativa supone mucho trabajo de desarrollo para poder ser utilizada en el laboratorio con fines docentes.

En primer lugar, es necesario desarrollar el núcleo del depurador, en segundo lugar es necesario diseñar la interfaz gráfica que se mostrará en el sistema operativo complejo, y en tercer y último hay que diseñar un protocolo de comunicación que permita el paso de información entre los dos elementos del sistema.

2.2.2. Depuración completa del SO

En una implementación diferente a la anterior, tanto el núcleo de depuración como la interfaz se encuentran en el mismo sistema operativo. La idea en este caso es que el sistema operativo más liviano y menos restrictivo sirva únicamente para alojar el código del alumno.

Lo que se busca sería nuevamente contar con un sistema operativo completo que presente interfaz y sea amigable para el usuario, y un segundo sistema operativo muy ligero que únicamente esté capacitado para permitir la ejecución del código del alumno. Con esto, el flujo de trabajo se basaría en depurar completamente el sistema operativo ligero, y con él el código del alumno, desde el sistema operativo principal a través de herramientas como GDB [9].

Es importante que el sistema operativo tenga el menor número de funcionalidades posible, evitando el ruido que la ejecución de este pueda introducir en el desarrollo y depuración de las prácticas del alumnado, pero que mantenga los mecanismos mínimos necesarios para operar, como la implementación de las llamadas al sistema que, por ejemplo, habiliten el uso de diferentes niveles de privilegios.

Este enfoque, además, permitiría que los alumnos puedan avanzar las prácticas de forma autónoma sin disponer de la placa física a través de la emulación con QEMU, de forma similar a como se explica en el *Apartado 2.2*, de modo que puedan revisar o avanzar en los contenidos planteados durante las horas de laboratorio fuera del horario lectivo, sin la necesidad de adquirir el hardware necesario.

Ambos planteamientos tienen ventajas y desventajas. Por un lado, el uso de hipervisores requiere de mucho más trabajo para ser implementado, ya que es necesario encontrar la forma de comunicar ambos sistemas operativos para controlar el depurador desde la interfaz gráfica, así como desarrollar el núcleo

de depuración y la interfaz que se ejecuten sobre RISC-V. Sin embargo, este planteamiento es el que resultaría más agradable al usuario, además de permitir una mayor cantidad de mejoras o añadidos a futuro.

Por otro lado, la opción de depurar completamente el sistema operativo con el código del alumno inicialmente puede resultar más sencilla de implementar inicialmente, pero de cara al futuro es posible que sea más limitante a la hora de desarrollar nuevas funcionalidades o mejoras. Por este motivo no me he decantado por ninguna de las alternativas descritas, y se han planteado y estudiado ambos escenarios de cara a la realización del análisis presentado en este trabajo.

Cabe destacar que, en los enfoques planteados, al igual que ocurre actualmente en el laboratorio basado en RaspberryPi, se permite la ejecución del código de los alumnos de forma nativa sin tener que pasar por el depurador, obteniendo de este modo una ejecución más directa y sencilla del código.

Existen otras alternativas para depurar código sobre RISC-V, como puede ser el uso de conectores JTAG [7] que permiten el depurado de código desde equipos externos, o la adaptación de sistemas operativos modernos para habilitar la ejecución de código que realice accesos a memoria privilegiada sin restricciones, lo cual se está explorando actualmente en otros proyectos académicos de esta universidad. Aunque otros posibles enfoques puedan resultar prácticos, estos se alejan de la visión y objetivos que se tienen en el laboratorio, que buscan la interacción del alumnado directamente con el hardware que se estudia.

3. Hipervisores

El hipervisor juega un papel fundamental en el planteamiento de este proyecto, ya que es la pieza que permite la ejecución de múltiples sistemas operativos dentro de la misma placa RISC-V. Por ese motivo se ha realizado un estudio de las diferentes alternativas disponibles para su uso.

En este capítulo se describen los principales hipervisores que se han estudiado durante el desarrollo de este proyecto. Se analizará el estado actual de los hipervisores, la proyección de estos, las capacidades que ofrecen y la adecuación de cada uno de ellos en el contexto del trabajo planteado.

El estado actual de la arquitectura RISC-V es un factor clave a la hora de entender la situación actual del mercado en lo referente al software disponible para esta plataforma. Al tratarse de una tecnología relativamente reciente y que aún no tiene una presencia en el mercado significativa, no es de extrañar que la cantidad de software desarrollado para ejecutarse sobre RISC-V sea limitada.

Si bien es cierto que, en lo referente a hipervisores, existen diferentes opciones que se pueden estudiar, no todas ellas resultan atractivas a los intereses de este trabajo. Por un lado, gran parte están aún en el proceso de desarrollo y por lo tanto inmaduras. Por otro, otras han dejado de recibir soporte y actualmente están obsoletas o tienen requisitos que obligan al uso de versiones específicas de software, con la consiguiente limitación de desarrollo y capacidad de uso en el futuro.

Estos factores mencionados motivan el desarrollo de un estudio de mercado de las diferentes alternativas, tanto anunciadas por los medios oficiales del proyecto RISC-V, como encontradas por otros canales como parte de este estudio.

3.1. XtratuM

Uno de los hipervisores disponibles para la arquitectura RISC-V es XtratuM [13] [28]. Xtratum es un hipervisor de tipo 1 orientado a la ejecución de sistemas embebidos de tiempo real, desarrollado inicialmente por la Universidad Politécnica de Valencia, cuya finalidad era la de la virtualización de software en la industria espacial, con la idea de extender su uso a la industria aeronáutica y automovilística.

Este hipervisor ofrece mecanismos de comunicación entre particiones basados en el estándar *ARINC-653*, además de otro conjunto de funcionalidades relacionadas con los sistemas críticos de tiempo real, como la detección de las situaciones de error, módulos encargados de la gestión de errores o sistemas de recolección de trazas referentes a la evolución del sistema.

En 2010 se fundó "FentISS", una empresa que surge como spin-off del proyecto de XtratuM y que comercializa su uso desde entonces [14]. A raíz del proyecto De-RISC, la compañía realizó una adaptación del hipervisor para el procesador NOEL-V, un modelo VHDL de un procesador basado en la arquitectura RISC-V diseñado para su uso en aplicaciones espaciales.

A causa de esto, y aunque el código original de XtratuM siga siendo público, ha quedado sin soporte, ya que su desarrollo ha pasado a comercializarse por la empresa FentIIS. Por este motivo, aunque pueda encontrarse que Xtratum es un hipervisor de código abierto, y que este además tiene soporte para la arquitectura RISC-V, ambas premisas no se dan de forma simultánea.

Como parte de las premisas planteadas en la *Sección 1.1*, y en el marco de la motivación de emplear herramientas y arquitecturas de tipo libre, uno de los requisitos es el uso de herramientas de código abierto para la implementación del nuevo laboratorio, por lo que, al no existir una alternativa gratuita de XtratuM que tenga soporte para la arquitectura RISC-V, esta opción queda descartada de cara a análisis más profundos de la herramienta.

3.2. Xen

Xen [20] es un hipervisor de tipo 1 desarrollado por la universidad de Cambridge para la arquitectura x86, aunque a día de hoy anuncia tener soporte oficial para la arquitectura ARM. Actualmente es una de las opciones más utilizadas en el mercado, y entre los miembros que forman parte de su proyecto de desarrollo se pueden encontrar a AMD, ARM y AWS (Amazon Web Services).

El hipervisor se describe por primera vez en 2003 como un proyecto open source [40]. En este artículo se describe cómo Xen se desarrolla buscando la máxima eficiencia de cara a la virtualización de los sistemas operativos, tratando de conseguir un impacto mínimo en el rendimiento de estos. Su meta era conseguir ejecutar hasta 100 instancias de máquinas virtuales de forma simultánea en un procesador moderno de su época.

En 2013 el proyecto se une a la *Linux Fundation* y pasa a recogerse bajo la marca comercial *Xen Project* buscando diferenciar el proyecto open source de las alternativas que se habían intentado comercializar bajo el nombre de Xen.

Xen por su parte proporciona una serie de funcionalidades que podrían ser explotadas de cara al trabajo propuesto, especialmente en el apartado de la comunicación entre máquinas virtuales, necesario para interconectar interfaz y núcleo de depuración.

En primer lugar, Xen permite definir regiones de memoria compartida entre diferentes guests [21], mecanismo que podría ser utilizado para, por medio de ring buffers, transmitir ficheros entre sistemas operativos, como podría ser el código del alumno escrito en el sistema operativo principal que necesita ser transferido al núcleo de depuración. Otro mecanismo interesante que proporciona Xen es la posibilidad de utilizar Canales de Eventos [17]. Los canales de eventos son una de las primitivas proporcionadas por el hipervisor para la notificación de eventos, y son el equivalente a interrupciones hardware. Haciendo uso de estos eventos sería posible, por ejemplo, notificar nuevas escrituras en memoria, transformando una posible comunicación síncrona basada en polling a una comunicación asíncrona basada en notificaciones. Esto permite agilizar el funcionamiento y reducir el impacto en tiempo de ejecución del polling. Otro posible uso de estos canales de eventos se basaría en definir diferentes tipos de canales para realizar diferentes acciones en el núcleo de depuración. Por ejemplo, a través de los manejadores definidos en el sistema operativo que contiene el núcleo de depuración, se podrían realizar acciones como iniciar, detener o continuar la ejecución de código mandando estos eventos desde el sistema operativo principal, el que proporciona la interfaz con el usuario.

Nuevamente, debido a que RISC-V es una arquitectura relativamente nueva y que aún no tiene una gran presencia en el mercado de procesadores, Xen no dispone todavía de una versión estable de su hipervisor para este tipo de hardware. Sin embargo, ya ha empezado a destinar recursos para que su plataforma pueda ejecutarse en esta arquitectura, tal y como detallan en las diferentes entradas de sus nuevos anuncios referentes al proyecto [19].

Xen es una alternativa prometedora que junta un proyecto open source muy apoyado por la comunidad con unas características que se adecuan al contexto de este trabajo. Pese a no estar aún disponible para la arquitectura RISC-V, han comenzado su desarrollo para estas plataformas, por lo que es una opción que debe ser valorada de cara a la implantación del futuro laboratorio.

3.3. Jailhouse

Similar al caso anterior, Jailhouse [46] es un hipervisor open source bare metal desarrollado por Siemens y basado en Linux, para las arquitecturas x86 y ARM, que permite ejecutar aplicaciones bare metal o sistemas operativos adaptados para su ejecución en esta plataforma. Su filosofía está orientada a proporcionar un hipervisor lo más sencillo posible a costa de disminuir sus funcionalidades. Esto implica que no soporta asignar los mismos recursos físicos como la CPU o RAM a múltiples dispositivos y solo virtualiza

aquellos elementos que no se puedan particionar desde el hardware.

A diferencia de otros hipervisores bare metal o de tipo 1, necesita ser cargado y configurado desde un entorno Linux subyacente. El proceso de arranque sería entonces cargar Linux en la máquina, configurar y activar Jailhouse, y solo entonces este último tomaría control total sobre los recursos del sistema.

Sin embargo y pese a su filosofía, existen una serie de funcionalidades referentes al uso de memoria compartida que están en desarrollo, pero que podrían ser aprovechadas por los intereses de este trabajo [45]. Jailhouse permite definir secciones de memoria compartida con diferentes propósitos, como por ejemplo regiones que guarden el estado de cada guest, regiones que solo puedan ser escritas por ciertos guest y leídas desde el resto o regiones con lectura y escritura habilitadas para todos los guests.

Además, estas nuevas funcionalidades incluyen la posibilidad de enviar interrupciones a través de señales de un sistema operativo a otro. Con estas funcionalidades sería posible establecer la comunicación entre interfaz y núcleo de depuración de forma similar a como se ha planteado en el caso de Xen. Sin embargo, y como ya se ha mencionado, estos sistemas aún no están implementados en Jailhouse, pero son características atractivas de cara a los intereses del trabajo realizado.

Igual que en el caso anterior, aunque Jailhouse aún no está disponible para la arquitectura RISC-V, los desarrolladores han comenzado a desarrollar la adaptación de su software para esta arquitectura. No obstante, el repositorio de código que aloja esta implementación hace 4 años que recibió la última contribución.

El hecho de que ya no se estén destinando recursos a esta rama de trabajo no quiere decir que ésta haya sido abandonada o que la parada del desarrollo sea permanente, pero sí que es algo que hace dudar de la implantación del hipervisor en un futuro previsible. Por este motivo, y aunque esta opción se adecúe a los requerimientos del trabajo, a día de hoy no se toma como una opción preferente de cara al desarrollo del futuro laboratorio.

3.4. Diosix

A diferencia de los casos anteriores, Diosix [50] es un hipervisor bare metal desarrollado exclusivamente para la arquitectura RISC-V de 64 bits que no requiere de soporte hardware (extensión para hipervisores) para ser ejecutado. Diosix nace de un proyecto de código abierto desarrollado en exclusiva por Chris Williams, editor jefe de la revista *The Register*.

Aun siendo un proyecto que no ha terminado su desarrollo, es posible ejecutar el hipervisor tanto sobre hardware real como a través de emuladores como QEMU. Sin embargo, en el estado actual de su desarrollo, no especifica disponer de funcionalidades relativas a la intercomunicación de los sistemas operativos, como mecanismos de memoria compartida o de envío de señales, a diferencia de las opciones anteriores.

Esto significa que, a priori, no se puede contar con comunicación entre sistemas operativos a través de mecanismos ofrecidos por el hipervisor. No obstante, como ya se ha mencionado en el *Capítulo 2.2*, sigue siendo posible establecer otros mecanismos, como, por ejemplo, mensajes a través de la red, que permitan la comunicación entre la interfaz gráfica y el núcleo de depuración, aunque esto signifique tener que disponer (o implementar) de estos mecanismos en el sistema operativo más liviano.

Diosix pone a nuestra disposición una pequeña demo que consta del hipervisor precompilado con una imagen de Linux que puede ser ejecutada desde QEMU, de modo que se puedan probar de forma preliminar sus capacidades de un modo sencillo. Para conseguirlo, únicamente es necesario descargarse los binarios y ejecutarlos a través de QEMU.

Sin embargo, durante las pruebas realizadas, se ha visto que Diosix sobre QEMU deja de funcionar si se utilizan versiones del emulador superiores a la versión 5.2, que es la última versión disponible en el

repositorio de Ubuntu en el momento de la realización de este proyecto. Concretamente se han realizado pruebas con las versiones de QEMU 6.2, 7.2 y 8.2, con las que la ejecución de Diosix quedaba congelada, sin ningún tipo de salida por la terminal.

Por otro lado, el archivo precompilado de Diosix trae una prueba en la que se ejecuta un único sistema operativo. Otra de las pruebas realizadas ha consistido entonces en tratar de configurar Diosix para poder ejecutar varios sistemas de forma paralela.

Diosix se puede configurar a través del fichero *manifest.toml* encontrado en la raíz del directorio que contiene el código fuente. Los aspectos más relevantes de cara a la prueba que se quiere desarrollar son los siguientes.

En primer lugar, los sistemas operativos que se quieren virtualizar se definen como se muestra en el *Listado 2*, donde se especifican los recursos que se quieren destinar para la virtualización del sistema operativo y la ubicación del mismo.

```
# a mildly useful Linux with busybox, micropython, zsh, and less
[guest.riscv64-linux-busybox-micropython]
path = "boot/guests"
url = "https://github.com/diodesign/diosix/raw/binaries/buildroot-guests/
riscv64-linux-busybox-micropython"
description = "64-bit RISC-V Linux with Busybox, Micropython, zsh, less"
ram = 128
cpus = 2
```

Listado 2: Definición de Guest en Diosix.

Después, en formato de array, se especifican qué sistemas operativos tienen que ser virtualizados durante la ejecución de Diosix, como se muestra en el *Listado 3*. En el ejemplo mostrado se le indica a Diosix que se debe virtualizar un único sistema operativo, el cual ha sido definido en el *Listado 2*. Es posible virtualizar varios sistemas operativos de forma simultánea añadiendo nuevos elementos a esta lista, habiéndolos definidos previamente tal y como se ha mostrado.

```
# RV64GC: this is the default target, loading one guest only by default
[target.riscv64gc-unknown-none-elf]
guests = [ "riscv64-linux-busybox-micropython" ]
```

Listado 3: Ejecución de un Guest con Diosix.

El proceso de compilado y ejecución de Diosix está automatizado por medio de la herramienta *Just* [43]. Sin embargo, al tratar de compilar y ejecutar Diosix desde el código fuente aparece el error mostrado en el *Listado 4*. En este error se indica que el código fuente tiene dependencias que se han deprecado y no han sido actualizadas.

```
error[E0557]: feature has been removed
  --> /home/saulftobias/.cargo/registry/src/index.crates.io-6f17d22bba15001f/lock_api-
  0.3.4/src/lib.rs:91:42
   |
91 | #![cfg_attr(feature = "nightly", feature(const_fn))]
   |                                          ^^^^^^^^ feature has been removed
   |
   = note: split into finer-grained feature gates
```

Listado 4: Error de dependencias de Diosix.

En el repositorio del código fuente encontramos además un *issue* [44] en el que se notifica de este mismo problema, a lo que el desarrollador responde informando que en ese momento se encuentra trabajando en proyectos relacionados, y que solventará el error cuando termine.

Al igual que Jailhouse, el repositorio de Diosix lleva más de 3 años sin actividad. Aunque el error de dependencias mostrado pueda ser en principio sencillo de solucionar, el hecho de que Diosix no vaya a tener soporte ni actualizaciones en el futuro previsible hace que ésta no sea una opción preferente de cara al futuro laboratorio.

3.5. Xvisor

Xvisor [39] es un hipervisor open source bare metal desarrollado por Anup Patel entre otros contribuidores para las arquitecturas ARM, x86_64 y RISC-V entre otras.

Una de las particularidades que presenta Xvisor es que no proporciona un manual con las funciones y procedimientos que se pueden realizar con el hipervisor, sino que defienden un proceso de documentación en el propio código fuente a partir de *Doxygen* que, afirman, permite una mayor mantenibilidad del código. Únicamente es posible encontrar una guía para la ejecución del mismo, pero no existe información detallada de las capacidades de este hipervisor, que permitan una evaluación previa sobre si se ajusta o no a las necesidades del proyecto.

Aunque tal afirmación sea cierta y realmente el comprender como el hipervisor está implementado te permite comprender cuáles son sus capacidades y cómo explotarlas, creo que esta opción es útil para los perfiles de usuarios que se orienten más al desarrollo y contribución de código, pero que es poco práctica de cara al usuario general.

Aproximaciones de este tipo podrían tener sentido en asignaturas en las que el funcionamiento del hipervisor sea el foco principal del temario impartido, pero se alejan de los objetivos docentes que se plantean en el laboratorio de computación. Un ejemplo de esta impracticidad es el hecho de que, analizando los *ChangeLogs* de las releases, sabemos que el hipervisor tiene mecanismos de memoria compartida, pero sin un análisis del código fuente no podemos saber cómo utilizar estas características del software.

Esta situación hace que, en caso de acabar eligiendo Xvisor como base para la virtualización de los sistemas operativos, haga falta dedicar recursos a crear una guía de usuario partiendo del análisis del código fuente de la herramienta. Dado que dicha tarea excede el alcance de este trabajo, y en vista de que existen otras alternativas más prometedoras, entendemos que Xvisor no es una opción preferente de cara a la

futura implementación del entorno descrito en la *Sección 2.2*.

No obstante, se ha realizado un despliegue en QEMU del hipervisor junto con Linux para analizar los posibles problemas que puedan existir durante este proceso. Tanto para compilar el código del hipervisor como el código del guest, Xvisor recomienda el uso de diferentes toolchains. Las opciones que se proponen son la toolchain proporcionada por ARM o la toolchain proporcionada por el sitio web oficial del kernel de Linux. No es posible utilizar la toolchain de ARM porque no permite hacer compilación cruzada desde x86 a RISC-V, por lo que en este caso es necesario utilizar la toolchain proporcionada por GNU.

Sin embargo, esta toolchain no incluye *glibc*, la librería para C de GNU, por lo que no es posible compilar ficheros que contengan referencias a estas librerías. Esto ocurre, por ejemplo, cuando se intenta compilar el RootFS, el sistema de ficheros que se monta en el directorio raíz de la máquina emulada. Si no se es consciente de esta casuística es posible obtener errores informando de falta de librerías necesarias.

Por este motivo, sumado a la falta de documentación y a la existencia de alternativas más prometedoras, se descarta el uso de Xvisor como opción preferente de cara al propósito de este trabajo.

3.6. RVirt

RVirt [41] es un hipervisor bare metal open source específico para la arquitectura RISC-V desarrollado por el MIT PDOS (grupo de Sistemas Operativos Paralelos y Distribuidos del Laboratorio de Ingeniería Informática e Inteligencia Artificial del MIT) que actualmente está orientado a su ejecución dentro de la máquina virtual genérica *Virt* de QEMU, aunque parcialmente tiene soporte para la placa *HiFive Unleashed* de SiFive. Una de las particularidades de RVirt es que, al igual que Diosix, no necesita de soporte hardware (extensión para hipervisores) para ejecutarse, ya que utiliza mecanismos de *trap and emulate* para lograr la virtualización de los sistemas.

Este hipervisor, aunque tiene todas las funcionalidades básicas para funcionar con varios sistemas operativos virtualizados, no ha terminado su desarrollo y no posee tampoco ninguna documentación, aparte de una breve demo que se puede ejecutar sobre QEMU. Además, al igual que ocurre con otros de los hipervisores mencionados, el proyecto ha sido aparentemente abandonado, ya que, a fecha de la redacción de este trabajo, hace más de 5 años que se recibió la última aportación al repositorio que aloja el proyecto, por lo que, nuevamente, no se esperan actualizaciones en un futuro previsible.

Por otro lado, dentro de la escasa documentación del proyecto no se hace referencia a ningún tipo de mecanismo que pueda ser explotado para los propósitos de este trabajo, como pueden ser mecanismos de memoria compartida o mecanismos de comunicación entre sistemas operativos. Sin embargo, al ser aparentemente un hipervisor que en el estado actual es funcional en la arquitectura RISC-V, se han realizado una serie de pruebas de ejecución con el objetivo de comprobar su correcto funcionamiento.

Durante la ejecución de RVirt aparece el error mostrado en el *Listado 5*, indicando que existen regiones de memoria que colisionan entre sí. Concretamente, están colisionando en memoria los binarios de la bios de *OpenSBI* que carga por defecto QEMU (direcciones *0x80000000 - 0x80012630*) y la cabecera ELF del ejecutable del hipervisor (direcciones *0x80000000 - 0x800002ae*).

```
qemu-system-riscv64: Some ROM regions are overlapping
These ROM regions might have been loaded by direct user request or by default.
They could be BIOS/firmware images, a guest kernel, initrd or some other file loaded into
guest memory.
Check whether you intended to load all this guest code, and whether it has been built to
load to the correct addresses.

The following two regions overlap (in the memory address space):
  /usr/share/qemu/opensbi-riscv64-generic-fw_dynamic.bin (addresses 0x0000000080000000
  - 0x0000000080012630)
  target/riscv64imac-unknown-none-elf/release/rvirt-bare-metal ELF program header segment 1
  (addresses 0x0000000080000000 - 0x00000000800002ae)
make: *** [Makefile:38: qemu] Error 1
```

Listado 5: Memory Overlap en la instalación de RVirt.

Este error ocurre cuando se utilizan instalaciones de QEMU que son superiores a la versión 5.2, habiéndose realizado pruebas de ejecución con las versiones 5.2, 6.2, 7.2 y 8.2. Sabiendo que la última aportación al repositorio data de hace más de 5 años se puede deducir que la versión de QEMU con la que se desarrolló debió ser anterior a la versión 5.0. Revisando los ChangeLog de QEMU, prestando atención a posibles cambios que pudieran provocar este error, se puede encontrar la siguiente entrada en el ChangeLog de Quemu 5.1: *OpenSBI loaded by default for virt and sifive_u machines.*

Esta entrada implica que, probablemente, en versiones anteriores de QEMU no se especificase una bios por defecto que debiera usarse durante el proceso de emulación. La siguiente prueba realizada ha consistido en tratar de ejecutar RVirt especificando en los argumentos de QEMU la opción *-bios none*. Al lanzar la ejecución se obtiene el error mostrado en el *Listado 21* del *Anexo A*. El error hace referencia a la Platform-Level Interrupt Controller, un componente de las placas RISC-V encargado de conectar las fuentes de interrupción con los destinatarios de dichas interrupciones. Revisando nuevamente los ChangeLog de QEMU, se puede observar cómo el *System Controller Device Tree* (estructura que permite reconocer los componentes hardware presentes en el sistema) ha sido modificado. No es posible determinar con certeza si este cambio es la raíz del error mencionado, pero merece la pena repetir la prueba de ejecución con una versión de QEMU anterior a los cambios mencionados.

Tanto la instalación de la versión 4.0 como 4.2 de QEMU provocan una serie de errores que necesitan ser tratados mediante modificaciones en los ficheros de QEMU, tal y como se expone en el *Anexo B*.

Finalmente, con la versión 4.2 de QEMU se consigue realizar de forma exitosa la prueba de ejecución de RVirt. Sin embargo, el hecho de que sea un proyecto que ha sido abandonado durante su desarrollo, que no posee características especialmente atractivas para los fines de este trabajo y que además no sea compatible con versiones modernas de QEMU y, por tanto, se desconozca si este será funcional sobre hardware real, hace que no se considere a RVirt como un posible candidato recomendable para la implementación del futuro laboratorio de computación.

3.7. KVM

KVM [18] son las siglas de Kernel-based Virtual Machine, y es el hipervisor open source que proporciona Linux para las arquitecturas x86, AMD y ARM principalmente, aunque también tiene soporte para PowerPC y S390. Más recientemente, desde la versión 5.16 Linux comienza a dar soporte también para la virtualización sobre la arquitectura RISC-V.

KVM es entonces una infraestructura de virtualización incorporada en el kernel de Linux, lo que permite a Linux actuar como un hipervisor. Es por tanto el propio módulo del kernel de Linux el responsable de gestionar y virtualizar los recursos físicos del sistema, como la CPU o la memoria del sistema.

Por otro lado, KVM se basa en QEMU, haciendo a éste responsable de la emulación de los dispositivos front-end de entrada/salida en el espacio de usuario, mientras que el sistema operativo anfitrión (Linux) es utilizado como el *backend* de los dispositivos de entrada-salida, a través de llamadas al sistema.

QEMU-KVM presenta un gran número de funcionalidades de interés para el desarrollo de este proyecto. Por un lado, con QEMU-KVM es posible definir regiones de memoria compartida, con la posibilidad de generar interrupciones derivadas de los accesos a memoria por parte de los diferentes sistemas operativos. De esta manera, los sistemas operativos guest pueden acceder a una región de memoria compartida como si fuera un dispositivo PCI. Esta región de memoria está implementada directamente en el host como un objeto de memoria compartida, o a través de un servidor de ivshmem [8] [10] [27].

Pese a que KVM es un módulo del kernel de Linux ampliamente utilizado en el mercado, este se encuentra deshabilitado en la imagen de ubuntu-server que proporciona Ubuntu para la arquitectura RISC-V. Una posible explicación de esto es que el soporte de KVM por parte de QEMU se da desde la versión 7.0 del emulador, mientras que en el momento de la redacción de esta memoria la versión de QEMU disponible en los repositorios de Ubuntu es la versión 5.2. Otra posible explicación es que KVM sobre RISC-V sea todavía una característica experimental que puede provocar algún tipo de inestabilidad en el sistema, y que por ese motivo no esté activada por defecto. De igual manera, el paquete *QEMU-kvm* no está disponible en el repositorio oficial de Ubuntu.

Una forma sencilla de comprobar la presencia del módulo de kvm es comprobar si dentro del sistema de ficheros existe o no alguna entrada en *path /dev/kvm* con, por ejemplo, el comando *file*, tal y como se muestra en el *Listado 6*. Se puede ver como en este caso la salida que se devuelve en la terminal indica que el módulo no está presente en el sistema.

```
1 ubuntu@ubuntu:~$ file /dev/kvm
2 /dev/kvm: cannot open `/dev/kvm` (No such file or directory)
```

Listado 6: Comprobación de la presencia del módulo de KVM.

Por estos motivos ha sido necesario recompilar el kernel del sistema operativo que estaba siendo emulado, forzando la presencia de este módulo. Una vez descargado el código fuente del kernel, es necesario realizar los pasos indicados en el *Apéndice C* para llevar a cabo su instalación. Cabe destacar que el modulo de KVM está marcado como *EXPERIMENTAL* dentro de las opciones de compilación mostradas.

Después de instalar el nuevo kernel habiendo activado el módulo de KVM, es posible comprobar cómo ahora este sí está presente en el sistema siguiendo el mismo proceso que se mostró en el *Listado 6*. En este caso se obtiene la siguiente salida: */dev/kvm: character special (10/232)*, con lo que se puede ver que ahora el módulo de virtualización si se encuentra presente.

3.8. Bao

Bao [35] es un hipervisor open source bare metal desarrollado por miembros de la Universidad de Minho, Portugal, disponible para las arquitecturas ARM y RISC-V, aunque el desarrollo de Bao no ha concluido,

es posible que nuevas arquitecturas estén disponibles en un futuro.

Bao está pensado para ser un hipervisor ligero que incluya el overhead mínimo posible en la ejecución de sus sistemas operativos guest. Este está orientado a la ejecución de sistemas críticos en conjunto de otros sistemas que no tengan esta criticidad de forma mixta. Por este motivo, Bao emplea un sistema de partición estática de los recursos que se le asigna a cada uno de los sistemas, con pass-through de toda la entrada/salida a los diferentes guests y mapeando las interrupciones virtuales directamente sobre las interrupciones físicas del sistema.

Aunque el desarrollo de Bao no haya terminado y existan diferentes funcionalidades que puedan no estar implementadas todavía, a diferencia de otros casos mencionados, el proyecto no ha sido abandonado por sus desarrolladores, por lo que este sigue recibiendo regularmente actualizaciones. Sin embargo, en el estado actual del proyecto no existe documentación acerca de Bao, sus capacidades, o la forma de utilizarlo. No obstante, existe una pequeña demo disponible para la arquitectura RISC-V en la que se muestra cómo ejecutar dos sistemas operativos entre los que se implementa un mecanismo de envío de mensajes.

Esta demo utiliza por un lado Linux y por otro lado FreeRTOS [32], un sistema operativo orientado a la ejecución de tareas de tiempo real. En esta demo se muestra cómo, a través de la UART, es posible comunicar ambos sistemas operativos. Por un lado FreeRTOS maneja esta comunicación de forma asíncrona, recibiendo una interrupción cada vez que un nuevo mensaje está disponible. Sin embargo, y debido a que el correspondiente driver no está implementado [26], Linux requiere de mecanismos de polling o encuesta continua para manejar los mensajes enviados desde FreeRTOS.

Pese al estado temprano en el que se encuentra el desarrollo de Bao, tanto las cualidades que presenta como la comunicación a través de mecanismos de memoria compartida y UART, junto con el origen académico del proyecto, hace que Bao sea una opción interesante y prometedora de cara a los objetivos propuestos en este trabajo.

Para tratar de obtener más información acerca de Bao y tratándose de un proyecto académico, se ha intentado contactar con los desarrolladores del software con la intención de abrir una vía de colaboración entre ambos proyectos. En esta propuesta se explicaban los objetivos de este proyecto y se solicitaba, en caso de ser posible, cualquier avance relativo a la documentación del hipervisor, de modo que se simplificase el desarrollo de cualquier prueba posterior y pudiendo entender de mejor manera cuáles son las cualidades de Bao y de qué modo se puede adaptar a los propósitos de este trabajo. Sin embargo, a día de hoy no se ha recibido contestación por parte del equipo de Bao.

Dadas sus cualidades y pese a no ser un proyecto terminado, por la misma naturaleza del mismo, considero a Bao como una opción a tener en cuenta a futuro de cara al desarrollo del laboratorio de computación, al ser una alternativa que simplifica el proceso de virtualización mediante la asignación estática de recursos. El pass-through de la entrada/salida es una característica atractiva de cara al uso de componentes como los pines I/O de las placas utilizado durante el desarrollo de las prácticas por parte del alumnado, y el hecho de que las demos disponibles se centren en la comunicación de ambos sistemas operativos hace pensar que esta será una de las características principales del hipervisor. Todo esto hace que Bao se convierta en una opción preferente respecto al resto de alternativas expuestas, y que merece por tanto un mayor número de pruebas y un análisis más exhaustivo de su implementación y capacidades.

4. Sistemas Operativos Ligeros

En el *Capítulo 2.2* se detallaron las diferentes estrategias de trabajo que se planteaban para este proyecto. Ambas propuestas se basaban en el uso de un sistema operativo completo, que sirviese como interfaz para el usuario, y un sistema operativo poco restrictivo que permitiese la libre ejecución del código ensamblador del alumno.

La diferencia principal entre ambas propuestas radica en qué sistema operativo aloja el núcleo de depuración, el sistema operativo menos restrictivo, comunicándose y siendo manejado por la interfaz, o el sistema operativo más capacitado, depurando de forma conjunta tanto al código del alumno como al sistema operativo que lo ejecuta.

En ambos supuestos, especialmente en el segundo, es interesante que el sistema operativo que ejecuta el código del alumno sea lo más simple posible, de modo que este interfiera lo mínimo posible en la ejecución del código. Con este pretexto, en este capítulo se detallan una serie de candidatos de sistemas operativos livianos que sirvan como base para el desarrollo del futuro laboratorio.

4.1. Egos2000

Egos2000 [52] es un sistema operativo desarrollado con fines docentes disponible para la arquitectura RISC-V, tanto a través de QEMU, como sobre placas físicas. Esta sistema operativo centrado en la simplicidad (contando con exactamente 2000 líneas de código) tiene como objetivo permitir que cualquier estudiante entienda el funcionamiento del mismo con tan solo leer el código fuente.

Este sistema operativo de código abierto desarrollado por Yunhao Zhang, Doctor en Ciencias de la Computación por la Universidad de Cornell, se organiza en 3 capas diferentes: *Grass*, *Earth* y *Application*.

- En la capa *Earth* se implementan aspectos relativos a las abstracciones específicas del hardware, como unidades de manejo de memoria, interfaces de disco, relojes, terminales *tty*, etc.
- En la capa *Grass* se implementan abstracciones que no son dependientes del hardware subyacente, como por ejemplo las interfaces de las systemcalls o del bloque de control de procesos.
- En la capa *application* se implementa el sistema de ficheros, shell y comandos de usuario.

El siguiente paso para valorar la viabilidad de este sistema operativo es seguir los pasos que se indican en el repositorio, para lograr ejecutar este sobre QEMU. El primer paso de este proceso es compilar el sistema operativo haciendo uso de la toolchain precompilada para RISC-V proporcionada por SiFive. No obstante, este proceso no puede ser realizado, tal y como se muestra en el *Listado 23* del *Anexo A*.

El error citado indica que la toolchain que se enlaza es para arquitecturas de 64 bits, mientras que en el fichero *Makefile*, que se emplea para el proceso de compilado del sistema operativo, se mezclan opciones referentes a toolchains de 32 y 64 bits. Para arreglar este problema se ha modificado dicho *Makefile*, de modo que se utilice la toolchain de 32 bits en lugar de una de 64 bits.

Como SiFive no proporciona dicho recurso de forma precompilada, y como la compilación de esta toolchain acarrea otro conjunto de errores que imposibilitan su instalación tal y como se ha descrito en la *Sección 2.1.1*, queda descartado el uso de esta toolchain para la compilación del sistema operativo.
En el repositorio se indica que es posible también utilizar la toolchain oficial de GNU, por lo que se ha procedido a descargar y compilar dicha toolchain asegurándose de que se compila para la arquitectura RISC-V de 32 bits.

En el nuevo intento de compilación aparece el error mostrado en el *Listado 24* del *Anexo A*. Este error indica la falta de una librería en la toolchain. Es posible que, probando con diferentes herramientas,

se pueda llegar a solventar este error, pero por el momento, dado que las opciones presentadas en la documentación y dada la existencia de otras opciones posibles, se ha considerado que no merece la pena realizar pruebas más profundas.

4.2. FreeRTOS

FreeRTOS [32] es un sistema operativo open source orientado a aplicaciones de tiempo real y microcontroladores. Uno de los principios de diseño de este sistema operativo es la sencillez, lo que se traduce en un kernel escrito en *C* que se desarrolla únicamente en 3 ficheros.

Las características de tiempo real sobre las que se basa FreeRTOS realmente no son algo que tenga demasiada utilidad en el escenario planteado, por lo que no hay un motivo concreto, más allá de su disponibilidad en la arquitectura RISC-V, con los que argumentar su uso en el desarrollo de este proyecto. Sin embargo, es importante conocer este sistema operativo, pues es la base de una de las demostraciones utilizadas por el hipervisor Bao, mencionado en el *Capítulo 3*.

Concretamente, sobre este sistema operativo se implementan los mecanismos de comunicación entre guests desarrollados por Bao, que serán objeto de estudio en el *Capítulo 5*.

4.3. Xv6

Xv6 [42] es un sistema operativo simple desarrollado con fines didácticos y diseñado para enseñar los principios fundamentales de los sistemas operativos. Este sistema operativo fue desarrollado por el MIT (Massachusetts Institute of Technology) en el año 2006 y se concibe como una reinterpretación moderna de Unix Version 6, un sistema operativo lanzado por AT&T en 1975, dadas las dificultades pedagógicas que presentaba este.

Este sistema operativo sigue una arquitectura de kernel monolítico, y actualmente está disponible para arquitecturas como x86, ARM o RISC-V. Al ser un sistema operativo sencillo y por tanto fácil de modificar y adaptar, este sistema podría ser un buen candidato de cara a la futura implementación del laboratorio de computación.

4.4. Mini RISC-V OS

Mini-riscv-os [5] es un sistema operativo de código abierto, desarrollado con fines académicos por la Universidad Nacional de Quemoy, Taiwan. Cuenta con una breve documentación acerca de su implementación disponible, gran parte de ella en chino tradicional, por lo que para hacer uso de la misma requiere de un trabajo previo de traducción y validación de esta.

En el repositorio de *mini-riscv-os* se presentan 10 versiones del mismo en las cuales se van implementando nuevas funcionalidades, que van desde el uso de la *UART* en la versión uno (desde la que se hace un *Hello World)*, hasta la versión 10 en la que se implementan las llamadas al sistema, pasando por *multi tasking*, cambios de contexto o interrupciones. Cada versión del sistema operativo corresponde de este modo con una posible práctica planteada por la institución académica que utiliza este software, de este modo, las versiones van aumentando sus capacidades y complejidad. Esta distribución de contenidos hace además más sencilla la comprensión del código de cara a los desarrollos que se plantean en este trabajo.

Un limitante de este sistema operativo es la falta de documentación. Creemos que este hecho se debe a los fines docentes de la misma, y de que este modo la documentación del sistema operativo estará ubicada en los materiales docentes utilizados durante las clases, como libros, transparencias o guías, pero no están presentes en el repositorio de GitHub. Sin embargo, en caso de ser necesario, queda abierta la posibilidad de solicitar dicho material o incluso abrir una posible vía de cooperación entre ambas universidades.

Se ha comprobado el funcionamiento de este sistema operativo mediante la emulación de sus diferentes versiones. Durante este proceso no se ha detectado nada fuera de lo común, ya que todas las mecánicas mostradas del mismo aparentan funcionar con normalidad. Por ejemplo, en el *Listado 7* se muestra una ejecución de la versión más completa del sistema. Después de arrancar, se ve en el listado como diferentes tareas van alternando sus ejecuciones por medio de interrupciones asociadas a un temporizador y gestionadas por el sistema operativo.

Consideramos este sistema operativo como un buen candidato para alojar el código de ejecución del alumno, ya que dada su simplicidad proporciona un buen punto de partida para las pruebas que se pretenden realizar. Además, este sistema operativo muestra similitudes a xv6 en cuanto a su implementación, pero siendo mucho más reducido.

Estas características hacen que sea un buen candidato para las implementaciones basadas en la aproximación descrita en la *Sección 2.2.2* en la que se depura el sistema operativo ligero al completo, y con ello el código del alumno, ya que introduciría la menor cantidad de ruido posible. Además, este sistema operativo sirve también como un buen punto de partida para implementaciones basadas en la aproximación descrita en la *Sección 2.2.1*, en la que el sistema operativo ligero aloja el núcleo de depuración. Al ser un sistema operativo tan simple permitirá un fácil depurado de posibles errores que se encuentren durante el proceso de integración con el hipervisor. Después, dadas sus similitudes con xv6 en cuando a la implementación, adaptar sistemas operativos de este estilo no debiera resultar demasiado costoso.

Por estos motivos se utilizará este sistema operativo durante las pruebas realizadas en el *Capítulo 5*. Se escoge este sistema operativo y no xv6 dada la sencillez que presenta, simplificando el depurado de posibles errores. No obstante, es posible que, según se vaya desarrollando la plataforma, se requiera de mecanismos que no se encuentran implementados en este sistema operativo, pero sí en xv6, especialmente en el caso de la aproximación en la que el sistema operativo ligero aloja el núcleo de depuración.

De cara a la migración del laboratorio se deberá usar un sistema operativo u otro en función de dos aspectos. En primer lugar, se deberá valorar si Mini-riscv-os es suficientemente completo como para utilizarse en el desarrollo final. Por otro lado, en este supuesto deberá valorarse si es más conveniente añadir las funcionalidades faltantes a Mini-riscv-os, o si por el contrario resulta más sencillo utilizar xv6, eliminando posiblemente las características de este que no pretenden ser explotadas.

```
ubuntu@ubuntu:/$ make qemu
...
HEAP_START = 8000f00c, HEAP_SIZE = 07ff0ff4, num of pages = 521999
TEXT:   0x80000000 -> 0x80004f1c
RODATA: 0x80004f1c -> 0x80005480
DATA:   0x80006000 -> 0x80006004
BSS:    0x80007000 -> 0x8000f00c
HEAP:   0x8008f100 -> 0x88000000
OS start
...
p = 0x8008f100
p2 = 0x8008f500
p3 = 0x8008f700
OS: Activate next task
Task0: Created!
Task0: Running...
Task0: Running...
Task0: Running...
...
Task0: Running...
timer_handler: 3
OS: Back to OS

OS: Activate next task
Task4: Running...
Task4: Running...
...
```

Listado 7: Ejecucion del sistema operativo Mini RISC-V OS.

5. Pruebas desarrolladas

En el presente capítulo se detallarán las pruebas desarrolladas para evaluar la viabilidad técnica y operativa de la migración del laboratorio de computación a la arquitectura RISC-V, utilizando los hipervisores descritos en el *Capítulo 3*, seleccionando aquellos que han resultado más prometedores.

Los hipervisores que se han considerado más interesantes en la sección anterior son KVM y BAO. Por este motivo, en este capítulo se hace un análisis más exhaustivo de sus capacidades, comprobando mediante pruebas prácticas la viabilidad del nuevo laboratorio de computación.

Sobre estos hipervisores se virtualizarán, además, algunos de los sistemas operativos descritos en el *Capítulo 4* debido a la simplicidad de los mismos y a su adecuación con los objetivos del proyecto desarrollado.

5.1. Prueba de uso con KVM

Como se ha explicado en el *Capítulo 3*, KVM es una tecnología de virtualización para el kernel de Linux. En este proyecto se ha usado QEMU como *frontend* de KVM, ocupándose este de tareas como la emulación de los dispositivos de entrada/salida en el espacio de usuario.

De este modo, es necesario añadir la *flag -accel kvm* durante la ejecución de QEMU si se quiere que este actúe como un hipervisor a través de KVM, en lugar de como un emulador, lo que aligera la ejecución de las máquinas virtuales.

El uso de KVM se plantea para el escenario descrito en la *Sección 2.2.2*, en la que un sistema operativo completo hospeda a un sistema operativo reducido a los elementos mínimos necesarios para su funcionamiento. Este segundo sistema se encarga de este modo de alojar y ejecutar el código proporcionado por el alumno, mientras que el sistema operativo anfitrión depura a su guest de forma íntegra, incluyendo consigo el código del alumno.

De cara al trabajo en el laboratorio, el alumnado dispondría de placas RISC-V con un sistema operativo principal desde el que, con herramientas como GDB o similares, depurar el código que ha desarrollado y que está siendo virtualizado con QEMU-KVM. Sin embargo, si el alumno quisiera avanzar o continuar sus desarrollos en casa sin disponer del hardware necesario, se plantearían dos soluciones diferentes.

Por un lado, el alumno podría emular con QEMU el hardware concreto del laboratorio, accediendo a un sistema operativo sobre el que nuevamente utilizaría QEMU-KVM para depurar la ejecución de su código. Este planteamiento pudiera parecer enrevesado, pero trae consigo la ventaja de que, tanto en casa como en el laboratorio, el flujo de trabajo sería igual, haciendo uso de las mismas *toolchains* y herramientas en ambos casos.

Otra posible solución sería que el alumno emulase directamente el sistema operativo liviano, y depurase su código usando su propio sistema operativo como host, lo cual simplifica el proceso de trabajo sin hardware. Sin embargo, en este caso el alumno sí necesitaría un conjunto de herramientas diferente al utilizado cuando se dispone de hardware real, ya que, por ejemplo, será necesario el uso de toolchains que hagan una compilación cruzada entre la arquitectura de origen, posiblemente x86, y RISC-V, mientras que la compilación del sistema operativo y el código en el laboratorio se realizaría de forma nativa.

Por lo tanto, en esta sección se detalla el estudio de la viabilidad de los escenarios planteados. En primer lugar se pretende estudiar el estado de la virtualización con KVM en RISC-V, estableciendo como entorno de trabajo el detallado en el *Capítulo 2.1*.

Como ya se ha explicado en la *Sección 3.7*, el sistema operativo proporcionado por Ubuntu para la arquitectura RISC-V, y que se usará como host en el laboratorio que se ha emulado, no dispone del módulo de KVM necesario para la virtualización. Tal y como se describe en la sección mencionada, ha sido por tanto

necesaria la recompilación del kernel de dicho sistema operativo, habilitando el módulo experimental del kernel de Linux para la virtualización.

Otra de las piezas clave, dentro de la implementación de esta estrategia, es comprobar el funcionamiento del depurador GDB en conjunto con QEMU para los diferentes escenarios planteados. Por lo tanto, en primer lugar se pretende probar el funcionamiento de GDB desde x86 depurando código emulado en RISC-V, y en segundo lugar probar el funcionamiento de GDB en un entorno RISC-V emulado, depurando el código de una máquina virtualizada con KVM.

Para conectar QEMU y el depurador de GNU es necesario añadir los parámetros *-s* y *-S* en la ejecución del primero [9]. La opción *-s* le indica a QEMU que debe escuchar en el puerto *1234* conexiones TCP provenientes de GDB; mientras que la opción *-S* provoca que la máquina se cree pero no se arranque, permitiendo controlar la ejecución completa de esta desde el depurador.

Para las pruebas descritas a continuación se ha modificado el sistema operativo ligero, de modo que, durante su ejecución, únicamente va a ejecutar la función *programa_alumno*, que llama a las funciones definidas en un fichero ensamblador, que podría representar una práctica del alumno. En este caso, el fichero consiste en la suma de dos números en ensamblador.

En la *Figura 4* se muestra un ejemplo muy básico de cómo se puede depurar este código ensamblador a través de GDB desde x86, emulando la arquitectura RISC-V. En esta captura se ve cómo se controla el flujo de la ejecución del código. Mediante el comando *stepi* es posible ejecutar instrucciones una a una. En la captura también se muestra cómo se imprime el valor de la variable *resultado* a través del comando *print*. Existen gran número de opciones, como mostrar las diferentes funciones definidas (esto es útil para empezar el depurado en la función que aloja el código del alumno, como ya se ha descrito previamente), establecer puntos de ruptura, cambiar el valor de las variables, etc.

Cabe destacar que este ejemplo es simplemente una prueba de concepto. En el caso de terminar utilizando soluciones como las ofrecidas con GDB, este no deberá ser más que el núcleo de depuración, pero seguirá siendo necesario desarrollar o implementar una interfaz gráfica (o usar una implementación de las múltiples disponibles) con la que realmente trabaje el usuario final. De este modo, será la interfaz la que utilice todas las opciones que proporciona GDB en lo referente al depurado de código, pero encargándose de presentar los datos y permitir realizar acciones de una forma cómoda y práctica para el usuario.

```
(gdb) stepi
0x800020d8      19          int b = 20;
(gdb) stepi
20          int resultado = suma_asm(a, b);
(gdb) stepi
0x800020e0      20          int resultado = suma_asm(a, b);
(gdb) stepi
0x800020e4      20          int resultado = suma_asm(a, b);
(gdb) print resultado
$1 = 0
(gdb) stepi
suma_asm () at suma_asm.S:8
8           add a0, a0, a1   # Sumar los argumentos
(gdb) stepi
9           ret              # Devolver el resultado
(gdb) stepi
0x800020e8 in programa_alumno () at os.c:20
20          int resultado = suma_asm(a, b);
(gdb) stepi
21          return 0;
(gdb) stepi
```

Figura 4: Depurado básico con GDB.

Sin embargo existen una serie de problemas cuando se trata de realizar este mismo proceso en RISC-V de forma nativa, virtualizando el sistema operativo ligero. Cuando se pretende ejecutar la máquina virtual con los parámetros de depuración ya mencionados se muestra el mensaje de error indicado en el *Listado 8*.

Este error muestra que no es posible habilitar el acelerador KVM, que habilita la virtualización, a la vez que se prepara la máquina para depurarla con GDB (flags *-s -S*). Sobre este problema es importante conocer si es una condición intrínseca a QEMU-KVM, o si bien depende de la arquitectura.

Dado que no se ha encontrado información relativa a la incompatibilidad de KVM y GDB se ha realizado una prueba de ejecución en x86, virtualizando una máquina con QEMU-KVM y depurándola con GDB. Esta nueva ejecución se completa sin mostrar los errores mencionados, por lo que se puede deducir que este error está ligado a la arquitectura RISC-V y a una falta de soporte de QEMU para esta operación, quizás debido al estado experimental del módulo de kvm.

```
1 ubuntu@ubuntu:/01-HelloOs$ sudo qemu-system-riscv64 -accel kvm -nographic -smp 4 -s -S
2     -machine virt -bios none -kernel os.elf
3 qemu-system-riscv64: -s: gdbstub: current accelerator doesn't support guest debugging
```

Listado 8: Fallo al ejecutar QEMU-KVM con las opciones de depurado.

Aunque actualmente no sea posible virtualizar con KVM y depurar en RISC-V al mismo tiempo, es posible continuar realizando el estudio de viabilidad de esta estrategia deshabilitando la virtualización, emulando las máquinas dentro de RISC-V.

Otro problema encontrado durante el transcurso de estas pruebas ha sido la imposibilidad de compilar directamente el sistema operativo liviano dentro de RISC-V. Concretamente, este proceso se detiene mostrando el siguiente error: */home/ubuntu/miniriscvos/10-SystemCall/src/start.s:10:(.text+0x8): dangerous relocation: The addend isn't allowed for R_RISCV_GOT_HI20.*

El error es provocado por la pseudoinstrucción *la sp, stacks + STACK_SIZE*, localizada en el fichero *start.s*. De acuerdo a la ISA de RISC-V [25], esta instrucción sirve para cargar direcciones de memoria, y se compone de las siguientes dos instrucciones: *auipc rd, symbol[31:12]*, la cual suma un inmediato de 20 bits a los 32 bits superiores del PC, y *addi rd, rd, symbol[11:0]*, que se utiliza para añadir un inmediato de 12 bits a un registro de origen, guardando el resultado en un registro de destino.

Sin embargo, buscando la causa del error se han visto casos similares en los que se notifica que la instrucción *addi* provoca problemas con la *ABI R_RISCV_GOT_HI20*. Para tratar de evitar este error se ha descompuesto la operación de suma tal y como se muestra en el *Listado 9*, pero, aunque el error ya no aparece y es posible compilar el sistema operativo, el fichero resultante no muestra ningún tipo de salida cuando se ejecuta con QEMU.

```
la  a7, stacks
la  a8, STACK_SIZE
add a7, a7, a8
mv  sp, a7
```

Listado 9: Instrucciones conflictivas para la compilación nativa de Mini RISC-V OS.

La única solución válida encontrada hasta el momento consiste en compilar el sistema operativo de forma cruzada y descargarlo en la máquina RISC-V emulada. Con este proceso sí que es posible virtualizar y depurar el sistema operativo en RISC-V. Este proceso nos sirve para realizar análisis sobre el funcionamiento de QEMU cuando se ejecuta en una máquina emulada, ya que es uno de los posibles casos de trabajo autónomo del alumno, pero no es un proceso que se pueda mantener de cara al flujo de trabajo normal del alumnado.

De este modo, en el supuesto de que en una futura implementación se quisiera usar esta configuración de hipervisor y sistema operativo, se tendría que estudiar el modo de adaptar el sistema operativo para reemplazar esta instrucción, o bien estudiar el uso de otras toolchains que no induzcan este tipo de errores.

Una vez se dispone del sistema operativo compilado en la máquina emulada, es posible continuar con el proceso de validación que se ha estado siguiendo hasta el momento. Por tanto, el siguiente paso consiste en la emulación (en el momento en el que se tenga soporte de QEMU pasará a virtualización) del sistema operativo liviano que contendrá y ejecutará el código del alumno.

Versiones sencillas de este sistema operativo, como las que se enfocan en el uso de la UART o cambios de contexto, se comportan de la manera esperada sin mostrar fallos o comportamientos anómalos. Sin embargo, se ha observado cómo mecánicas mas complejas, en concreto la gestión de interrupciones, no funcionan como se esperaría.

En el *Listado 7* de la *Sección 4.4* se puede ver el flujo normal de la ejecución de esta prueba, en la que, por medio de interrupciones, se alterna la ejecución de diferentes tareas. Sin embargo, cuando se ejecuta dentro de una máquina RISC-V emulada con QEMU, se obtiene la salida mostrada en el *Listado 10*.

```
ubuntu@ubuntu:/$ make qemu
...
Task0: Created!
Task0: Running...
Fault store!
Fault store!
Fault store!
...
```

Listado 10: Comportamiento no esperado durante la ejecución de Mini RISC-V OS.

Si se analiza el código del sistema operativo se puede ver que lo que separa el flujo normal de ejecución de los mensajes de error mostrados es un salto condicional en la función *trap_handler* del fichero *trap.c* del sistema operativo [23]. Dicha función procesa interrupciones en función del código de causa que tienen

asociado. Lo primero que se comprueba en esta función es el primer bit del código de causa a través de la máscara *0x80000000*, que indica si se trata de una interrupción asíncrona. o de una excepción síncrona [25]. Después se comprueba el código de la interrupción presente en los últimos 12 bits con la máscara *0xfff*. En función de este código se realizan unas funciones u otras, dependiendo del tipo de interrupción.

Tanto el manejo de la interrupción del timer, como el mensaje de error mostrado, tienen el mismo código de causa menos el primer bit, que indica la clase de interrupción recibida.

El envío de estas interrupciones está siendo gestionado por QEMU, ya que este está siendo el encargado de emular el hardware sobre el que el sistema operativo se estaría ejecutando. Sin embargo, se observa que, cuando el SO ligero se emula en RISC-V desde x86, los códigos recibidos son los correctos, mientras que, cuando se emula el SO ligero desde un sistema operativo que a su vez está emulado en RISC-V, el primer bit del código de interrupción cambia.

Para tratar de comprender esta situación se ha abierto un *issue* [15] a QEMU explicando este cambio de comportamiento en función de si hay uno o dos niveles de emulación. En este *issue* recomiendan repetir estas mismas pruebas usando la última versión de QEMU disponible en el momento (8.2) en los dos niveles de emulación, en lugar de la versión que se había utilizado hasta el momento (7.2).

Esta nueva ejecución con la versión 8.2 muestra cómo las primeras interrupciones se gestionan como se espera, es decir, de manera similar a la mostrada en el *Listado 7* de la *Sección 4.4*, pero que al cabo de unas pocas iteraciones se vuelve al comportamiento mostrado en el *Listado 10*.

Esto nos hace creer que existe algún tipo de "bug" referente a los códigos de interrupción cuando se emula una instancia de QEMU dentro de otra. A fecha de la redacción de esta memoria no se ha vuelto a recibir respuesta por parte del equipo de QEMU, pero esperamos que este fallo acabe siendo corregido en alguna versión posterior del emulador.

El conjunto de pruebas descritas en esta sección muestran, de este modo, las posibles dificultades que se pueden encontrar en el caso de querer desplegar un laboratorio en el que se empleen placas RISC-V, usando la estrategia de trabajo descrita en la *Sección 2.2.2* en conjunto con KVM y Mini RISC-V OS.

5.2. Prueba de uso con Bao

Como se expuso en el *Capítulo 3*, Bao es un hipervisor bare metal desarrollado como parte de un proyecto de la Universidad de Minho, Portugal. Una de las mayores limitaciones en lo referente a este hipervisor es la falta de documentación que se tiene sobre el mismo. Para tratar de paliar este problema se ha intentado contactar con el equipo de desarrollo de Bao, con la idea de exponer el proyecto desarrollado y abriendo una vía de cooperación entre ambas Universidades. Sin embargo, a fecha de la redacción de este trabajo, aún no se ha recibido ninguna respuesta, por lo que de cara a las pruebas desarrolladas no se tiene acceso a documentos que expliquen cómo funciona Bao o de qué modo se puede utilizar.

Por este motivo, las pruebas desarrolladas se enfocan en comprender cómo funciona Bao. En primer lugar, se pretende estudiar cómo se configura tanto el hipervisor como los sistemas operativos para crear un entorno virtualizado. Por otro lado, se intentará comprender cómo funciona la comunicación entre sistemas operativos que se expone en la demostración, con el fin de poder aplicar estas funcionalidades con otros sistemas operativos diferentes.

Toda la configuración necesaria para la ejecución de máquinas virtuales en Bao se detalla a través de un fichero en lenguaje *C*, inicializando en él un conjunto de estructuras de datos que están ya definidas dentro del código de Bao. Esta estructura es la mostrada en el *Listado 11* que se ve a continuación. Con el fin de simplificar la comprensión de dicha configuración y evitar opciones que puedan introducir ruido, se han eliminado algunos de los parámetros y opciones que se ha visto que no son necesarios para la ejecución de Bao.

Como se ve, es necesario especificar a través de *arrays* el conjunto de regiones de memoria compartida que se quieren definir y el conjunto de máquinas virtuales que se quieren declarar. El contenido de estas listas son, a su vez, estructuras. Para cada región de memoria compartida inicialmente basta con definir el tamaño de la misma, mientras que para cada máquina virtual es necesario inicializar la estructura definida en el *Listado 12*.

```
1  extern struct config {
2
3      /* Definition of shared memory regions to be used by VMs */
4      size_t shmemlist_size;
5      struct shmem* shmemlist;
6
7      /* The number of VMs specified by this configuration */
8      size_t vmlist_size;
9
10     /* Array list with VM configuration */
11     struct vm_config vmlist[];
12
13 } config;
```

Listado 11: Estructura de configuración principal del hipervisor BAO.

Dentro de esta configuración de la máquina virtual es necesario especificar tanto aspectos relativos a la memoria (tamaño, dirección donde debe cargarse la VM, punto de entrada, etc.), como características propias de la plataforma donde se va a ejecutar el sistema operativo. La inicialización de los atributos relativos a la plataforma debe hacerse atendiendo a la estructura de datos mostrada en el *Listado 13*.

```
struct vm_config {

    struct {
        /* Image load address in VM's address space */
        vaddr_t base_addr;
        /* Image load address in hyp address space */
        paddr_t load_addr;
        /* Image size */
        size_t size;
    } image;

    /* Entry point address in VM's address space */
    vaddr_t entry;

    /**
     * A description of the virtual platform available to the guest, i.e., the virtual
     * machine itself.
     */

    struct vm_platform platform;
};
```

Listado 12: Estructura de configuración de máquinas virtuales en el hipervisor BAO.

Dentro de la estructura de la plataforma es necesario configurar tanto el número de CPUs como las regiones de memoria que se le quieren asignar a la máquina virtual. Dentro de esta estructura es posible, además, especificar configuraciones relativas a la comunicación entre máquinas virtuales (IPC) o al uso de dispositivos (e.g. UART). Sin embargo, para las pruebas iniciales no es necesario entrar en configuraciones más avanzadas de este tipo, por lo que se omite el análisis de las estructuras de datos asociadas a dichas características.

```
struct vm_platform {
    size_t cpu_num;

    size_t region_num;
    struct vm_mem_region* regions;

    size_t ipc_num;
    struct ipc* ipcs;

    size_t dev_num;
    struct vm_dev_region* devs;

    struct arch_vm_platform arch;
};

```

Listado 13: Especificaciones hardware de las máquinas virtuales en BAO.

Completando las estructuras descritas se logra configurar los entornos que se quieren virtualizar. Sin embargo, antes de esta estructura, es necesario declarar los sistemas operativos que se quieren virtualizar. Esto proceso se realiza en el mismo fichero *C*, a través de la macro mostrada en el *Listado 14*, especificando un identificador y la ruta al binario del sistema operativo que se quiere virtualizar.

```
1 VM_IMAGE(vm1, "/path/to/vm1/binary.bin");
2 VM_IMAGE(vm2, "/path/to/vm2/binary.bin");
```

Listado 14: Macros de configuración de los sistemas operativos guests en BAO.

Conociendo toda esta información ya debería ser posible configurar Bao para virtualizar un sistema operativo diferente a los exhibidos en las demostraciones disponibles en el repositorio del hipervisor. Para este propósito se ha considerado una de las versiones del sistema operativo *Mini-riscv-os* como la mejor opción dada su simplicidad.

Como se ha mostrado, Bao requiere de un sistema operativo ya compilado y en formato binario (*.bin*). Sin embargo, la compilación del *Mini-riscv-os* deja como resultado un fichero en formato *ELF*. Para convertir del formato *ELF* a un fichero binario es necesario utilizar la herramienta *objcopy* disponible en la toolchain de RISC-V del siguiente modo:

```
1 riscv64-unknown-elf-toolchain/bin/riscv64-unknown-elf-objcopy -O binary os.elf os.bin
```

Listado 15: Prueba de conversión entre los formatos *.elf* y *.bin* de un sistema operativo.

Una vez el sistema operativo ya está en el formato deseado, se puede tratar de ejecutar una máquina virtual del mismo a través de Bao. Lamentablemente, Bao falla en la ejecución del mismo, mostrando el error presente en el *Listado 16*.

```
1 Bao Hypervisor
2 BAO WARNING: trying to flush caches but the operation is not defined for this platform
3 BAO ERROR: unkown synchronous exception (2)
4 BAO WARNING: trying to flush caches but the operation is not defined for this platform
```

Listado 16: Error en la ejecución de un sistema operativo diferente a los proporcionados en BAO.

El error mostrado no permite conocer grandes detalles sobre la causa del mismo, por lo que esta vía de desarrollo queda por el momento bloqueada, debido a la falta de documentación que presenta actualmente

el proyecto de Bao, ya que no se especifica de qué modo debe ser configurado éste o qué adaptaciones es necesario realizar al sistema operativo invitado para la virtualización del mismo.

Como se mencionaba previamente, sobre Bao se requería tratar de comprender por una parte cómo se debía configurar el hipervisor para ejecutar un sistema operativo específico, y por otra parte cómo se implementaba en este la comunicación entre sistemas operativos. Para este segundo punto se tratará de analizar de qué modo se ha implementado la comunicación entre los sistemas operativos Linux y FreeRTOS.

El primer paso dentro de este proceso es comprobar el funcionamiento de esta funcionalidad. Una vez está ejecutándose Bao, desde Linux se pueden enviar mensajes escribiendo dentro del dispositivo */dev/baoipc0*. Al realizar esta operación se puede ver que el mensaje escrito aparece en la terminal que está ejecutando FreeRTOS. Cuando esto ocurre, FreeRTOS envía un mensaje por el mismo canal, notificando el número de interrupciones que se han recibido. Estas comunicaciones se realizan escribiendo el contenido del mensaje en una región de memoria compartida, y notificando dicho envío, en principio, a través de la UART. Una pequeña prueba de este funcionamiento es mostrada en la *Figura 5*.

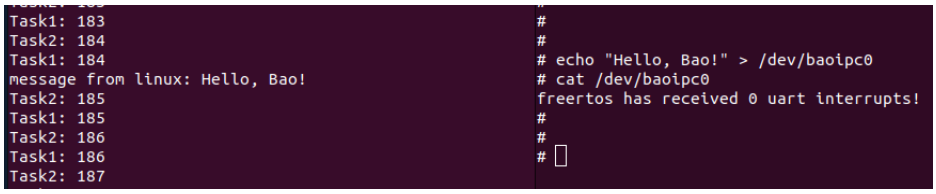

Figura 5: Primera prueba de comunicación entre sistemas operativos guest en BAO.

Como se ve, FreeRTOS escribe un mensaje indicando que no se ha recibido ninguna interrupción en la UART, aunque el mensaje enviado desde Linux sí se haya recibido y mostrado. El comportamiento que se ha detectado durante estas pruebas es que FreeRTOS reconoce como interrupciones de la UART las pulsaciones del teclado, y son estas las que desencadenan el envío de mensajes desde FreeRTOS a Linux, como se muestra en la *Figura 6*, mientras que la recepción de los mensajes se realiza utilizando otra serie de interrupciones asociadas a la escritura en memoria compartida.

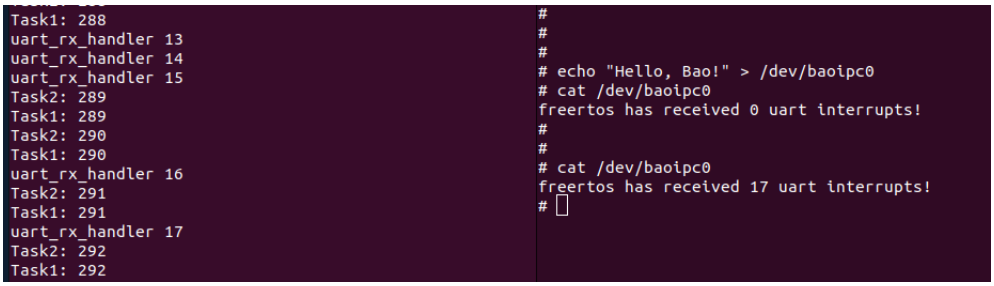

Figura 6: Segunda prueba de comunicación entre sistemas operativos guest en BAO.

Lo primero que cabe destacar es que tanto Linux como FreeRTOS han sido modificados para poder imple-

mentar estos mecanismos. En cuando a Linux, estas modificaciones se implementan a modo de parches, que pueden ser encontrados en el repositorio de Bao [33]. Por otra parte, las modificaciones en FreeRTOS se han implementado directamente sobre el código del mismo. Esta versión de FreeRTOS modificada se encuentra en un repositorio creado por el equipo de desarrollo de Bao, dentro del proyecto del hipervisor [34].

A continuación, se va a proceder a analizar la implementación de estos mecanismos en el lado de FreeRTOS. Dentro de la ejecución de este sistema operativo, lo primero que se observa es la llamada a 5 funciones relacionadas con la configuración de los mecanismos de interrupción y comunicación, mostrados en el *Listado 17*.

En este listado se observa cómo en las líneas 5-8 se realizan diferentes llamadas encargadas de configurar las interrupciones originadas por la UART, mientras que en la línea 10 se realiza una llamada a una función encargada de inicializar los mecanismos de comunicación a través de la memoria compartida. Se ve cómo a la UART se le asigna el ID de interrupción almacenado en la constante *UART_IRQ_ID*. Sin embargo, esta constante no se encuentra definida en ningún lugar dentro del código fuente de FreeRTOS.

El índice asociado a esta interrupción se puede conocer si se analiza la configuración de Bao para este despliegue concreto. Como se ve en el *Listado 18*, dentro de la máquina virtual donde se ejecuta FreeRTOS se declara un único dispositivo, correspondiente con la UART, al que se le asigna la interrupción con ID 10.

```
1  int main(void){
2
3      printf("Bao FreeRTOS guest\n");
4
5      uart_enable_rxirq();
6      irq_set_handler(UART_IRQ_ID, uart_rx_handler);
7      irq_set_prio(UART_IRQ_ID, IRQ_MAX_PRIO);
8      irq_enable(UART_IRQ_ID);
9
10     shmem_init();
11
12     ...
13     //Configuración de las tareas que ejecuta FreeRTOS
14     ...
15 }
```

Listado 17: Configuración de los manejadores de interrupción en FreeRTOS.

```
...
.dev_num = 1,
.devs =  (struct vm_dev_region[]) {
    {
        .pa = 0x10000000,
        .va = 0xff000000,
        .size = 0x1000,
        .interrupt_num = 1,
        .interrupts = (irqid_t[]) {10}
    },
},
...
```

Listado 18: Configuración del índice de interrupción de FreeRTOS en BAO.

Cada vez que llega esta interrupción se ejecuta el manejador asociado a ella, que simplemente incrementa el contador de interrupciones recibidas que tiene declarado, y escribe un nuevo mensaje en la dirección del canal de memoria compartida que tiene definido para este propósito, ubicado en la dirección *0x70000000* y notificando esta escritura a través de la *IPC*, como se muestra en el *Listado 19*

```
char* const freertos_message = (char*)0x70000000;
char* const linux_message    = (char*)0x70002000;
const size_t shmem_channel_size = 0x2000;

void uart_rx_handler(){
    static int irq_count = 0;
    printf("%s %d\n", __func__, ++irq_count);
    shmem_update_msg(irq_count);
    ipc_notify(0);
    uart_clear_rxirq();
}

void shmem_update_msg(int irq_count) {
    sprintf(freertos_message, "freertos has received %d uart interrupts!\n",
        irq_count);
}
```

Listado 19: Manejo de las interrupciones de la UART en FreeRTOS.

Por último, en lo referente a la inicialización de la memoria compartida, se sigue un procedimiento similar al explicado para el caso de las interrupciones de la UART. En primer lugar, se asignan los canales de memoria para los mensajes enviados por cada uno de los sistemas operativos y se escribe un mensaje inicial indicando que se han recibido 0 interrupciones de la UART hasta el momento (mostrado en la *Figura 5*). Posteriormente se asigna un manejador al ID de la interrupción del *IPC* que esta vez, además de en la configuración de la máquina virtual especificada en Bao, si está definido dentro del código de FreeRTOS. El manejador por su parte simplemente lee el mensaje escrito por Linux y lo imprime por

consola, tal y como se observa en el *Listado 20*.

```
#define SHMEM_IRQ_ID (52)
...
void shmem_handler() {
    linux_message[shmem_channel_size-1] = '\0';
    char* end = strchr(linux_message, '\n');
    *end = '\0';
    printf("message from linux: %s\n", linux_message);
}

void shmem_init() {
    memset(freertos_message, 0, shmem_channel_size);
    memset(linux_message, 0, shmem_channel_size);
    shmem_update_msg(0);
    irq_set_handler(SHMEM_IRQ_ID, shmem_handler);
    irq_set_prio(SHMEM_IRQ_ID, IRQ_MAX_PRIO);
    irq_enable(SHMEM_IRQ_ID);
}
```

Listado 20: Manejo de interrupciones asociadas a la memoria compartida en FreeRTOS.

6. Conclusiones

El objetivo principal de este proyecto ha consistido en realizar un estudio del estado del arte de las herramientas disponibles para la arquitectura RISC-V en lo referente a hipervisores y sistemas operativos. Este análisis ha sido motivado por el interés hacia una migración del laboratorio de computación de la Universidad, actualmente basado en ARM, a la arquitectura RISC-V.

El laboratorio actual cuenta con RISC-OS, un sistema operativo que permite la ejecución de código con bajas restricciones, a la vez que proporciona una interfaz fácil de usar. Sin embargo, no se han encontrado alternativas similares existentes en RISC-V. Por ese motivo, se han planteado dos aproximaciones basadas en el uso de hipervisores, de modo que se permita tener de forma simultánea un sistema operativo amigable para el usuario y otro sistema operativo que proporciona un entorno de ejecución con bajas restricciones.

RISC-V es una arquitectura abierta y modular, que, aunque a día de hoy aún no haya alcanzado un estado de madurez, considero que tiene mucha proyección de cara a futuro y que su elección para uso docente en futuros cursos está plenamente justificada. Esto ha motivado mi elección por esta temática a la hora de desarrollar este proyecto, puesto que además de ser muy interesante, me ha permitido conocer con mayor detalle el estado actual de los proyectos que se desarrollan para esta arquitectura.

He visto de este modo que muchos proyectos han quedado abandonados, y que otros tantos aún están en desarrollo, pero que tienen potencial para ser herramientas que se puedan utilizar en un futuro. Sin embargo, la documentación de estos es limitada, por lo que conocer qué capacidades tiene cada una de ellas y cómo se adecuan o se pueden utilizar para los intereses del proyecto no ha resultado sencillo, lo que ha causado que en diferentes instalaciones se hayan tenido que realizar múltiples intentos a modo de "prueba y error".

De estas pruebas, sin embargo, he podido escoger dos opciones preferentes de cara a una posible implementación futura del laboratorio. Por un lado, aunque KVM esté en una fase aún experimental, confío en el éxito del proyecto dado que está respaldado por la *Linux Fundation*. Se ha visto además cómo, a través del emulador QEMU y de la herramienta de depuración GDB, es posible depurar el código del alumno siguiendo una de las aproximaciones descritas en este proyecto, pero que, por el momento, no es posible activar la virtualización en este proceso a causa del estado de las herramientas para la arquitectura.

Por otro lado, se ha encontrado BAO, un hipervisor que ha despertado nuestro interés dadas las capacidades que este oferta y dado que se trata de un proyecto académico de una universidad portuguesa. Creemos que puede incluso llegar a ser posible una futura colaboración con el equipo de desarrollo de BAO, habilitando incluso posibles adaptaciones de la herramienta a los intereses concretos del proyecto. Sin embargo, por el momento no ha sido posible contactar con dicho equipo, y el proyecto, a día de hoy y dado su estado prematuro, prácticamente carece de documentación; por lo que nos hemos visto limitados a la hora de comprender el funcionamiento de este y de profundizar en las pruebas desarrolladas sobre el mismo.

Por último, en lo referente a sistemas operativos, hemos escogido a *Mini-riscv-os* como punto de partida de cara al sistema operativo liviano que aloje la ejecución del código del alumno. Esta elección ha venido motivada por la simplicidad de dicho sistema operativo y las semejanzas que este presenta con otros sistemas operativos como xv6, con los que se trabaja en otros cursos de la titulación. Como interfaz hemos utilizado una versión de Ubuntu server, dado que se trata de un sistema operativo completo, muy versátil y que ya se encuentra presente con relativa madurez para dispositivos basados en la arquitectura RISC-V.

Se concluye, entonces, que la migración del laboratorio actual a la arquitectura RISC-V puede resultar viable, pero que, en el momento de la redacción de esta memoria, las herramientas prometedoras para dicho cometido se encuentran aún en un estado poco maduro en su desarrollo, por lo que será necesario prestar atención a la evolución de las mismas, así como a la del hardware en las que se ejecuten.

Este proyecto me ha permitido profundizar en el estado del arte de las herramientas disponibles para la arquitectura RISC-V, específicamente en relación con hipervisores y sistemas operativos. Considero que este desarrollo ha resultado enriquecedor, ya que me ha proporcionado una visión crítica sobre las limitaciones y el potencial futuro de RISC-V. Además, el proyecto ha fomentado mi capacidad para investigar, analizar y adaptarme a tecnologías emergentes.

6.1. Trabajo futuro

A raíz de la realización de este proyecto se han descubierto un conjunto de acciones o tareas que se deberán realizar a futuro, si finalmente se realiza la migración de equipos que ha motivado la realización de este trabajo.

En primer lugar será necesario desarrollar una interfaz gráfica que sirva como punto de interacción con el usuario. Como se ha explicado a lo largo del proyecto, esta interfaz se localizará en el sistema operativo más capacitado (en los casos propuestos se ha usado Ubuntu como dicho sistema). Además de la interfaz, en el caso en el que se quieran utilizar aproximaciones que no dependan de GDB (como aquellas descritas en el *Capítulo 2.2.2*) se deberá realizar el desarrollo del núcleo de depuración.

En el *Capítulo 2.1.1* se indicaron las ventajas que presentaba usar una toolchain compilada directamente desde su código fuente, como por ejemplo el uso de código más actualizado o la posibilidad de modificación. Sin embargo, se vio cómo durante este proceso se producían errores cuya resolución excede el alcance de este proyecto, ya que se necesita realizar un análisis del código e implementación de dicha toolchain. Queda de este modo como parte del trabajo futuro comprender la causa de estos errores y tratar de solucionarlos, habilitando la posibilidad de compilar la toolchain en lugar de utilizar los binarios precompilados ya proporcionados.

Como se ha visto durante el desarrollo del trabajo, las herramientas disponibles están o bien obsoletas o bien aún en desarrollo, por lo que entre otras cosas no se dispone de la documentación necesaria para su uso. Por este motivo será necesario realizar un seguimiento de las herramientas que durante este trabajo se han catalogado como prometedoras. Por otro lado, puede ser interesante, en el caso de decantarse por una alternativa en concreto, comenzar a realizar un estudio profundo de su implementación, documentando sus capacidades y modo de uso.

Se ha visto cómo el sistema operativo Mini-riscv-os hacía uso de una instrucción en ensamblador que no permitía su compilación nativa dentro de un sistema operativo emulado para la arquitectura RISC-V. Sería interesante tratar de comprender de forma precisa cuáles son las causas de este error, tratando o bien de solucionarlas (por ejemplo buscando una toolchain que sí permita ese tipo de operaciones) o bien de modificar el sistema operativo para prescindir de su uso.

Durante el desarrollo del trabajo se ha detectado una inconsistencia en las ejecuciones de un sistema sistema operativo emulado cuando este se ejecuta dentro de un nivel de emulación o de dos (QEMU anidado dentro de QEMU), motivo por el cual se ha abierto un *issue* notificando esta situación. Será necesario, por este motivo, realizar un seguimiento de la evolución de este ticket.

Por último, se ha explicado cómo, en el momento de la realización de este trabajo, no ha sido posible habilitar la interfáz gráfica del sistema operativo principal al utilizarse la versión de servidores y al no estar disponibles los paquetes necesarios en el repositorio oficial de Ubuntu. Como parte del trabajo futuro se requerirá un seguimiento del estado del sistema operativo principal, comprobando que este añada los paquetes necesarios para la instalación de dicha interfaz a los repositorios oficiales para RISC-V.

Bibliografía

[1] SiFive Inc Andrew Waterman1 Krste Asanovi. *The RISC-V Instruction Set Manual. Último acceso: Junio 2024.* URL: https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf.

[2] Adrián Barredo Ferreira. *Evaluación de la plataforma Raspberry Pi para la docencia de Microprocesadores. Último acceso: Junio 2024.* URL: https://repositorio.unican.es/xmlui/handle/10902/6669.

[3] Fabrice Bellard. *QEMU ChangeLog. Último acceso: Junio 2024.* URL: https://wiki.qemu.org/ChangeLog.

[4] Fernando Vallejo. Carmen Martínez. *Informe Final del proyecto de innovación docente: Docencia de Estructura y Organización de Computadores basada en la Arquitectura ARM, III Convocatoria de Proyectos de Innovación Docente de la Universidad de Cantabria.* URL: https://web.unican.es/consejo-direccion/vcprimeroyprofesorado/Documents/InnovacionDocente/Arm.pdf/.

[5] Ian Cheng, Austin Liu et al. *Mini riscv os repository. Último acceso: Junio 2024.* URL: https://github.com/cccriscv/mini-riscv-os.

[6] No lo tengo claro. *riscv-gnu-toolchain. Último acceso: Junio 2024.* URL: https://github.com/riscv-collab/riscv-gnu-toolchain.

[7] R. Dettmer. "JTAG-setting the standard for boundary-scan testing". En: *IEE Review* 35.2 (1989), págs. 49-52. DOI: 10.1049/ir:19890023.

[8] The QEMU Project Developers. *Device Specification for Inter-VM shared memory device. Último acceso: Junio 2024.* URL: https://www.qemu.org/docs/master/specs/ivshmem-spec.html.

[9] The QEMU Project Developers. *GDB usage. Último acceso: Junio 2024.* URL: https://qemu-project.gitlab.io/qemu/system/gdb.html.

[10] The QEMU Project Developers. *Inter-VM Shared Memory device. Último acceso: Junio 2024.* URL: https://www.qemu.org/docs/master/system/devices/ivshmem.html.

[11] The QEMU Project Developers. *QEMU User Documentation. Último acceso: Junio 2024.* URL: https://www.qemu.org/docs/master/system/qemu-manpage.html.

[12] The QEMU Project Developers. *Supported build platforms. Último acceso: Junio 2024.* URL: https://www.qemu.org/docs/master/about/build-platforms.html.

[13] FentISS. *XtratuM. Último acceso: Junio 2024.* URL: https://github.com/lfd/XtratuM.

[14] FentISSWEB. *XtratuM main page. Último acceso: Junio 2024.* URL: https://www.fentiss.com/xtratum/.

[15] Saúl Fernández. *The cause code of a trap changes when qemu is nested in another qemu. Último acceso: Junio 2024.* URL: https://gitlab.com/qemu-project/qemu/-/issues/2259#note_1839839804.

[16] Pablo Fuentes. *Repositorio de github del UCDebug. Último acceso: Julno 2024.* URL: https://github.com/fuentesp/UCDebug.

[17] The Linux Fundation. *Event Channel Internals. Último acceso: Junio 2024.* URL: https://wiki.xenproject.org/wiki/Event_Channel_Internals.

[18] The Linux Fundation. *KVM Main Page. Último acceso: Junio 2024.* URL: https://linux-kvm.org/page/Main_Page.

[19] The Linux Fundation. *New Xen updates on RISC-V. Último acceso: Junio 2024.* URL: https://xcp-ng.org/blog/2023/05/23/new-xen-updates-on-risc-v/.

[20] The Linux Fundation. *Xen Project. Último acceso: Junio 2024.* URL: https://xenproject.org/.

[21] The Linux Fundation. *Xen Shared Memory and Interrupts Between VMs. Último acceso: Junio 2024.* URL: https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/2251030537/Xen+Shared+Memory+and+Interrupts+Between+VMs.

[22] Sarah Harris y David Harris. "Digital Design and Computer Architecture: ARM Edition". En: *Digital Design and Computer Architecture: ARM Edition*. Cambridge, MA: Morgan Kaufmann, 2016. Cap. 9, Chapter 9. ISBN: 978-0-12-800911-6.

[23] Austin Liu Ian Chen. *trap.c source code: User-Level ISA. Último acceso: Junio 2024*. URL: https://github.com/cccriscv/mini-riscv-os/blob/master/10-SystemCall/src/trap.c.

[24] RISC-V International. *RISC-V. Último acceso: Junio 2024*. URL: https://riscv.org/.

[25] RISC-V International. *The RISC-V Instruction Set Manual: Volume II. Último acceso: Julio 2024*. URL: https://drive.google.com/file/d/17GeetSnT5wW3xNuAHI95-SI1gPGd5sJ_/view.

[26] Sunt-git José Martins. *How to implement two-way communication between virtual machines?. Último acceso: Junio 2024*. URL: https://github.com/bao-project/bao-hypervisor/issues/26.

[27] Michael Kerrisk. *Linux manual page. Último acceso: Junio 2024*. URL: https://www.qemu.org/docs/master/specs/ivshmem-spec.html.

[28] Chongkyung Kil et al. "Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software". En: *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*. 2006, págs. 339-348. DOI: 10.1109/ACSAC.2006.9.

[29] ARM Limited. *ARM Architecture Reference Manual*. URL: https://developer.armcom/documentation/ddi0487/latest.

[30] ARM Limited. *Arm limited roadshow slides Q2 2020*. URL: https://group.softbank/system/files/pdf/ir/presentations/2020/arm-roadshow-slides_q2fy2020_01_en.pdf.

[31] RISC OS Open Limited. *RISC OS Open. Último acceso: Junio 2024*. URL: https://www.riscosopen.org/content/.

[32] Real Time Engineers Ltd. *FreeRTOS Main Page. Último acceso: Junio 2024*. URL: https://www.freertos.org/index.html.

[33] José Martins et al. *Bao Hypervisor. Último acceso: Junio 2024*. URL: https://github.com/bao-project/bao-hypervisor.

[34] José Martins et al. *Freertos over Bao. Último acceso: Junio 2024*. URL: https://github.com/bao-project/freertos-over-bao.

[35] José Martins et al. *Bao repository. Último acceso: Junio 2024*. URL: https://github.com/bao-project/bao-hypervisor.

[36] C. Martínez P. Fuentes C. Camarero y V. Mateev F. Vallejo D. Herreros. *Addressing Student Fatigue in Computer Architecture Courses. IEEE Transactions on Learning Technologies. IEEE. Marzo 2022. DOI: 10.1109/TLT.2022.3163631.*

[37] C. Martínez P. Fuentes C. Camarero y F. Vallejo. *Tecnología low-cost para motivar al alumno. Actas de la XXV Edición de las Jornadas sobre la Enseñanza Universitaria de la Informática (JENUI 2019), Murcia, 3-5 Julio 2019. ISSN: 2531-0607*. URL: https://repositorio.unican.es/xmlui/bitstream/handle/10902/18474/Tecnolog%c3%adalow-cost.pdf?sequence=3&isAllowed=y.

[38] F. Vallejo P. Fuentes C. Camarero y C. Martínez. *La importancia del uso de hardware real en la docencia de Estructura y Organización de Computadores. XVI Foro Internacional sobre la Evaluación de la Calidad de la Investigación y de la Educación Superior (FECIES 2019), Santiago de Compostela, 29-31 Mayo 2019*. URL: https://personales.unican.es/fuentesp/refs/FECIES'19.pdf.

[39] Anup Patel. *Xvisor an open-source bare-metal monlithic hypervisor. Último acceso: Junio 2024*. URL: https://xhypervisor.org/.

[40] Keir Fraser Paul Barham Boris Dragovic. *Xen and the Art of Virtualization. Último acceso: Junio 2024*. URL: https://www.cl.cam.ac.uk/research/srg/netos/papers/2003-xensosp.pdf.

[41] MIT PDOS. *RVirt repository. Último acceso: Junio 2024*. URL: https://github.com/mit-pdos/RVirt.

[42] MIT PDOS. *Xv6 repository. Último acceso: Junio 2024*. URL: https://github.com/mit-pdos/xv6-public.

[43] Casey Rodarmor. *Just. Último acceso: Junio 2024*. URL: https://github.com/casey/just.

[44] Ravi Sahita. *Failed to build due to rust error[E0557]. Último acceso: Junio 2024.* URL: `https://github.com/diodesign/diosix/issues/20`.

[45] Siemens. *IVSHMEM Device Specification. Último acceso: Junio 2024.* URL: `https://github.com/siemens/jailhouse/blob/master/Documentation/ivshmem-v2-specification.md`.

[46] Siemens. *Jailhouse repository. Último acceso: Junio 2024.* URL: `https://github.com/siemens/jailhouse/tree/wip/riscv`.

[47] SiFive. *freedom-tools. Último acceso: Junio 2024.* URL: `https://github.com/sifive/freedom-tools`.

[48] Elena Zaira Suárez Santamaría. *Desarrollo de un entorno de prácticas de laboratorio en ensamblador de ARM para un sistema Raspberry Pi con el sistema operativo RISC OS. Último acceso: Junio 2024.* URL: `https://repositorio.unican.es/xmlui/handle/10902/12612`.

[49] Ubuntu. *Ubuntu RISC-V Qemu Guide. Último acceso: Junio 2024.* URL: `https://wiki.ubuntu.com/RISC-V/QEMU`.

[50] Chris Williams. *Diosix hypervisor. Último acceso: Junio 2024.* URL: `https://diosix.org/`.

[51] DENX Software Engineering Wolfgang Denk. *Das U-Boot. Último acceso: Junio 2024.* URL: `https://github.com/u-boot/u-boot`.

[52] Yunhao Zhang. *Egos-2000 repository. Último acceso: Junio 2024.* URL: `https://github.com/yhzhang0128/egos-2000`.

A. Mensajes de error

En este anexo se agrupan diferentes mensajes de errores encontrados durante la realización de este proyecto.

Error ocurrido durante la ejecución de Rvirt especificando que no se debe utilizar ninguna bios durante el proceso de emulación.

```
qemu-system-riscv64 -machine virt -nographic -m 2G -smp 1  \
-kernel target/riscv64imac-unknown-none-elf/release/rvirt-bare-metal -initrd fedora-vmlinux \
-bios none \
-append "console=ttyS0 ro root=/dev/vda" \
-object rng-random,filename=/dev/urandom,id=rng1 \
-device virtio-rng-device,rng=rng1,bus=virtio-mmio-bus.0 \
-device virtio-blk-device,drive=hd1,bus=virtio-mmio-bus.1 \
-drive file=stage4-disk.img,format=raw,id=hd1 \
-device virtio-net-device,netdev=usernet1,bus=virtio-mmio-bus.2 \
-netdev user,id=usernet1,hostfwd=tcp::10001-:22
panicked at 'PLIC address not specified', src/libcore/option.rs:1188:5
```

Listado 21: Memory Overlap en la instalación de Rvirt.

Error durante la instalación de la toolchain de SiFive en el que se muestra el uso de librerías ilegales durante la compilación de la herramienta.

```
/home/saulftobias/TFM/freedom-tools/obj/x86_64-linux-ubuntu14/install/riscv64-unknown-
elf-gdb-10.1.0-2020.12.8-x86_64-linux-ubuntu14/bin/riscv64-unknown-elf-gdb
Checking dynamic library usage for: /home/saulftobias/TFM/freedom-tools/obj/x86_64-
linux-ubuntu14/install/riscv64-unknown-elf-gdb-10.1.0-2020.12.8-x86_64-linux
-ubuntu14/bin/riscv64-unknown-elf-gdb
  Illegal library used: libncursesw.so.6 => /lib/x86_64-linux-gnu/libncursesw.so.6
    (0x00007fc3746ee000)
  Illegal https://github.com/riscv-collab/riscv-gnu-toolchain/issues/1320library used:
    libtinfo.so.6 => /lib/x86_64-linux-gnu/libtinfo.so.6 (0x00007fc3746bc000)
  Illegal library used: liblzma.so.5 => /lib/x86_64-linux-gnu/liblzma.so.5
    (0x00007fc374691000)
saulftobias@saulftobias-OMEN-by-HP-Laptop-15-dc1xxx:~/TFM/freedom-tools$
```

Listado 22: Memory Overlap en la instalación de Rvirt.

Error ocurrido durante la instalación del sistema operativo Egos2000 utilizando la toolchain de SiFive. En este error se indica que la arquitectura de la toolchain utilizada no corresponde con la arquitectura especificada en el fichero *Makefile*.

```
...
/mnt/tfm/tfm/riscv64-unknown-elf-toolchain/bin/../lib/gcc/riscv64-unknown-elf/10.2.0/
    ../../../../riscv64-unknown-elf/bin/ld: /mnt/tfm/tfm/riscv64-unknown-elf-toolchain/
    bin/../lib/gcc/riscv64-unknown-elf/10.2.0/../../../../riscv64-unknown-elf/lib/
    libc.a(lib_a-reent.o): ABI is incompatible with that of the selected emulation:
target emulation `elf64-littleriscv' does not match `elf32-littleriscv'
...
/mnt/tfm/tfm/riscv64-unknown-elf-toolchain/bin/../lib/gcc/riscv64-unknown-elf/10.2.0/
    ../../../../riscv64-unknown-elf/bin/ld: /mnt/tfm/tfm/riscv64-unknown-elf-toolchain/
    bin/../lib/gcc/riscv64-unknown-elf/10.2.0/../../../../riscv64-unknown-elf/lib/
    libc.a(lib_a-malloc.o): file class ELFCLASS64 incompatible with ELFCLASS32
/mnt/tfm/tfm/riscv64-unknown-elf-toolchain/bin/../lib/gcc/riscv64-unknown-elf/10.2.0/
    ../../../../riscv64-unknown-elf/bin/ld: final link failed: file in wrong format
collect2: error: ld returned 1 exit status
```

Listado 23: Error en la instalación de Egos2000 utilizando la toolchain de SiFive.

Error encontrado durante la instalación del sistema operativo Egos2000 haciendo uso de la toolchain de GNU. El error indica la falta de librerías en la toolchain necesarias para la instalación del sistema operativo.

```
...
In file included from /mnt/tfm/tfm/riscv-gnu-toolchain/bin/sysroot/usr/include/features.h:527,
                 from /mnt/tfm/tfm/riscv-gnu-toolchain/bin/sysroot/usr/include/sys/types.h:25,
                 from library/egos.h:3,
                 from library/servers/servers.c:11:
/mnt/tfm/tfm/riscv-gnu-toolchain/bin/sysroot/usr/include/gnu/stubs.h:8:11: fatal error:
gnu/stubs-ilp32.h: No such file or directory
    8 | # include <gnu/stubs-ilp32.h>
      |           ^~~~~~~~~~~~~~~~~~~
compilation terminated.
make: *** [Makefile:52: build/release/cat.elf] Error 1
```

Listado 24: Error en la instalación de Egos2000 utilizando la toolchain de GNU.

B. Instalación de Qemu 4.X

Durante la instalación de Rvirt se ha visto cómo, dado que hace varios años que este no recibe actualizaciones y dados cambios significativos en el emulador QEMU, es necesario el uso de versiones anteriores a la versión 5.0 del emulador. Sin embargo, este proceso no ha resultado trivial vistos los errores encontrados durante su instalación.

En primer lugar se ha tratado de instalar la versión 4.0 de QEMU. Sin embargo, este proceso se interrumpe a causa del error mostrado en la *Figura 7*. En este error se puede ver cómo existen referencias

que ya no se encuentran, pero que el compilador nos notifica que existen otras de nombre similar. Por ejemplo, el compilador no es capaz de encontrar la referencia a *SIOCGSTAMPNS*, pero, sin embargo, sí que encuentra una referencia a *SIOCGSTAMP_OLD*.

Este tipo de mensajes nos sugieren que en algún punto se ha refactorizado el nombre de esta referencia, deprecando su uso, pero que este cambio no se ha visto reflejado en todo el código del emulador.

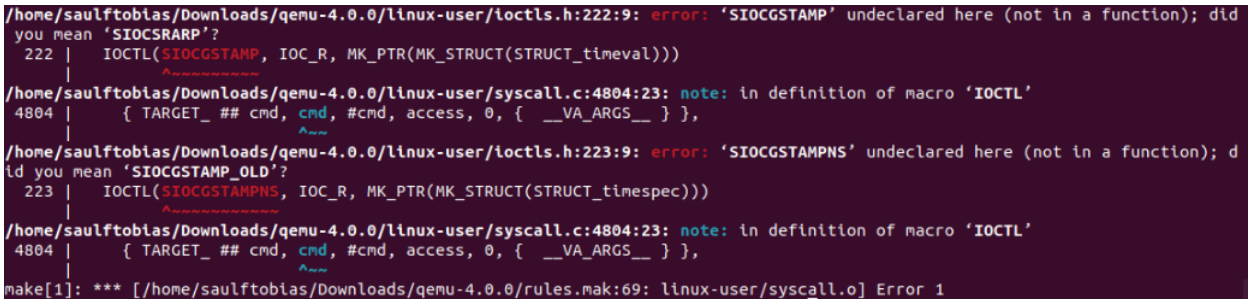

Figura 7: Depurado básico con GDB.

Se ha tratado también de instalar la versión 4.2 de QEMU. Al igual que en el caso anterior, durante este proceso se obtiene un error que interrumpe la instalación. En este caso el error obtenido es el siguiente: *ld: Error: unable to disambiguate: -nopie (did you mean –nopie ?)*.

Si se consulta la documentación de LD se puede ver como este parámetro se encarga de evitar que se produzca un ejecutable que use el mecanismo *Position Independent Executable*, utilizado para aplicar técnicas como *ASLR* [28]. También se puede ver como la sintaxis correcta de esta opción es *-no-pie*.

Si se sustituyen todas las ocurrencias de *-nopie* por *-no-pie* en los ficheros de instalación de QEMU es posible completar la instalación de forma satisfactoria.

C. Instalación del nuevo kernel de Linux

En este apartado se detalla el proceso de instalación y compilación del kernel de Linux activando el módulo KVM. En primer lugar, y una vez descargado el código fuente de dicho kernel, será necesario ejecutar los comandos mostrados en el *Listado 25*.

```
1 make menuconfig
2 make
3 make modules_install
4 make install
5 grub-mkconfig -o /boot/grub/grub.cfg
```

Listado 25: Comandos necesarios para compilar un nuevo kernel de Linux.

El comando *make menuconfig* permite configurar las características que se quieren tener activas en el nuevo kernel. Para obtener los resultados esperados, es necesario acceder al menú de *Virtualization* y marcar la opción *"Kernel-based Virtual Machine (KVM) support (EXPERIMENTAL)"* tal y como se muestra en las imágenes 8 y 9.

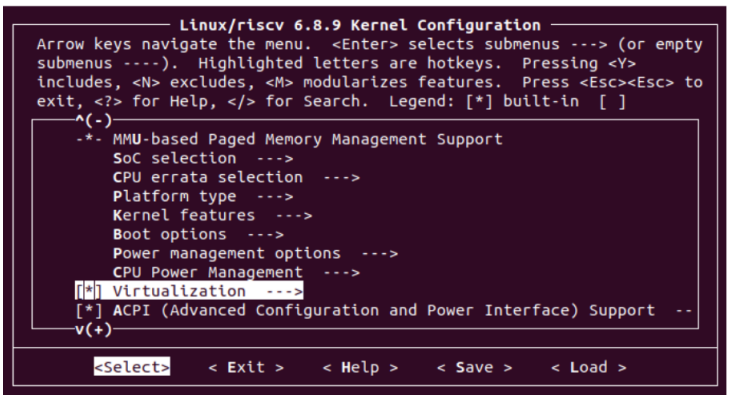

Figura 8: Vista principal del menú de configuración del kernel de Linux.

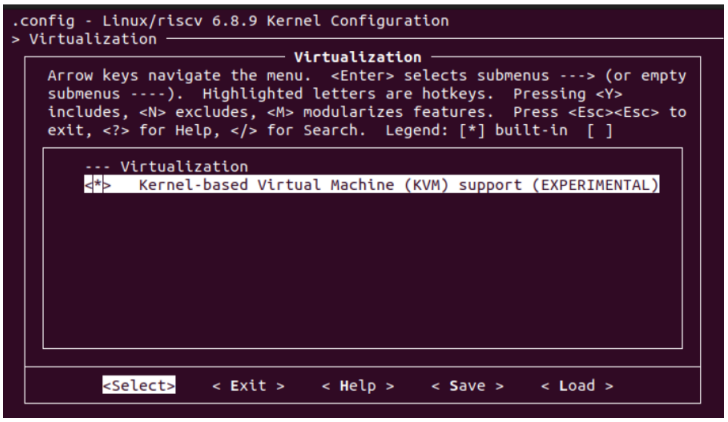

Figura 9: Activación del módulo de KVM en el kernel de Linux.