UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

DISSENY I IMPLEMENTACIÓ D'UN EMULADOR DE BUS CXL SOBRE QEMU

DAVID CAÑADAS LOPEZ

Director/a
XAVIER MARTORELL BOFILL (BARCELONA SUPERCOMPUTING CENTER-CENTRO NACIONAL DE SUPERCOMPUTACION)

Ponente: DANIEL JIMENEZ GONZALEZ (Departamento de Arquitectura de Computadores)

Titulación
Grado en Ingeniería Informática (Ingeniería de Computadores)

Memoria del trabajo de fin de grado

Facultat d'Informàtica de Barcelona (FIB)

Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

01/07/2025

Resumen

Con la esperanza de aportar una herramienta para el desarrollo de tecnologías HPC para RISC-V, el proyecto PSIPE (Prototype for socket-based inter-VM PCIe emulation) tiene como objetivo expandir QEMU para permitir emular el sistema en que el ordenador "host" controlará una serie de "chiplets" RISC-V a través de un bus CXL (PCI Express) de forma transparente al programador, con tal de poder controlar el estado de cada acelerador y hacer offload de tareas a cada uno de ellos para consumar un programa heterogéneo. Una vez completada la infrastructura, se han realizado pruebas con una aplicación de producto de matrices.

Resum

Amb l'esperança d'aportar una eina per al desenvolupament de tecnologies HPC per a RISC-V, el projecte PSIPE (Prototype for socket-based inter-VM PCIe emulation) té com a objectiu expandir QEMU per permetre emular el sistema en què l'ordinador "host" controlarà una sèrie de "chiplets" RISC-V a través d'un bus CXL (PCI Express) de forma transparent al programador, per tal de poder controlar l'estat de cada accelerador i fer offload de tasques a cadascun d'ells per consumar un programa heterogeni. Un cop completada la infrastructura, s'han realitzat proves amb una aplicació de producte de matrius.

Abstract

Hoping to provide a tool for the development of RISC-V HPC technologies, the PSIPE (Prototype for socket-based inter-VM PCIe emulation) project aims to expand QEMU to allow for emulation of the system in which the "host" computer will control a series of RISC-V "chiplets" through a CXL (PCI Express) bus in a way that is transparent to the programmer, in order to be able to control the state of each accelerator and offload tasks to each one of them in order to complete a heterogeneous program. Once the infrastructure has been completed, it has been demonstrated with an application implementing matrix multiplication.

Glosario

- **BAR**: *Base Address Register*. Región de memoria reservada por un dispositivo PCI para comunicarse con el procesador.
- **Benchmark**: Prueba para evaluar el rendimiento de un sistema (software o hardware), o uno de sus componentes, comparándolo con un estándar u otros similares.
- **Chiplet**: Instancia de QEMU que emula un acelerador y recibe tareas a realizar del Master.
- **CXL**: *Compute Express Link*. Ver sección 1.2.2.
- **DARE**: *Digital Autonomy with RISC-V in Europe*. Ver sección 1.1.
- **DMA**: *Direct Memory Access*. Mecanismo mediante el cual un dispositivo hardware puede transferir datos desde o desde memoria sin necesidad de usar el procesador.
- **Driver**: Programa que permite que el sistema operativo controle un dispositivo hardware concreto.
- **GCC**: *GNU Compiler Collection*[18].
- **GDB**: *GNU Debugger*[19].
- **Handle**: Referencia abstracta a un recurso gestionado por otro sistema.
- **Host**: Computadora donde se ejecutan las instancias de QEMU del proyecto.
- **HPC**: *High Performance Computing*. Práctica que consiste en combinar múltiples recursos computacionales para resolver problemas complejos y costosos.
- **IA**: Inteligencia Artificial[8].
- **IRQ**: *Interrupt Request*. Una señal que un dispositivo hardware envía al procesador para indicar que requiere atención.
- **ISA**: *Instruction Set Architecture*. Define el conjunto de instrucciones que un procesador puede entender y ejecutar.
- **Kernel**: Parte fundamental de un sistema operativo, que se ejecuta en modo privilegiado.
- **Master**: Instancia de QEMU que ejecuta el programa principal y distribuye las tareas a los Chiplets.
- **Offload**: Transferir una tarea de procesado (carga de trabajo concreta) de un componente a otro, con tal de mejorar el rendimiento del programa.
- **PCI/PCIe**: *Peripheral Component Interconnect (Express)*. Ver sección 1.2.2.
- **PSIPE/Proto-SIPE**: *Prototype for socket-based inter-VM PCIe emulation*[13]. La implementación de este proyecto.
- **QEMU**: *Quick Emulation*. Ver sección 2.1.4.
- **RISC/RISC-V**: *Reduced Instruction Set Computer (Five)*. Ver sección 1.2.1.
- **VM**: *Virtual Machine*. Ver sección 1.2.3.

Índice

1. Introducción

Este trabajo de fin de grado (TFG) se ha llevado a cabo en el marco de la Facultat d'Enginyeria Informàtica de Barcelona (FIB), perteneciente a la Universitat Politècnica de Catalunya (UPC), y está dirigido por Xavier Martorell Bofill del Barcelona Supercomputing Center (BSC). Este trabajo se sitúa en la mención de Ingeniería de Computadoras dentro del Grado en Ingeniería Informática, enfocándose en aspectos como los sistemas operativos y los modelos de programación.
El objetivo de este documento es el de definir de la manera más ajustada posible los factores relacionados con la gestión de este TFG.

1.1. Contexto

Digital Autonomy with RISC-V in Europe[3] (DARE) es un proyecto europeo a gran escala que pretende desarollar prototipos de sistemas HPC e IA utilizando chiplets diseñados y desarrollados en europa con la última tecnología de fabricación en silicio para lograr el mayor rendimiento y sostenibilidad energética. Se tiene previsto usar una placa principal (o "host") donde se conectarán una serie de aceleradores RISC-V con los que ejecutar programas heterogéneos en los que la carga de trabajo se distribuirá entre estos chiplets.
El objeto final de esta iniciativa es el de desarrollar todas las partes y herramientas necesarias para un supercomputador, notablemente los mencionados procesadores europeos de alto rendimiento y alta eficiencia energética.

1.2. Conceptos

A continuación se describen algunos de los conceptos más importantes de este trabajo, así como los términos que los acompañan.

1.2.1. RISC-V

RISC-V es un estándar abierto de arquitectura de conjunto de instrucciones (ISA) basado en los principios del computador de conjunto de instrucciones reducido (RISC). EL proyecto se inició en 2010 en la Universidad de Berkeley en California, en 2015 se transfirió a la Fundación RISC-V, y más adelante, en 2019, se volvió a transferir a RISC-V International. Sin embargo, el término RISC tiene su origen mucho antes, en la década de 1980.
RISC-V (y otras ISAs basadas en RISC) se distingue de otras ISAs por ser ofrecida bajo licencias de código abierto libres de regalías para cualquier propósito, significando que cualquier persona es libre de fabricar y vender hardware o software basado en RISC-V sin pagar un tributo a nadie, y distribuirlo de forma abierta y gratuita o cerrada y de propiedad exclusiva.
La importancia de esta iniciativa radica en que los proveedores de chips comerciales cobran tarifas de licencia por el uso de sus diseños, patentes y derechos de autor, además de requerir acuerdos de confidencialidad cuando se habla sobre ellos. Sumando a este secretismo el hecho de que el desarrollo de una CPU es una tarea de gran complejidad que requiere conocimiento en muchas áreas de la informática[1], es muy difícil encontrar ISAs modernos de alta calidad para propósitos generales. RISC-V se inició para hacer un ISA de código abierto, disponible para uso académico, y desplegable en cualquier producto para cualquier propósito.

[1] Especialidades tales como la lógica digital electrónica, compiladores, o sistemas operativos.

1.2.2. PCI Express y CXL

Peripheral Component Interconnect Express, o PCI Express (también abreviado como PCIe o PCI-E), es un estándar de bus de expansión de tipo serie creado para sustituir a los anteriores (PCI, PCI-X y AGP). Es la interfaz para placas base más habitual utilizado en conexiones con tarjetas de cualquier tipo o componentes adicionales.
Por otra parte, *Compute Express Link* (CXL) es un estándar abierto de interconexión entre CPU y dispositivos o memoria de alta velocidad y capacidad, diseñado para computadoras de centros de datos de alto rendimiento. CXL está construido sobre la interfaz física y eléctrica en serie PCI Express e incluye protocolos para entrada/salida basada en PCIe (CXL.io), además de protocolos con coherencia de cache para accesos a memoria del sistema (CXL.cache) y de dispositivos (CXL.mem).

1.2.3. Máquina virtual

Una máquina virtual (VM) se refiere a la virtualización o emulación de un sistema informático. Es decir, es un equipo virtual definido por software cuyo objetivo es replicar el comportamiento de un equipo físico concreto. De la misma forma que un ordenador convencional, una máquina virtual cuenta con componentes como una CPU, memoria, e incluso otros dispositivos como discos duros, tarjetas de red, et cétera.
Existen dos tipos de máquinas virtuales:

- **Sistema**. Máquinas virtuales que no tienen correspondencia con ningún hardware existente, y se encargan de proveer un sustituto para una máquina real. En este contexto, nos referimos al hardware físico "real" que ejecuta la máquina virtual como "host" o "anfitrión", mientras que denominamos a la máquina virtual que está siendo emulada como "guest" o "invitada". Un host puede emular a más de un guest al mismo tiempo, siempre que disponga de los recursos necesarios, y cada uno de ellos puede emular diferentes sistemas operativos o plataformas hardware.
- **Proceso**. Máquinas virtuales que se ejecutan como una aplicación usual dentro de un sistema operativo anfitrión con un único proceso simultáneo. Esta máquina virtual sólo existe dentro del contexto del proceso, pues se inicia cuando éste es creado, y se destruye cuando finaliza. Su objetivo es abstraer los detalles del hardware del sistema anfitrión para que el proceso se ejecute en un entorno agnóstico, pudiéndose ejecutar de la misma forma sea cual sea la plataforma que lo contiene.

2. Justificación

Como se ha mencionado antes, en el proyecto DARE se pretende utilizar la arquitectura RISC-V para diseñar los chiplets para los procesadores. Sin embargo, no se dispone de ninguna herramienta especializada en la que simular el entorno que se desea desarrollar para, por ejemplo, poder analizar las partes del proyecto o poder evaluar el rendimiento esperado. Por tanto, aunque lo que se busca es desarrollar una aplicación que pueda ser usada para prototipar estas innovaciones, se enfocará como un trabajo de investigación.

2.1. Elección de herramientas

Resulta crucial determinar la herramienta que se usará para probar el entorno, puesto que será todo nuestro entorno de desarrollo durante todo lo que dure el proyecto, y será el programa con el que estaremos interactuando para contruir las interfaces necesarias para la correcta emulación del sistema que se desea desarrollar eventualmente.
Para este proyecto será necesaria una máquina virtual capaz de emular una arquitectura distinta (RISC-V) a la del ordenador anfitrión, que probablemente sea de tipo x86. Para ello empezaremos analizando algunas de las opciones más conocidas en cuando a la emulación de sistemas, con tal de conocer las distintas alternativas y por qué se ha optado por QEMU.

2.1.1. Bochs

Bochs[10] es un emulador y depurador compatible con arquitecturas IA-32 y x86-64, distribuido como software libre, y capaz de ejecutarse en múltiples plataformas. Sin embargo, no posee la capacidad de emular un sistema RISC-V, por lo que no cumple con los requisistos para este proyecto.

2.1.2. VirtualBox

VirtualBox[15] es un software de virtualización de código abierto desarrollado por Oracle Corporation para arquitecturas x86 de 32 y 64 bits. De igual forma que Bochs, no es compatible con RISC-V, por lo que tampoco se adapta a los requisitos del proyecto.

2.1.3. VMware

VMWare[1] (concretamente el programa VMWare Workstation Player) es otro software de virtualización para la arquitectura x86. La situación es la misma que en los dos casos anteriores, por lo que no es una opción viable al no poder emular RISC-V.

2.1.4. QEMU

El Quick Emulator[20] (QEMU) es un emulador distribuido como software libre y de código abierto capaz de realizar traducción dinámica de binarios para emular procesadores de distintas arquitecturas. Esto quiere decir que QEMU efectúa una conversión del código binario al formato binario equivalente que se ejecuta en la máquina huésped.
El mecanismo del que QEMU hace uso para la traducción dinámica de binarios es el llamado *Tiny Code Generator* (TCG), un compilador just-in-time que reescribe bloques del código a ejecutar en una notación intermedia independiente de la máquina, para más tarde volver a ser compilados a la arquitectura de la máquina anfitriona por el mismo TCG en tiempo

de ejecución. Las arquitecturas con las que el TCG de QEMU es compatible son x86, ARM, PowerPC, RISC-V, MIPS, SPARC, y muchas otras.

2.2. Conclusión

La única opción que cumple todos los requisitos es QEMU, y esto la hace la más viable para el proyecto hasta ahora. Al ser de código abierto y estar documentada no será difícil extenderla para dar soporte a la interconexión de múltiples instancias para lograr simular un sistema similar al que se pretende usar en el proyecto DARE.

3. Alcance

El objetivo de este TFG será el de dar soporte para la emulación de un bus CXL en QEMU a través del cual múltiples máquinas virtuales puedan transferir datos entre ellas mientras trabajan en conjunto para consumar un programa concreto. Es decir, pretendo dar soporte a QEMU para permitir emular el sistema en que el ordenador "host" controlará una serie de "chiplets" RISC-V (los cuales se encontrarán en ejecución en otras máquinas virtuales distintas) a través de un bus CXL de forma transparente al programador, con tal de poder controlar el estado de cada acelerador y hacer *offload* de tareas a cada uno de ellos para luego obtener los resultados.

3.1. Objetivos y requisitos

- **Soporte para RISC-V**. Debemos asegurar que tanto el entorno como las herramientas desarrolladas son compatibles con la ISA RISC-V.
- **Conexión punto a punto**. La conexión entre el host i los chiplets debe ser lo más directa posible, minimizando el overhead que se pudiera introducir por los nuevos mecanismos.
- **Transferencia de datos sustanciales**. Dado que se pretende hacer offload de tareas computacionales a los chiplets, es necesario que la capacidad de transferencia de datos entre el host i éstos sea lo más generosa posible y que se minimizen los posibles cuellos de botella.

3.2. Obstáculos y riesgos

- **Imprevistos durante el desarrollo**: Durante el proyecto pueden surgir imprevistos que afecten a la planificación del proyecto. En nuestro caso lo más probable pueden ser problemas difíciles de diagnosticar -que requerirán tiempo de depuración adicional con las herramientas adecuadas- y problemas en el rendimiento de los programas.
- **Gestión del tiempo**. La documentación del proyecto también puede afectar a la planificación del mismo. Dado que esta parte se solapa con el resto, puede que afecte al desarrollo. Teniendo en cuenta que durante este cuatrimestre estoy cursando dos asignaturas y estoy haciendo prácticas, es posible que no pueda destinar el tiempo al proyecto en los plazos establecidos.

4. Metodología y rigor

La metodología de trabajo es de gran importancia para el proyecto, puesto que contar con las herramientas adecuadas para gestionar los procesos del desarrollo y llevar la cuenta de los cambios realizados puede resultar esencial para nuestro caso.

4.1. Metodología de trabajo

Dada la naturaleza de este proyecto (un programa informático donde su totalidad será código), he optado por la herramienta Git[25]: un software libre y de código abierto desarrollado por Linus Torvalds cuya función es la de gestión de versiones. El uso local de esta herramienta se complementará con un repositorio remoto[13] en el GitLab del BSC.
El uso de Git será esencial para el proyecto, pues llevar la cuenta de los cambios con la granularidad que ofrece el programa puede resultar muy útil a la hora de, por ejemplo, hacer revisiones sobre el código o gestionar múltiples implementaciones diferentes para una misma funcionalidad.

4.2. Seguimiento

A la hora de hacer un seguimiento del trabajo realizado, se programarán reuniones con el director del proyecto para evaluar el progreso realizado cada semana, lo cual resultará de gran ayuda para comentar aspectos sobre el proyecto o dudas sobre la funcionalidad que vayan surgiendo sobre la marcha. También se organizarán reuniones cada dos semanas con el grupo de OmpSs@FPGA en el BSC, con el propósito de informar del estado del proyecto al grupo, intercambiar ideas y responder dudas sobre éste.
Para complementar las reuniones se irán haciendo también pruebas periódicas sobre el funcionamiento del proyecto, para verificar que el estado es correcto y poder actuar sobre errores graves en el programa rápidamente, a medida que surjan.

5. Planificación temporal

El conjunto de la asignatura del trabajo de final de grado se corresponde con 18 créditos. Sin embargo, este proyecto lleva en desarrollo desde el cuatrimestre pasado, en el que cursé 12 créditos de prácticas en los que trabajé en las etapas primerizas del proyecto y senté las bases sobre lo que se desarrollará en este último cuatrimestre. Puesto que un crédito equivale a entre 25 y 30 horas de formación, el tiempo esperado es de entre 750 y 900 horas en total, teniendo en cuenta las de ambos cuatrimestres.
Por tanto, para poder realizar el proyecto y tenerlo listo para la fecha esperada, en esta sección dividiremos el conjunto en tareas, y para cada una estimaremos una duración, riesgo y otras características que ayudarán a su planificación.

5.1. Planificación de las tareas

El proyecto se puede diferenciar en cuatro partes principales: La fase de gestión del proyecto, en la que se define su estructura, metodología y rasgos; La fase de diseño e implementación, donde se desarrolla el cuerpo del proyecto; La fase de documentación y comunicación, en la que se va anotando y discutiendo el progreso en el proyecto; Y la fase de conclusión, en la cual se compila y prepara el trabajo realizado para ser defendido y evaluado. En las horas que se han estimado para cada una de las tareas se incluye un margen para imprevistos que pudieran ralentizar el desarrollo de la tarea en cuestión.

5.1.1. Gestión del proyecto (GP)

- **GP1 – Contextualización y alcance**: Elaboración de un documento definiendo el contexto del proyecto, explicando conceptos clave del proyecto, justificando la solución elegida, y la metodología que se usará durante el desarrollo y seguimiento. Dependencias: Ninguna. Duración: 10 horas.
- **GP2 – Planificación temporal**: Elaboración de un documento desglosando el proyecto en tareas a realizar (y las ya realizadas durante el , y para cada una de ellas estimar su duración, concluyendo el listado con un diagrama de Gantt. Dependencias: GP1. Duración: 10 horas.
- **GP3 – Gestión económica y sostenibilidad**: Elaboración de un documento estimando el presupuesto de costes del conjunto del proyecto y un informe de sostenibilidad. Dependencias: GP2. Duración: 10 horas.
- **GP4 – Elaboración del documento final**: Elaboración de un documento que compile la documentación escrita hasta este punto. Dependencias: GP3. Duración: 10 horas.

5.1.2. Diseño e implementación (DI)

- **DI1 – Preparación del entorno**: Instalación de los programas de edición, control de versiones, emulación y herramientas binarias nativas y de RISC-V. Dependencias: Ninguna. Duración: 20 horas.
- **DI2 – Estudio de librerías**: Investigar las interfaces y mecanismos necesarios en QEMU, el kernel de Linux y las librerías de C disponibles para poder implementar el soporte requerido. Dependencias: DI1. Duración: 20 horas.

- **DI3 – Búsqueda y análisis de ejemplos**: Invesigación de ejemplos de implementaciones sencillas de extensiones de QEMU que permitan emular un dispositivo PCIe sencillo, y la elección de una que estudiar en profundidad, con tal de tener una base sobre la que trabajar. Dependencias: DI2. Duración: 20 horas.
- **DI4 – Adaptación del ejemplo**: Modificación del ejemplo escogido para superar las limitaciones que tiene, como su adaptación a la versión de QEMU deseada, implementar algunas funciones especiales útiles para el proyecto, o eliminar partes innecesarias del código. Dependencias: DI3. Duración: 20 horas.
- **DI5 – Ampliar la ventana de transferencia**: En el ejemplo la capacidad de transferencia de datos por DMA del dispositivo está muy limitada, por lo que se debe diseñar e implementar la posibilidad de enviar y recibir más de una página por transacción, valorando varios mecanismos para superar las limitaciones iniciales. Dependencias: DI4. Duración: 80 horas.
- **DI6 – Desarrollo del intermediario**: Desarrollar un sistema intermediario que permita la comunicación bidireccional entre dos dispositivos de la máquina virtual. Dependencias: DI4. Duración: 80 horas.
- **DI7 – Protocolo de transferencia**: Desarrollar un protocolo que permita la transferencia de una gran cantidad de datos entre usuarios, mediante el intermediario entre dos dispositivos de la máquina virtual. Dependencias: DI6. Duración: 80 horas.
- **DI8 – Depuración**: Configuración del código del kernel Linux, QEMU y el del resto del proyecto para hacer permitir depurar el código de cualquiera de sus partes. Dependencias: DI7. Duración: 10 horas.
- **DI9 – Microbenchmarks**: Añadir conteo del tiempo de ejecución de las funciones de transferencia de datos dentro del código de usuario. Dependencias: DI8. Duración: 5 horas.
- **DI10 – Múltiples chiplets por master**: Soporte para que el usuario pueda hacer offload de tareas a más de un dispositivo simultáneamente, valorando aprovechar todo lo posible el protocolo intermediario o ampliarlo con nuevas funcionalidades. Dependencias: DI7. Duración: 80 horas.
- **DI11 – Adaptación de aplicaciones**: Adaptar varias aplicaciones heterogéneas[2] para demostrar el correcto funcionamiento del sistema. Dependencias: DI10. Duración: 80 horas.
- **DI12 – Comparación y evaluación de CXL**: Evaluar y comprobar si dar soporte a características más específicas de CXL, tales como la coherencia de cache de los accesos a memoria, para las transferencias entre dispositivos. Dependencias: DI10. Duración: 80 horas.
- **DI13 – Benchmarks y optimización**: Modificar las aplicaciones anteriormente adaptadas (o implementar unas nuevas) para poder evaluar el rendimiento del sistema desarrollado. Con los datos obtenidos se evaluará realizar optimizaciones sobre el

[2] Aplicaciones tales como el producto de matrices, la ecuación de Jacobi, el movimiento de partículas y la descomposición de matrices por el método de Cholesky.

sistema para poder aumentar su rendimiento y maximizar su potencial para ejecutar programas más complejos. Dependencias: DI12. Duración: 60 horas.

5.1.3. Documentación y comunicación (DC)

- **DC1 – Seguimiento**: Durante la realización del proyecto se irán anotando los eventos que transcurran para llevar un seguimiento del mismo y ser incluidos en la documentación. Dependencias: Ninguna. Duración: Dado que se hará esporádicamente es difícil estimar una duración exacta, pero se pueden estimar unas 20 horas.
- **DC2 – Documentación**: Elaboración de la documentación del desarrollo del proyecto, además de la revisión y actualización de la previamente realizada. Dependencias: Ninguna. Duración: 100 horas.
- **DC3 – Comunicación**: Se organizarán reuniones semanales con el director del proyecto con tal de presentarle el progreso, dudas o problemas que hayan surgido durante el desarrollo del proyecto. También se organizarán reuniones con el grupo de trabajo para informar sobre el estado del proyecto. Dependencias: Ninguna. Duración: unas 40 horas, repartidas durante todo el proyecto

5.1.4. Conclusión (CN)

- **CN1 – Integración final del proyecto**: Preparar la versión final de la aplicación. Dependencias: DI13. Duración: 10 horas.
- **CN2 – Revisión de la memoria**: Fin de la redacción de la memoria del proyecto, y su revisión. Dependencias: DC2, CN1. Duración: 20 horas.
- **CN3 – Preparación de la defensa**: Preparación del material, presentación y puesta a punto del sistema para ser expuesto y evaluado. Dependencias: CN2. Duración: 20 horas.
- **CN4 – Defensa del proyecto**: La defensa del proyecto realizado frente el tribunal. Dependencias: CN3. Duración: 1 hora.

5.2. Recursos

En este apartado se detallan los recursos necesarios para realizar el proyecto, sean de personal, material o software. Una vez conocidos los recursos, se deben asociar las tareas descritas anteriormente para más adelante estimar un presupuesto.
Para cada recurso de personal y de software se especifica una leyenda en cursiva, que se usará en las tablas como abreviación una vez se asocien a las tareas del proyecto.

5.2.1. Recursos humanos (Roles)

- **Director de proyecto** (*D*): Encargado de establecer el alcance, las características y las tareas necesarias del proyecto, así como controlar su realización y asignación. También se encarga de la comunicación con el cliente.
- **Analista** (*A*): Encargado de diseñar los diversos elementos que constituyen el proyecto, además de determinar los efectos de las decisiones que se han ido tomando. También se encarga de redactar la documentación y memoria.

- **Programador** (P): Encargado de traducir los diseños a código funcional, además de depurar el proyecto y programar las pruebas necesarias.

5.2.2. Recursos materiales

- **Espacio de trabajo**: Dada la naturaleza del proyecto, es posible realizarlo desde casa o el campus de la UPC sin problemas. También se cuenta con una mesa reservada en el centro BSC-Repsol que será usada esporádicamente.
- **Computadora**: El BSC ha proporcionado un portátil Dell Latitude 7450, con un procesador Intel Core Ultra 7, disco duro SSD de 512GB, y 16GB de memoria RAM. Este ordenador será usando tanto para el desarrollo como para las pruebas.

5.2.3. Recursos digitales (Software)

- **Vim**[14] (V): Editor de texto para desarrollar el código del proyecto.
- **Git** (G): Junto con GitLab serán las herramientas de control de versiones.
- **QEMU** (Q): Herramienta de emulación de procesadores.
- **Firefox** (F): Navegador de internet para buscar documentación o consultar el estado del repositorio, entre otros.
- **GCC**[18] (C): Compilador de C con soporte para varias arquitecturas.
- **GDB**[19] (D): Depurador de código para múltiples lenguajes de programación.
- **Overleaf**[16] (O): Editor web de LaTeX colaborativo para la redacción de documentos.
- **GanttProject**[7] (N): Software libre de edición de diagramas de Gantt.

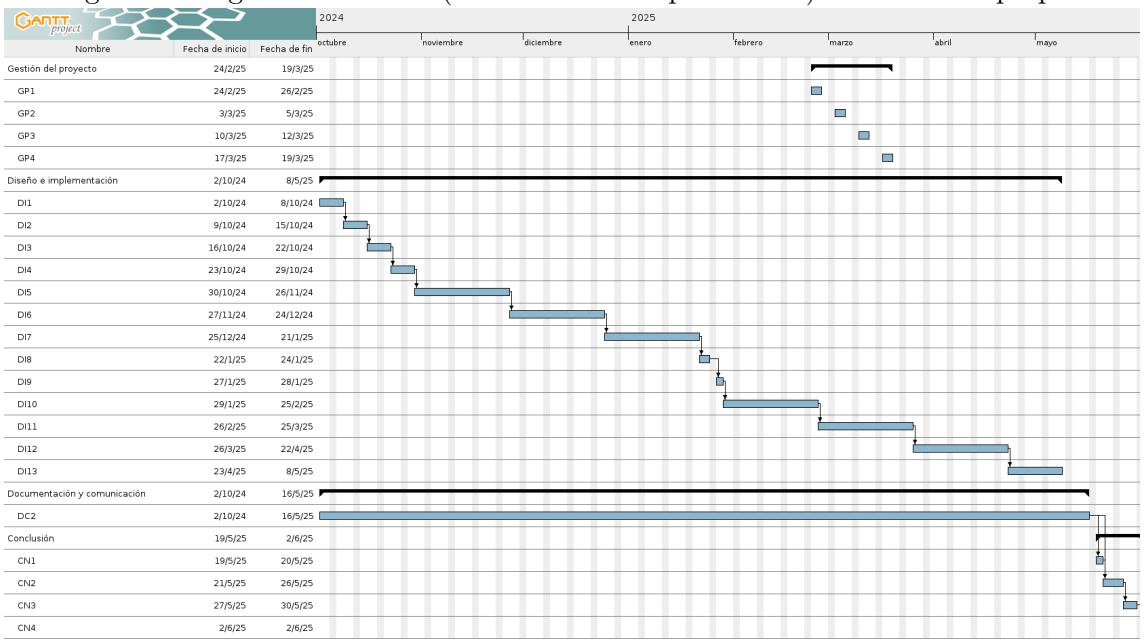
5.3. Resumen de las tareas

Cuadro 1: Requisitos y roles por tarea. Elaboración propia.

Id.	Tarea	Horas	Roles	Software
GP	**Gestión del proyecto**	40	–	–
GP1	Contextualización y alcance	10	D	O
GP2	Planificación temporal	10	D	O, N
GP3	Gestión económica y sostenibilidad	10	D	O
GP4	Elaboración del documento final	10	D	O
DI	**Diseño e implementación**	635	–	
DI1	Preparación del entorno	20	A, P	G, V, F
DI2	Estudio de librerías	20	A, P	V, F
DI3	Búsqueda y análisis de ejemplos	20	A	F
DI4	Adaptación del ejemplo	20	P	G, V, C, Q
DI5	Ampliar la ventana de transferencia	80	P	G, V, C, Q
DI6	Desarrollo del intermediario	80	A, P	G, V, C, Q, F
DI7	Protocolo de transferencia	80	A, P	G, V, C, Q, F
DI8	Depuración	10	P	V, C, Q, D, F
DI9	Microbenchmarks	5	P	G, V, C, Q
DI10	Múltiples chiplets por master	80	A, P	G, V, C, Q, F
DI11	Adaptación de aplicaciones	80	P	G, V, C, Q, F
DI12	Comparación y evaluación de CXL	80	A, P	G, V, C, Q, F
DI13	Benchmarks y optimización	60	P	G, V, C, Q, D, F
DC	**Documentación y comunicación**	160	–	–
DC1	Seguimiento	20	D, A, P	–
DC2	Documentación	100	A	O
DC3	Comunicación	40	D, A, P	–
CN	**Conclusión**	51	–	–
CN1	Integración final del proyecto	10	P	O
CN2	Revisión de la memoria	20	A	O
CN3	Preparación de la defensa	20	D	–
CN4	Defensa del proyecto	1	D	–
–	**Total**	886	–	–

5.4. Diagrama de Gantt

Figura 1: Diagrama de Gantt (fecha de defensa provisional). Elaboración propia.



Nombre	Fecha de inicio	Fecha de fin
Gestión del proyecto	24/2/25	19/3/25
GP1	24/2/25	26/2/25
GP2	3/3/25	5/3/25
GP3	10/3/25	12/3/25
GP4	17/3/25	19/3/25
Diseño e implementación	2/10/24	8/5/25
DI1	2/10/24	8/10/24
DI2	9/10/24	15/10/24
DI3	16/10/24	22/10/24
DI4	23/10/24	29/10/24
DI5	30/10/24	26/11/24
DI6	27/11/24	24/12/24
DI7	25/12/24	21/1/25
DI8	22/1/25	24/1/25
DI9	27/1/25	28/1/25
DI10	29/1/25	25/2/25
DI11	26/2/25	25/3/25
DI12	26/3/25	22/4/25
DI13	23/4/25	8/5/25
Documentación y comunicación	2/10/24	16/5/25
DC2	2/10/24	16/5/25
Conclusión	19/5/25	2/6/25
CN1	19/5/25	20/5/25
CN2	21/5/25	26/5/25
CN3	27/5/25	30/5/25
CN4	2/6/25	2/6/25

6. Gestión económica

Una vez definido el tiempo asignado a cada una de las tareas del proyecto, el material necesario, y los roles que participarán, se puede comenzar la gestión económica del proyecto. Para el contexto de este proyecto, consideramos los siguientes costes:

- Costes del personal: Salarios del director del proyecto, el analista y el programador.
- Costes de material: Materiales y herramientas del personal.
- Costes energéticos: Consumo eléctrico del equipamiento utilizado.

6.1. Costes del personal

Teniendo en cuenta los tres roles que se han establecido para este proyecto, y las tareas asignadas a cada uno de ellos, podemos determinar el coste de cada tarea, y por tanto el coste total, asociado al personal del proyecto.
Los sueldos de cada uno de los roles se han estimado tras una conversación con el director del proyecto sobre los sueldos aproximados de cada rol en el BSC.

Cuadro 2: Salarios anuales y por hora del personal. Elaboración propia.

Rol	**Sueldo bruto (anual)**	**Sueldo bruto (por hora)**
Director de proyecto	60000€	34.09€
Analista	20000€	11.36€
Programador	20000€	11.36€

Una vez estimados los salarios por hora de cada rol, podemos obtener los de cada tarea. Sobre el coste total del personal se debe añadir una cantidad adicional, asociada a la contribución a la Seguridad Social e impuestos adicionales. Este valor se ha estimado como un 30 % del sueldo de cada rol, y por tanto se debe incrementar el coste del personal en este mismo porcentaje.
Algo que cabe destacar en el cálculo de los costes, y que puede dar paso a confusión, es que en las tareas de diseño e implementación (DI) se ha supuesto que el analista (A) y el programador (P) no trabajan simultáneamente, sino que primero el analista diseña la solución y la estudia, y después el programador la implementa y depura.

Cuadro 3: Costes de personal por tarea. Elaboración propia.

Id.	Tarea	Horas	Roles	Coste
GP	**Gestión del proyecto**	40	–	1363.6€
GP1	Contextualización y alcance	10	D	340.9€
GP2	Planificación temporal	10	D	340.9€
GP3	Gestión económica y sostenibilidad	10	D	340.9€
GP4	Elaboración del documento final	10	D	340.9€
DI	**Diseño e implementación**	635	–	7668€
DI1	Preparación del entorno	20	A, P	454.4€
DI2	Estudio de librerías	20	A, P	454.4€
DI3	Búsqueda y análisis de ejemplos	20	A	227.2€
DI4	Adaptación del ejemplo	20	P	227.2€
DI5	Ampliar la ventana de transferencia	80	P	908.8€
DI6	Desarrollo del intermediario	80	A, P	908.8€
DI7	Protocolo de transferencia	80	A, P	908.8€
DI8	Depuración	10	P	113.6€
DI9	Microbenchmarks	5	P	56.8€
DI10	Múltiples chiplets por master	80	A, P	908.8€
DI11	Adaptación de aplicaciones	80	P	908.8€
DI12	Comparación y evaluación de CXL	80	A, P	908.8€
DI13	Benchmarks y optimización	60	P	681.6€
DC	**Documentación y comunicación**	160	–	4544.6€
DC1	Seguimiento	20	D, A, P	1136.2€
DC2	Documentación	100	A	1136€
DC3	Comunicación	40	D, A, P	2272.4€
CN	**Conclusión**	51	–	1056.69€
CN1	Integración final del proyecto	10	P	113.6€
CN2	Revisión de la memoria	20	A	227.2€
CN3	Preparación de la defensa	20	D	681.8€
CN4	Defensa del proyecto	1	D	34.09€
–	**Total**	886	–	19022.75€
–	Total (bruto)	–	–	14632.89€
–	Total (adicional empresa)	–	–	4389.86€

6.2. Costes de material y energéticos

En este proyecto el material fundamental utilizado es el hardware y el software sobre el que se desarrolla. En cuanto a la estancia donde se trabaja, el coste de esta puede ser ignorado ya que el proyecto puede ser realizado remotamente desde casa sin ningún coste adicional asociado. Por tanto, a pesar de que el BSC dispone de estaciones de trabajo disponibles, en esta sección no se tendrá en cuenta este coste.
Dado que los costes energéticos están directamente relacionados con los componentes electrónicos en uso y el tiempo que estén funcionando, se tratarán en esta misma sección.

6.2.1. Software

Todos los programas informáticos utilizados en el proyecto son completamente gratuitos: no requieren de pagos o suscripciones, por lo que se ignorarán estos costes.

6.2.2. Hardware

El componente electrónico más significativo para el desarrollo del proyecto es el portátil que ha sido proporcionado por el BSC para desarrollar el proyecto, y en este hardware en el que se basarán todos los precios de material y costes eléctricos.
En primer lugar se calculará el precio del portátil en sí, que se corresponderá con el coste amortizado con su tiempo de uso durante el proyecto. Para ello, se ha tomado como coste la fracción de su precio original calculada como $P \times H/U$, donde:

- P: Precio total del dispositivo.
- H: Horas de uso del dispositivo.
- U: Horas útiles del dispositivo (tiempo de vida). Se ha estimado un tiempo de vida de 4 años (7040 horas).

Cuadro 4: Coste (amortizado) del material. Elaboración propia.

Dispositivo	**Precio**	**Horas en uso**	**Amortización**
Dell Latitude 7450	1600€	886h	201.36€

Para obtener el coste energético del portátil, se ha estimado que el precio del kilovatio por hora es de 0.19€/kWh. Esta estimación se ha realizado haciendo una consulta[17] sobre el precio para un consumidor promedio. Es probable que si el proyecto se realizara en el centro del BSC este precio fuera diferente (a causa de un contrato distinto con la compañía eléctrica), pero dado que se ha decidido ignorar el coste del material del lugar de trabajo en dicho centro, no se usará.

Cuadro 5: Coste energético. Elaboración propia.

Dispositivo	**Potencia**	**Horas**	**Consumo total**	**Coste**
Dell Latitude 7450	65W	886h	57.59kWh	10.94€

6.3. Recuento

Sabiendo cada uno de los costes cada sección, podemos computar el total para obtener un presupuesto para el proyecto.

Cuadro 6: Cómputo del coste final. Elaboración propia.

Personal	**Material**	**Energético**	**Total (final)**
19022.75€	201.36€	10.94€	19235.05€

7. Sostenibilidad

Evaluar la sostenibilidad del proyecto es importante ya que puede revelar las posibles implicaciones negativas que pudiera en cualquiera de sus dimensiones, ya sean repercusiones económicas, sociales o ambientales. Siendo transparente sobre estos datos del proyecto, es posible buscar formas de hacerles frente y mejorarlos, sea en este proyecto o en futuros.

7.1. Dimensión económica

Como se ha explicado anteriormente, si el proyecto lograra contribuir a la iniciativa DARE, podría causar un impacto positivo en el desarrollo informático en Europa. Es por ello que el presupuesto destinado al proyecto es una buena inversión, sabiendo los beneficios que podría conllevar.
Dado que construir los chiplets del proyecto DARE es una tarea larga y costosa, y una a largo plazo, contar con una herramienta que se ajustase lo máximo posible al sistema objetivo (con tal de lograr una emulación más fiel) puede ahorrar mucho tiempo y dinero al proyecto.

7.2. Dimensión social

Más allá de los conocimientos adquiridos como parte del TFG, este proyecto puede dejar una marca positiva en la sociedad gracias a que el desarrollo de esta herramienta puede ser útil no solo para la iniciativa DARE, sino para cualquier otra persona (o entidad) que desee probar un sistema informático configurado de forma concreta para evaluar la ejecución de aplicaciones distribuidas o heterogéneas, pero que no cuente con una herramienta adecuada para lograrlo.
Es decir, aunque el objetivo de este proyecto es dar soporte para un sistema concreto, se podría extrapolar a otros usos distintos sin mucha dificultad, lo que podría ser útil a mucha gente.

7.3. Dimensión ambiental

Teniendo en cuenta que el desarrollo de este proyecto se basa en su totalidad en la elaboración de un programa informático, el impacDigital Autonomy with RISC-V in Europe (DARE). to medioambiental es muy reducido. Este programa, por su parte, puede ser ejecutado en uno o más ordenadores a la vez (lo que implica un coste energético adicional durante su funcionamiento), pero más allá de este hecho no tiene otros efectos ambientales significativos.
Por otra parte, como se ha explicado en la sección de costes, el único material necesario es un portátil donde se realizarán todas las tareas.

8. Seguimiento intermedio

En esta sección se detallará el estado del progreso en los objetivos, los ajustes a realizar (y su justificación), y la planificación de cara al final del proyecto.

8.1. Planificación y objetivos

A continuación se revisarán los objetivos que han sido establecidos para el proyecto en la entrega inicial (para la asignatura de GEP), y el estado de cada uno de ellos.

8.1.1. Soporte para RISC-V

Este objetivo consistía en asegurar que tanto la solución desarrollada como las herramientas utilizadas son compatibles con la ISA RISC-V.
En este aspecto del desarrollo no ha habido problemas, ya que me he asegurado de antemano que todas las herramientas que he seleccionado para el proyecto cuentan con soporte para la arquitectura. El único factor que ha ralentizado el progreso ha sido, en cierto punto, tener que compilar el kernel de Linux para RISC-V manualmente, porque ninguna de las distribuciones existentes que contaba con una versión binaria para RISC-V (mayoritariamente Debian) se ajustaba a las necesidades del proyecto.
Las herramientas en cuestión son las siguientes:

- **Linux**. El kernel Linux cuenta con soporte para RISC-V, de 32 y 64 bits. La versión utilizada es la 6.6.72.
- **QEMU**. Tal y como fue explicado en la *Fita Inicial* del proyecto, QEMU cuenta con el TCG, que permite la ejecución de código para arquitecturas distintas a la del procesador del host. Concretamente, soporta RISC-V para 32 y 64 bits. La versión utilizada es la 9.1.2.
- **GDB**. Usando el programa *gdb-multiarch* es posible depurar el código del kernel y del driver de la VM. La versión utilizada es la 12.1.
- **GCC**. El compilador GCC soporta hacer *cross-compilation* para RISC-V. La versión utilizada es la 11.4.

8.1.2. Conexión punto a punto

Este objetivo consistía en minimizar los overheads que implican la nueva capa de comunicación entre el host i los chiplets.
El mecanismo de comunicación entre un host y un chiplet está dividido en varias partes: una de usuario (donde el programador comunica al kernel una operación que desea realizar sobre una longitud de bytes concreta), una del kernel (un driver de Linux que prepara los datos del usuario y entabla la comunicación con el *proxy*), y una a nivel de VM (un dispositivo virtual de QEMU, el *proxy*, que es configurado por el driver y se encarga de leer/escribir memoria y recibir/enviar datos con otro *proxy* de otra VM).
En el diseño actual, algunos de los factores que limitan esta comunicación son:

- **Las llamadas a sistema**. El tiempo que se tarda en entrar al código del driver desde que el usuario lo indica no es algo que se pueda controlar directamente.

- **Las regiones críticas**. Dado que el dispositivo de QEMU se comunica con el driver a través de una interrupción, y sabiendo que el kernel es reentrante, hay algunas partes del código que han tenido que ser protegidas.
- **La transferencia por sockets**. La comunicación entre *proxys* se realiza a través de sockets, y el *thoughput* del que son capaces al enviar datos no es algo que se pueda controlar directamente.
- **Las transferencias por DMA**. El dispositivo de QEMU lee y escribe en memoria del usuario a través de operaciones DMA que programa. La velocidad a la que se realizan tampoco es algo que podamos controlar directamente.
- **La virtualización del procesador**. El hecho de emular una arquitectura no-nativa introduce overheads al usar el TCG de QEMU para poder ejecutar el código dentro de la VM. Al no ser un hardware real, el rendimiento está sujeto a la programación de QEMU, y aunque bueno, difícilmente será el ideal.

8.1.3. Transferencia de datos sustanciales

Este objetivo consistía en permitir la capacidad de transferir grandes cantidades de datos entre el host i los chiplets, con tal de permitir tareas computacionalmente complicadas trabajar con mucha información.
En el diseño que he desarrollado, la cantidad de datos que se pueden enviar o recibir en una sola operación del usuario viene determinada por el espacio disponible en el dispositivo *proxy* de QEMU. Ampliando o reduciendo de antemano determinados valores en su código (antes de la compilación) manualmente, es posible ajustar la cantidad de referencias a páginas de datos que pueden estar temporalmente almacenadas en el *proxy* con el propósito de evitar llamar al kernel cada vez que haga falta usar una nueva, ahorrando overheads.
Actualmente el valor por defecto que uso es 131072, es decir, el dispositivo puede tener almacenadas hasta 131072 referencias a páginas de usuario en su memoria. Entonces, suponiendo que el tamaño de página es de 4kB, esto significa que en una sola operación es posible gestionar 512MB sin tener que volver a iniciar una nueva. Sin embargo, si la dirección de memoria donde inician los datos a leer/escribir no está alineada al tamaño de página, es posible que se use una referencia adicional.

8.2. Ajustes y justificación

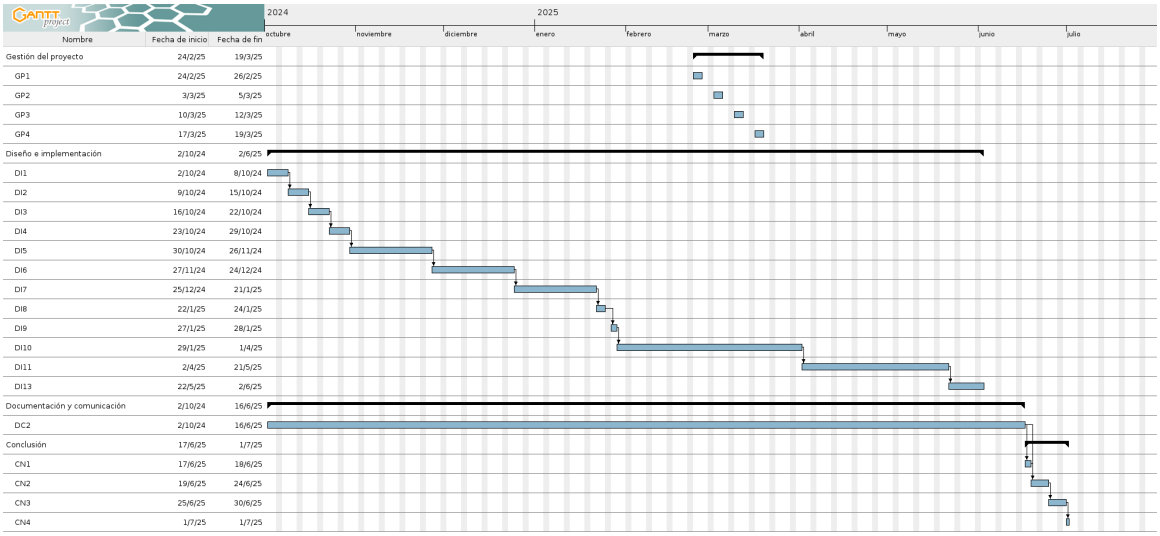
Varios problemas han alargado la etapa definida como DI10 (*Múltiples chiplets por master*) en la planificación inicial. La causa es que, al tener que desarrollar un código que interactúe con el sistema de gestión de memoria del kernel, hay muchos detalles que se pueden pasar por alto en un primer momento, y que al ocurrir un fallo puede no ser evidente lo ocurrido. Es por ello que en el estado actual aún no se ha logrado la comunicación entre un master y más de un chiplet para realizar un mismo programa de forma heterogénea. Aun así, el sistema que he desarrollado está perfectamente preparado para soportarla, lo único que lo detiene es la forma en que interactúa con el sistema de gestión de memoria (que provoca un error).

8.3. Planificación prevista

Preveo poder solucionar el problema y lograr la comunicación entre varios chiplets y un master, y luego realizar la adaptación de las aplicaciones para demostrar que el sistema es completamente funcional (DI11). Asumo que esto será sencillo una vez logre reparar la comunicación entre las distintas VMs.
Si el problema resultase ser más complejo de lo esperado, tendría que seguir adelante con la versión que comunica un único chiplet con el master. No sería la versión ideal, pero como "proof of concept" puede servir. Aun si este fuera el caso, el progreso que he realizado y el sistema desarrollado sería mostrado y defendido como corresponde.
Si bien la etapa de evaluación de funciones específicas de CXL (DI12) puede no ser posible por tiempo, espero poder preparar algunos benchmarks y evaluar el rendimiento del programa (DI13), potencialmente con una versión distinta a la anticipada.

8.3.1. Diagrama de Gantt

En la Figura 2 se muestra un diagrama de Gantt (de elaboración propia), actualizado a partir de la planificación prevista en la entrega inicial del proyecto, para tener en cuenta la situación actual y los ajustes a su planificación.



Nombre	Fecha de inicio	Fecha de fin
Gestión del proyecto	24/2/25	19/3/25
GP1	24/2/25	26/2/25
GP2	3/3/25	5/3/25
GP3	10/3/25	12/3/25
GP4	17/3/25	19/3/25
Diseño e implementación	2/10/24	2/6/25
DI1	2/10/24	8/10/24
DI2	9/10/24	15/10/24
DI3	16/10/24	22/10/24
DI4	23/10/24	29/10/24
DI5	30/10/24	26/11/24
DI6	27/11/24	24/12/24
DI7	25/12/24	21/1/25
DI8	22/1/25	24/1/25
DI9	27/1/25	28/1/25
DI10	29/1/25	1/4/25
DI11	2/4/25	21/5/25
DI13	22/5/25	2/6/25
Documentación y comunicación	2/10/24	16/6/25
DC2	2/10/24	16/6/25
Conclusión	17/6/25	1/7/25
CN1	17/6/25	18/6/25
CN2	19/6/25	24/6/25
CN3	25/6/25	30/6/25
CN4	1/7/25	1/7/25

Figura 2: Diagrama de Gantt actualizado según la planificación prevista.

9. Arquitectura del sistema

En esta sección se detallará la arquitectura final del sistema desde una perspectiva de diseño: las partes que lo componen, la función de cada una de ellas, y las operaciones disponibles.

9.1. Objetivo

El objetivo del sistema diseñado es cumplir con las expectativas planteadas en la sección 3, es decir: Expandir QEMU para permitir emular el sistema en que el ordenador "host" controlará una serie de "chiplets" RISC-V (también instancias de QEMU) a través de un bus CXL (PCI Express) de forma transparente al programador, con tal de poder controlar el estado de cada acelerador y hacer *offload* de tareas a cada uno de ellos para luego obtener los resultados.
Para ello, basaré el proceso de los programas de usuario en la transferencia mutua de unidades discretas de datos: Recibir o enviar información a partir de una dirección base de memoria y una longitud determinada, las cuales son usadas por el sistema para configurar y ejecutar cualquier operación intermediaria que fuera necesaria.

9.2. Capas

Para lograr que dos programas de usuario distintos en dos instancias de QEMU diferentes puedan comunicarse es necesario estructurar el código en varias "capas":

- **Usuario**: Esta es la capa más familiar, ya que consiste en los programas de usuario que serán diseñados o adaptados para operar con los datos que el programa necesite. En el caso de la multiplicación de matrices (ver sección 11.2) esta parte es donde se aloja espacio para las matrices, inicializa sus valores, u opera con sus celdas.
- **Kernel**: Esta capa consiste en un módulo del kernel de Linux, concretamente un driver para el dispositivo (ver siguiente punto), que será el intermediario entre el programa de usuario y la parte del código externo a la VM. Se debe encargar de gestionar los datos que el usuario pide para asegurarse de que se transfieren o reciben sin problemas.
- **Dispositivo**: Esta capa es donde se encuentra el dispositivo virtual CXL (PCIe) de QEMU. Se encargará de recibir las órdenes del driver y realizar la comunicación a nivel de VM-VM.

En la Figura 3 se muestra un esquema de las capas en una situación uno a uno (dos instancias de QEMU). Con esta estructura podemos arrojar algo de luz acerca del diseño final, además de ser el motivo por el que el código del proyecto está dividido en tres partes diferenciadas.

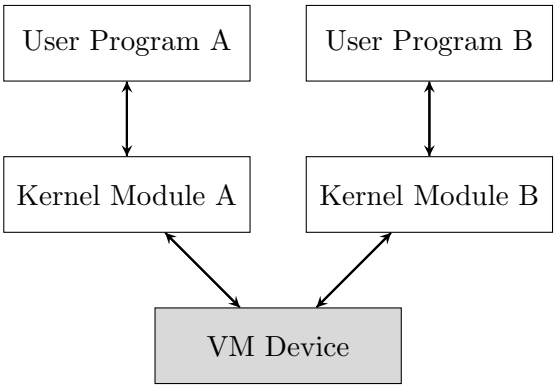


Figura 3: Configuración del entorno QEMU CXL por capas: Usuario, Kernel y Dispositivo.

9.3. Protocolos

La división entre las capas también implica una comunicación diferente entre cada una de ellas, ya que interactúan con el sistema operativo desde un contexto diferente: Por una parte, la capa de usuario deberá hacer uso de algún mecanismo de entrada al sistema para comunicarse con la capa del kernel, mientras que ésta deberá hacer uso de otros drivers u otras funcionalidades internas del kernel Linux para comunicarse con el dispositivo de QEMU. El dispositivo PCIe, por su parte, tendrá a su disposición las funciones que implementa QEMU, además del contexto de usuario de la máquina host, donde se está ejecutando el programa de la VM.
Para que la comunicación entre estas partes se puede establecer, y las transferencias de datos se puedan completar, hacen falta por tanto varios protocolos que puedan regir las distintas conexiones. Estos protocolos son necesariamente distintos entre cada capa, pues como hemos visto el contexto de ejecución es distinto para cada una de ellas.
Las estrategias usadas son las siguientes:

- **Usuario-Kernel**: Una llamada a sistema permite entrar en el código del driver para establecer la comunicación. Es a través de esta funcionalidad que se permite el paso de parámetros a las funciones del driver, entre los cuales se incluye el código de la operación concreta que realizar, y si fuera necesario también la dirección base de los datos con los que operar y la longitud de los mismos. El driver se debe asegurar de que las opciones que ha indicado el usuario tienen el formato correcto (dirección válida en memoria, longitud adecuada).

- **Kernel-Dispositivo**: El kernel configura el dispositivo y se asegura de que sus registros sean accesibles (para lectura y escritura). A través de éstos, el driver puede proporcionar la información necesaria al dispositivo para que pueda realizar sus operaciones. Una vez ha finalizado, el dispositivo hace uso de una interrupción para que el driver sea invocado de nuevo.

- **Dispositivo-Dispositivo**: Las VMs se ejecutan en procesos distintos, lo que tiene varias implicaciones. Primero, cada VM tendrá que disponer de su propio dispositivo PCIe propio (en vez de uno "compartido", como aparece en la Figura 3). Segundo, estos dispositivos no pueden compartir memoria directamente, por lo que he usado una herramienta estándar (sockets) para que puedan comunicarse. A partir de ahora

me referiré al conjunto de funciones del dispositivo de QEMU que se encargan de gestionar la comunicación entre dos instancias como el "proxy" de dicho dispositivo.

Sin embargo, la transferencia de información entre capas no puede hacerse arbitrariamente: se debe seguir un cierto orden en las operaciones entre ellas para garantizar que dos instancias independientes son capaces de colaborar para completar una operación conjunta. Es por ello que distinguiremos las siguientes etapas en el proceso de comunicación:

- Etapas de los protocolos *Usuario-Kernel* y *Kernel-Dispositivo* (diseño)

  1. **Petición**: El usuario inicia una petición (como el envío o recepción de datos) al kernel a través de una llamada a sistema establecida de antemano.
  2. **Atención**: El driver atiende la petición, comprueba y prepara los datos del usuario. Una vez preparada, la operación se añade a una cola interna de peticiones pendientes. Si la cola estaba vacía, se continúa a la fase 3. En caso contrario, se retorna a modo usuario.
     Excepción: Si la operación es una que no requiere de coordinación especial con el dispositivo (como una de consulta, de espera o de gestión de la cola), se resuelve inmediatamente sin necesidad de encolarla, volviendo a modo usuario al finalizar.
  3. **Configuración**: El driver consulta la primera operación pendiente y configura el dispositivo como corresponda mediante operaciones de entrada/salida. Al finalizar la configuración de los registros del dispositivo, se escribe en un registro especial que indica que la ejecución debe empezar. El programa retorna a modo usuario hasta que se reciba la interrupción del dispositivo, momento en que se pasará a la etapa 4.
  4. **Conclusión**: Al recibir una interrupción del dispositivo, el driver moverá la operación actual (la primera de la cola de operaciones pendientes) al final de otra cola distinta, la de operaciones finalizadas. Luego, se pasa a la etapa 3.

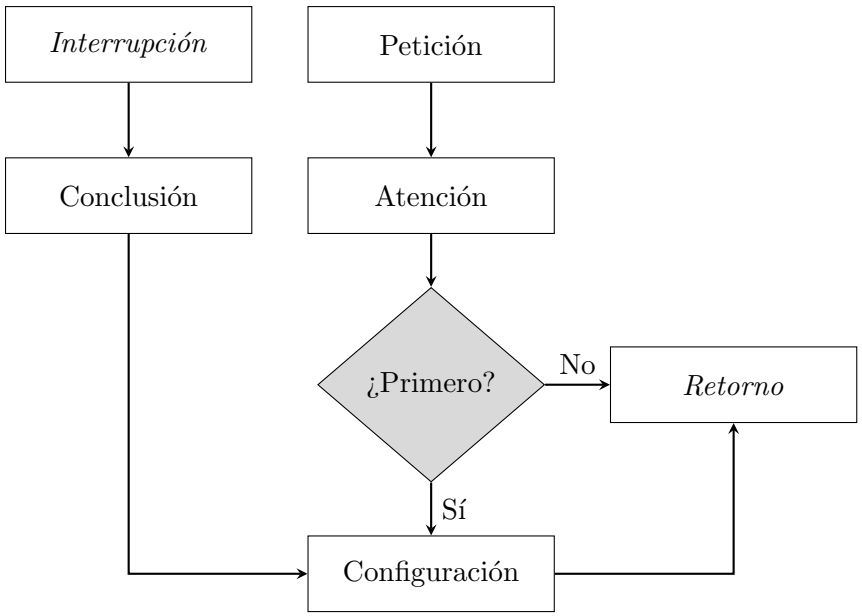


Figura 4: Diagrama de flujo de las etapas en las capas *Usuario-Kernel* y *Kernel-Dispositivo*. Las etapas que aparecen en cursiva son externas a estas capas.

En cuanto a la última capa, hay otros aspectos adicionales que tener en cuenta. Para una operación dada entre dos dispositivos, que requiera de sincronización entre ellos, siempre habrá uno que actúe como "emisor" (aquel que tiene la información y la desea enviar) y otro que actúe como "receptor" (aquel que desea recibir la nueva información), y por ello debemos distinguir entre dos listados de etapas distintos.

- Etapas del protocolo *Dispositivo-Dispositivo* (receptor) (diseño)

  1. **Configuración**: El driver escribe en los registros del dispositivo para configurar sus funcionalidades o para especificar los datos de entrada o salida (su posición en memoria y su longitud). Adicionalmente, también se especifica si el dispositivo realizará la operación como "emisor" o como "receptor". Cuando se realice la escritura en un registro especial, se dará la configuración por finalizada (no se permitirán más escrituras) y se procederá a la etapa 2.
  2. **Sincronizar**: El dispositivo espera a que el emisor pida la longitud disponible de los datos (el tamaño de la región que ha indicado el usuario para recibirlos). Una vez llega la petición, la trata adecuadamente (enviando la longitud al emisor) y continúa a la etapa 3.
  3. **Recepción**: El dispositivo recibe a través de su proxy toda o parte de los datos de usuario que el emisor envía, almacenándolos en un búfer interno. Luego, se pasa a la etapa 4.
  4. **Escritura**: El dispositivo escribe en memoria de usuario los datos almacenados en el búfer con una operación DMA que programa. Si no quedan más datos por recibir, se pasa a la etapa 5. En caso contrario, se vuelve a la etapa 3.
  5. **Conclusión**: El dispositivo genera una interrupción y finaliza la operación.

- Etapas del protocolo *Dispositivo-Dispositivo* (emisor) (diseño)

  1. **Configuración**: El driver escribe en los registros del dispositivo para configurar sus funcionalidades o para especificar los datos de entrada o salida (su posición en memoria y su longitud). Adicionalmente, también se especifica si el dispositivo realizará la operación como "emisor" o como "receptor". Cuando se realice la escritura en un registro especial, se dará la configuración por finalizada (no se permitirán más escrituras) y se procederá a la etapa 2.
     Observación: La lectura de un registro especial donde se almacene la longitud disponible provocará que el dispositivo genera una petición de sincronización de la misma al dispositivo receptor. De esta forma, el driver es capaz de determinar si el tamaño de los datos a emitir es correcto o no antes de empezar la operación.
  2. **Lectura**: El dispositivo lee de memoria de usuario toda o parte de los datos a transmitir con una operación DMA que programa, almacenándolos en un búfer interno. Luego, se pasa a la etapa 3.
  3. **Envío**: El dispositivo envía a través de su proxy los datos almacenados en el búfer. Si no quedan más datos por enviar, se pasa a la etapa 4. En caso contrario, se vuelve a la etapa 2.
  4. **Conclusión**: El dispositivo genera una interrupción y finaliza la operación.

Algunas de las etapas del protocolo en la capa *Dispositivo-Dispositivo* se comparten entre ambos modos del dispositivo, como la de configuración y conclusión, por lo que no hace falta diferenciarlas en el diseño. Esto es útil para la implementación, ya que no se requerirán más etapas de lo necesario.

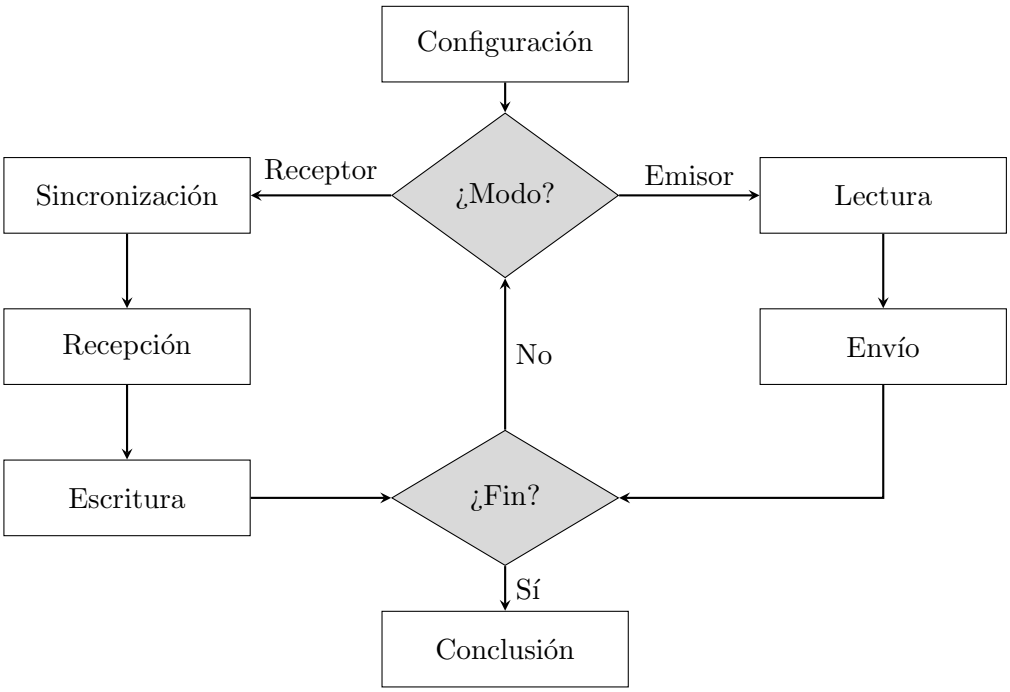


Figura 5: Diagrama de flujo de las etapas en la capa *Dispositivo-Dispositivo*.

10. Implementación

En esta sección se detallarán las estrategias utilizadas para implementar las decisiones de diseño de la arquitectura del sistema.

10.1. Ejemplo inicial

A la hora de implementar el diseño del proyecto, he utilizado el proyecto *pciemu*[24] como base en las etapas primerizas. *pciemu* es un ejemplo de emulación en QEMU de un dispositivo PCI Express, el cual implementa:

- Un **dispositivo virtual de QEMU**, que emula uno PCIe. El dispositivo registra una región de memoria personalizada y la asocia a su BAR (*Base Address Register*) 0, y la usa para definir operaciones de lectura y escritura sobre ella. Con estas operaciones se permite la configuración de registros especiales del dispositivo, representados como variables concretas en el código, para consultar o modificar su funcionamiento. Por otro lado, el dispositivo es capaz de leer o escribir un entero en uno de sus registros a través de operaciones DMA que configura, usando una dirección que almacena en otro registro y un búfer interno de 4 kB (externo a la región del BAR0) para operaciones por DMA. Tanto el dispositivo como las operaciones por DMA operan con direcciones de memoria de 64 bits (8 bytes).
- Un **driver para el kernel de Linux**, que se expone al usuario a través de un dispositivo lógico en el directorio `/dev/pciemu` del sistema de ficheros. Usando la llamada a sistema `ioctl()` (que cuenta con una función equivalente en la librería estándar) un usuario puede indicar que desea realizar una operación de lectura o escritura en el dispositivo. Para hacerlo, se indica el descriptor de fichero del dispositivo (debe haber sido abierto antes con `open()` para obtenerlo) y una dirección de memoria que la función permite como parámetro. El driver también se encarga de que la página donde se encuentra el valor en memoria quede "anclada"[4] (hacer *pin* de la página) y sea correctamente mapeada para su uso en operaciones de DMA[6]. Por otro lado, también aloja y configura una interrupción para el dispositivo, que se usa para "desanclar" (hacer *unpin* de la página) la página de memoria y para liberar el mapeo para DMA.
- Un **programa de usuario** sencillo que sirve como demostración del uso de la interfaz que implementa el driver mencionado anteriormente. El programa, además de mostrar la forma de abrir el dispositivo lógico del driver, realiza una serie de lecturas y escrituras de enteros en la memoria del dispositivo PCIe emulado para demostrar que es posible leer y escribir en sus registros internos a través del driver.

Habiendo investigado el proyecto *pciemu* en profundidad, estas partes son más que suficientes para servir como la base de mi proyecto. Sin embargo, va a ser necesario adaptar el código para que pueda soportar las funcionalidades esperadas y poder cumplir los objetivos y requisitos del proyecto.
Los principales cambios realizados al proyecto de ejemplo son los siguientes:

- Modificar la interfaz por `ioctl()` para que el usuario pueda especificar el tamaño de los datos además de su posición en memoria, si es que la operación lo requiere.

- Añadir el proxy al dispositivo PCIe, junto con sus registros de configuración y protocolo de comunicación.
- Dar soporte para la gestión de peticiones mediante colas (pendientes y finalizadas) en el driver.
- Permitir la posibilidad de trabajar con más de una página de memoria por operación (permitiendo transferencias de más datos).
- Expandir la memoria interna accesible del dispositivo para permitir transferencias de mayor tamaño.

10.2. Capa de Usuario

El código de usuario se encuentra en el directorio `src/sw/userspace`, y la cabecera principal (donde se encuentran las definiciones de los comandos disponibles mediante el uso de `ioctl()`) es `include/sw/module/psipe_ioctl.h`.
La comunicación entre las capas de usuario y kernel se implementa llamando a la función `ioctl()` sobre un descriptor de fichero obtenido al usar `open()` sobre el dispositivo lógico que corresponde al dispositivo, el cual es creado y gestionado por el driver (ver más adelante). Las operaciones definidas para los dispositivos disponibles desde los programas de usuario usando identificadores de comandos son las siguientes:

10.2.1. Operación de envío de datos: SEND

```
psipe_handle_t ioctl(int fd, PSIPE_IOCTL_SEND, struct psipe_data *data)
```

Descripción: El comando `SEND` indica al driver que se desea configurar el dispositivo (asociado a `fd`) para realizar una operación de envío a su pareja . Los datos a enviar se indican con la estructura `data`. La operación de envío se encolará al final de la cola de operaciones pendientes, y se moverá a la cola de operaciones finalizadas una vez concluya.
Valor de retorno: Un identificador de la operación encolada (mayor o igual a 0) en caso de éxito, o un valor menor a 0 en caso de error (consultar `errno` para más información).

10.2.2. Operación de recepción de datos: RECV

```
psipe_handle_t ioctl(int fd, PSIPE_IOCTL_RECV, struct psipe_data *data)
```

Descripción: El comando `RECV` indica al driver que se desea configurar el dispositivo (asociado a `fd`) para esperar una operación de envío desde su pareja. El espacio donde almacenar los datos a recibir se indica con la estructura `data`. La operación de recepción se encolará al final de la cola de operaciones pendientes, y se moverá a la cola de operaciones finalizadas una vez concluya.
Valor de retorno: Un identificador de la operación encolada (mayor o igual a 0) en caso de éxito, o un valor menor a 0 en caso de error (consultar `errno` para más información).

10.2.3. Operación de espera a petición: WAIT

```
long ioctl(int fd, PSIPE_IOCTL_WAIT, psipe_handle_t handle)
```

Descripción: El comando `WAIT` indica al driver que se desea esperar pasivamente a que una operación anterior, encolada en cualquiera de las colas del dispositivo asociado a `fd` y descrita por el identificador `handle`, finalice. La operación indicada es eliminada de la cola de peticiones finalizadas una vez concluye, por lo que el uso posterior del identificador de operación no dará resultado y se devolverá un error.
Valor de retorno: 0 en caso de éxito, o un valor menor a 0 en caso de error (consultar `errno` para más información).

10.2.4. Operación de limpieza de colas: FLUSH

```
long ioctl(int fd, PSIPE_IOCTL_FLUSH)
```

Descripción: El comando `FLUSH` indica al driver que se desea vaciar las colas de peticiones pendientes y finalizadas del dispositivo asociado a `fd`. Como también se vacía la cola de operaciones pendientes, se recomienda esperar a la última operación antes de usar este comando para asegurar que todas han finalizado.
Valor de retorno: 0 en caso de éxito, o un valor menor a 0 en caso de error (consultar `errno` para más información).

Las estructuras adicionales mencionadas en los comandos tienen el siguiente formato:

- La estructura `psipe_data` contiene dos campos: `addr` indica la dirección de memoria de los datos, mientras que `len` indica su tamaño en bytes. Ambos son de tipo `unsigned long`.
- El tipo `psipe_handle_t` es una re-definición de `unsigned long`, por lo que para la comprobación de errores, el resultado de las llamadas a `ioctl()` para los comandos `SEND` y `RECV` debe ser reinterpretado como `long` o `int`.

En los ficheros `psipe_wrappers.h` y `psipe_wrappers.c` (en el directorio `src/sw/userspace` del proyecto) se definen funciones para los programas de usuario que simplifican parte del proceso de trabajar con estos dispositivos.

10.3. Capa del Kernel

El código del driver se encuentra en el directorio `src/sw/kernel`, y la cabecera principal (donde se encuentran las definiciones de las estructuras en uso) es `psipe_module.h`.
Por su parte, el driver del kernel se encarga de gestionar las colas de operaciones y asegurarse de que los datos que el dispositivo lee o escribe tienen el formato correcto. También se debe encargar de configurar el dispositivo accediendo a sus registros una vez se deba programar la siguiente operación pendiente.
Para lograrlo se hace uso de dos listas doblemente encadenadas (`list_head`), de las librerías del kernel[22]. Las estructuras encoladas en ellas son `psipe_op`, que contiene toda la información necesaria para la gestión de una operación concreta. Sin embargo, como el kernel

de Linux es reentrante[21] (*preemptive*), y se hace uso de una interrupción para sincronizar el driver con el dispositivo, es necesario[5] que las estructuras de las listas estén protegidas con algún tipo de mecanismo de exclusión. En el caso del proyecto, la estructura utilizada son los *spinlocks*, ya que evitan que el kernel se "duerma" mientras espera a ser desbloqueado. Por otra parte, a cada nueva operación pendiente se le asigna un identificador único implementado como un contador[3], el cual puede ser usado para encontrar una operación encolada anteriormente.
En cuanto a la gestión de la memoria del usuario, resulta ahora necesario "anclar" todas las posibles páginas de una operación tal y como se encola, ya que no es seguro que cuando la interrupción se genere y se deba mapear las páginas de la nueva operación para DMA (este paso se puede dejar para este momento para evitar usar más memoria de lo necesario) estas estén mapeadas en memoria del proceso. Una vez se genera dicha interrupción por parte del dispositivo, el driver libera los mapeos para DMA de las páginas de la operación actual, así como deshace su estado "anclado", mueve la operación a la cola de operaciones finalizadas y prepara la (ahora primera de la cola) operación siguiente.
Tanto la lista de páginas "ancladas" como los mapeos para DMA de una operación concreta se guardan en su estructura `psipe_op`, estos últimos concretamente en una estructura `scatterlist`[23], que facilita su gestión.
Por otro lado, la interfaz con los programas de usuario se realiza a través de un dispositivo lógico que se encuentra en `/dev/psipe/dDbBdSfF_barX`, donde:

- `D` es el número de dominio del dispositivo PCI.
- `B` es el número de bus del dispositivo PCI.
- `S` es el número del "slot" del dispositivo PCI.
- `F` es el número de la función del dispositivo PCI.
- `X` es el número del BAR[4] del dispositivo PCI.

Estos ficheros son creados automáticamente por el driver, que hace uso de la librería `cdev` del kernel de Linux para crear un dispositivo de carácteres (*character device*) asociado a cada uno de los dispositivos. Esto ocurre cuando el driver configura un nuevo dispositivo que haya sido detectado por el kernel.

10.4. Capa del Dispositivo

El código del dispositivo se encuentra en el directorio `src/hw`, y la cabecera que utilizan el resto de capas (donde se encuentran las definiciones de las direcciones constantes de los registros en la memoria del dispositivo) es `include/hw/psipe_hw.h`.
La región de memoria registrada como BAR 0 contiene los siguientes registros, cuyos nombres van precedidos por el prefijo `PSIPE_HW_BAR0_`:

- `IRQ_0_RAISE`: La escritura en este registro provoca que el dispositivo genere la interrupción. Está disponible únicamente con el objetivo de depurar el dispositivo, pues ninguna operación hace uso de este registro.

[3] El contador es de tipo `unsigned long`, por lo que puede soportar 18446744073709551615 valores distintos antes de que haya *overflow*. He asumido que esta cantidad será suficiente para el proyecto.
[4] Para este proyecto, únicamente se usa el BAR 0.

- `IRQ_0_LOWER`: La escritura en este registro provoca que el dispositivo desactive la interrupción. El driver hace uso de este registro para dar el *acknowledge* de la interrupción.
- `DMA_CFG_LEN`: Contiene el tamaño restante (en bytes) por enviar/recibir de la operación actual.
- `DMA_CFG_PGS`: Contiene el número de mapeos para DMA que debe usar el dispositivo para acceder a memoria de usuario.
- `DMA_CFG_MOD`: El modo en que debe operar el dispositivo. Puede ser pasivo (receptor) o activo (emisor).
- `DMA_CFG_LEN_AVAIL`: El driver del receptor escribe en este registro el tamaño que espera recibir, y cuando el driver del emisor lee este registro, el dispositivo inicia una petición de sincronización a través del proxy con el dispositivo receptor.
- `DMA_DOORBELL_RING`: La escritura en este registro provoca que el dispositivo empiece a ejecutar la operación configurada. El resto de registros deben haber sido modificados de acorde con la operación deseada antes de escribir en este registro.
- `DMA_HANDLES`: Inicio del vector de *handles* de los mapeos para DMA del usuario, el cual puede almacenar hasta 131072 entradas. Es decir, en una sola operación el dispositivo es capaz de acceder hasta 512 MB de datos (suponiendo que el tamaño de página es de 4 kB) antes de que sea necesaria otra operación adicional. La cantidad de *handles* a utilizar es configurada por el driver a través del registro `DMA_CFG_PGS` (ver más arriba).

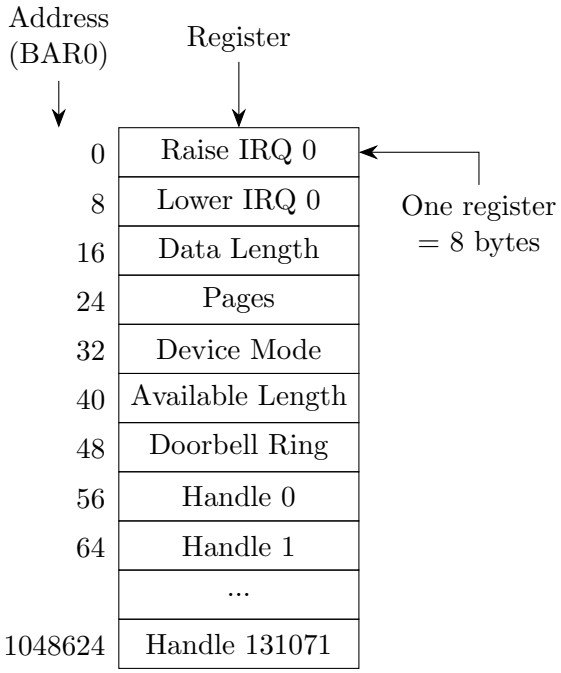


Figura 6: Diagrama de los registros del dispositivo en el BAR0.

En cuanto a la comunicación entre dispositivos, cada uno de ellos cuenta con
Las peticiones disponibles entre proxies se encuentran en el fichero `proxy.h`, en el directorio `src/hw`, y las más dos importantes son las siguientes, cuyos nombres van precedidos por el prefijo `PSIPE_REQ_`:

- `SLN` (*Send your available length*): Pide al proxy opuesto que responda por el socket con un `RLN`, seguido del contenido de su registro `DMA_CFG_LEN_AVAIL`.
- `RLN` (*Receive my available length*): Pide al proxy opuesto que lea del socket el nuevo valor para su registro `DMA_CFG_LEN_AVAIL`.

Estas dos peticiones son usadas durante la configuración de las operaciones en el dispositivo para asegurar que el tamaño de los datos que el emisor desea enviar al receptor es válido, es decir, que caben en la región que ha concretado el usuario.

10.5. Protocolos

Conociendo ahora los detalles de cada una de las capas del proyecto, podemos definir las etapas de los protocolos (ver sección 9.3) con más exactitud, es decir, revelando más detalles sobre la interacción entre los mecanismos usados en la implementación:

- Etapas de los protocolos *Usuario-Kernel* y *Kernel-Dispositivo* (implementación)
  1. **Petición**: El usuario inicia una petición al kernel a través de una llamada con `ioctl()` sobre el descriptor de fichero asociado al dispositivo.
     Observación: El descriptor de fichero asociado al dispositivo es obtenido en el programa con una llamada a `open()` sobre el fichero lógico del driver.
  2. **Atención**: El driver atiende la petición, comprueba que el tamaño de los datos del usuario es válido y "ancla" las páginas que corresponden en memoria. Luego, la operación se añade a la cola de peticiones pendientes del dispositivo. Si la cola estaba vacía, se continúa a la fase 3. En caso contrario, se retorna a modo usuario.
     Excepción: Si la operación es una de tipo `WAIT` o `FLUSH`, se resuelve inmediatamente sin necesidad de encolarla, volviendo a modo usuario al finalizar.
  3. **Configuración**: El driver consulta la primera operación pendiente y mapea para DMA las páginas que correspondan a sus datos, además de configurar los registros del BAR 0 del dispositivo de acorde con la operación y sus datos. Finalmente, se escribe en el registro `DMA_DOORBELL_RING` para que el dispositivo inicie la ejecución de la operación. El programa retorna a modo usuario hasta que se reciba la interrupción del dispositivo, momento en que se pasará a la etapa 4.
  4. **Conclusión**: Al recibir una interrupción del dispositivo, el driver "desanclará" las páginas de memoria de los datos que corresponden a la operación actual, y también liberará sus mapeos para DMA. Luego, moverá la operación actual (la primera de la cola de operaciones pendientes) al final de la cola de operaciones finalizadas. Luego, se pasa a la etapa 3.
- Etapas del protocolo *Dispositivo-Dispositivo* (receptor) (implementación)

1. **Configuración**: El driver escribe en los registros del dispositivo para configurar sus funcionalidades o para especificar los datos de entrada o salida (su posición en memoria y su longitud). Adicionalmente, también se especifica si el dispositivo realizará la operación como "emisor" o como "receptor" mediante el registro `DMA_CFG_MOD`. Cuando se realice la escritura en el registro `DMA_DOORBELL_RING`, se dará la configuración por finalizada (no se permitirán más escrituras) y se procederá a la etapa 2.
2. **Sincronizar**: El dispositivo espera una petición `SLN`, y una vez tratada continúa a la etapa 3.
3. **Recepción**: El dispositivo recibe a través de su proxy hasta una página de datos (4 kB) que el emisor envía, almacenándolos en su búfer para DMA. Luego, se pasa a la etapa 4.
4. **Escritura**: El dispositivo escribe en memoria de usuario los datos almacenados en el búfer con una operación DMA que programa. Si no quedan más datos por recibir, se pasa a la etapa 5. En caso contrario, se vuelve a la etapa 3.
5. **Conclusión**: El dispositivo genera la interrupción IRQ 0 y finaliza la operación, permitiendo volver a ser configurado.

- Etapas del protocolo *Dispositivo-Dispositivo* (emisor) (implementación)

1. **Configuración**: El driver escribe en los registros del dispositivo para configurar sus funcionalidades o para especificar los datos de entrada o salida (su posición en memoria y su longitud). Adicionalmente, también se especifica si el dispositivo realizará la operación como "emisor" o como "receptor" mediante el registro `DMA_CFG_MOD`. Cuando se realice la escritura en el registro `DMA_DOORBELL_RING`, se dará la configuración por finalizada (no se permitirán más escrituras) y se procederá a la etapa 2.
2. **Lectura**: El dispositivo lee de memoria de usuario hasta una página de datos (4 kB) con una operación DMA que programa, almacenándolos en el búfer para DMA. Luego, se pasa a la etapa 3.
3. **Envío**: El dispositivo envía a través de su proxy los datos almacenados en el búfer. Si no quedan más datos por enviar, se pasa a la etapa 4. En caso contrario, se vuelve a la etapa 2.
4. **Conclusión**: El dispositivo genera la interrupción IRQ 0 y finaliza la operación, permitiendo volver a ser configurado.

11. Pruebas

Por cada funcionalidad implementada, se han realizado pruebas con tal de comprobar su correcto funcionamiento. En caso de tener un comportamiento incorrecto o no esperado, diagnosticados a través de herramientas de depuración, se ha modificado la implementación hasta que los resultados sean los esperados.
Los diseños de los programas de prueba (todos ellos de usuario, con tal de usar el sistema completo) han ido aumentando en complejidad y ambición a medida que las funciones disponibles se han ido implementando y mejorando a través de las iteraciones del desarrollo, evolucionando con el proyecto. Sus códigos fuente se encuentran en el directorio `src/sw/userspace` del proyecto.

11.1. Pruebas progresivas

A continuación listo algunas de las pruebas realizadas sobre el código del proyecto, en orden cronológico. Dado que los mismos programas de usuario que he ido usando han ido expandiéndose y corrigiéndose a lo largo del proyecto, actualmente no hay un listado donde estén todos (de hecho, ninguno excepto la prueba final (más adelante) es funcional, estando desactualizadas), sin embargo siguen siendo accesibles a través del historial de los repositorios remotos[13][12][11] donde se almacena remotamente el código del proyecto.

- *Envío de un entero.* En el estado inicial del proyecto, el sistema sólo permitía la lectura/escritura de un único valor entero en los registros del dispositivo PCIe, el cual aún no contaba con ninguna funcionalidad que permitiera conectarse con otras VMs (es decir, la capa de "dispositivo" mencionada en la arquitectura del sistema no se había implementado aún). El objetivo de esta prueba era verificar que el dispositivo PCIe era reconocido correctamente por el kernel, configurado correctamente por el driver, y funcional a la hora de establecer la comunicación con el programa de usuario (es decir, que la escritura y lectura de registros del dispositivo era posible, además de poder programar operaciones por DMA y que su interrupción se tratase adecuadamente por el driver).

- *Envío entre VMs.* Al asegurar la transferencia correcta de un entero por DMA entre la memoria de usuario y la del dispositivo, se ha probado a enviarlo a través del proxy a la pareja del dispositivo. Esto ha requerido dos programas de usuario distintos, uno que programase una operación de recepción mientras que el otro programase una de envío complementaria.

- *Envío de un vector.* Una vez comprobado el correcto envío de un valor entero, el siguiente paso es enviar *varios en una única operación*, o un vector de ellos. Para ello, se ha tenido que tener en cuenta el sistema de gestión de memoria del kernel Linux, pues si los datos son suficientemente grandes pueden estar en dos o más páginas de memoria distintas, lo cual dificulta la lógica de las operaciones y requiere gestión adicional por parte del driver y el dispositivo. También se ha probado a que el programa de usuario receptor, una vez recibe los datos los modifica de una determinada forma, la cual es comprobada posteriormente por el emisor (los datos son enviados de vuelta) para verificar que no ha habido problemas.

11.2. Prueba final

La prueba final que se ha desarrollado es la adaptación del algoritmo de multiplicación de matrices en OpenMP, cuyo código fuente se encuentra en el fichero `matmul-offload-1o.c`, para su uso con el proyecto PSIPE. Para ello, se han tenido que traducir las directivas al compilador a operaciones de la capa de usuario del proyecto (definidas en la sección 10.2), además de realizar la distribución manual de los datos que OpenMP automatiza, para que sea compatible con el proyecto.
El resultado son dos programas, uno encargado de particionar las matrices para los dispositivos, indicarles qué parte les corresponde y construir los resultados de vuelta (`master-mm.c`), y otro encargado de recibir la información, calcular el resultado y devolverlo (`chiplet-mm.c`). Ambos programas hacen uso de funciones compartidas para gestionar los identificadores de dispositivo y otros wrappers para las llamadas a `ioctl()`, que se encuentran en los ficheros `psipe_wrappers.h` y `psipe_wrappers.c`.
En conjunto, los programas realizan la prueba siguiente:

1. El programa "master" aloja espacio e inicializa tres matrices distintas (A, B y C). Las matrices A y B contienen los datos de entrada, y son inicializadas con valores calculados por el "master", mientras que la matriz C es la que almacenará los resultados de la multiplicación de A y B, y por tanto se inicializa como una matriz nula. Mientras tanto, los programas "chiplet" esperan a recibir sus argumentos.

2. El "master" consulta el directorio `/dev/psipe` y abre todos los dispositivos que encuentra, almacenando sus descriptores de fichero en un vector interno.

3. Luego itera para cada uno de ellos, enviando los argumentos de entrada (que incluyen las dimensiones de las matrices y la cantidad de celdas que le corresponde procesar a un programa "chiplet" determinado), seguidos de las matrices A, B y las celdas de C que le correspondan[5].

4. El "master" espera a que (al menos) la última operación haya finalizado antes de limpiar las colas de operaciones y pasar al siguiente dispositivo.

5. Tras acabar de enviar y recibir los datos de los dispositivos, el "master" repite la multiplicación de A por B en otra matriz C_G distinta, el contenido de la cual es comparado con la matriz C para evaluar posibles errores de cálculo.

Si bien esta estrategia de multiplicación de matrices no es la más eficiente (hacer *blocking* podría ser mucho más efectivo en este caso, puesto que entonces no haría falta enviar la totalidad de las matrices A y B a cada dispositivo), considero que esta adaptación es suficiente para demostrar el funcionamiento correcto del proyecto.

11.2.1. Resultados de la prueba final

La tabla 7 muestra algunos tiempos de ejecución en segundos de multiplicaciones de matrices por *offload* en la prueba final, para diferentes cantidades de dispositivos y tamaños de matrices (en este caso se han utilizado matrices cuadradas para A, B y C, donde "N" es su

[5]Para determinar la cantidad de celdas a procesar para un programa "chiplet" concreto, el "master" calcula la división entera de las celdas de la matriz C entre el número total de dispositivos abiertos, repartiendo el resto (si lo hubiera) entre los dispositivos (empezando por el el que fue abierto primero).

orden, y por tanto su dimensión es N×N). Para estas pruebas se ha esperado a que cada operación finalizase antes de programar una nueva, lo cual se ha logrado dando un valor positivo a la constante `WAIT_ALL_OPS` definida en `psipe_wrappers.h`.

Dispositivos	N=**20**	N=**50**	N=**100**	N=**200**
2	0.873249	0.850725	0.857914	1.724158
4	1.732719	1.661537	1.687291	2.361655
8	3.418901	3.313935	3.372142	4.193653

Cuadro 7: Tiempos de ejecución (en segundos) de la prueba final, esperando a que cada operación termine antes de encolar una nueva.

12. Conclusiones

Tras haber desarrollado e implementado las distintas etapas del proyecto durante este TFG, concluyo que el resultado final al que he llegado es exitoso, pues ha cumplido con los objetivos y requisitos (ver sección 3.1) establecidos durante su planificación, a pesar de haber encontrado muchas dificultades (ver sección 8.2) durante el proceso.
En cuanto a mi experiencia personal, considero que el proyecto me ha ayudado a asentar conocimientos que he adquirido durante el grado, y que hacía tiempo que quería poner a prueba en un caso más serio. En particular, trabajar con el kernel de Linux y conocer más acerca de sus mecanismos y funcionamientos ha resultado ser muy importante -además de interesante- para lograr que el host y los chiplets logren completar un programa heterogéneo juntos.

12.1. Prospectiva de futuro

A partir de su estado actual, el proyecto puede ser expandido en los siguientes aspectos:

- Implementando funcionalidades más específicas de CXL, como *CXL.mem* o *CXL.cache*. En su estado actual, podemos considerar que el proyecto implementa *CXL.io*.
- Adaptando programas de prueba más computacionalmente costosos (o con muchas instancias de QEMU a la vez), con tal de evaluar el rendimiento de PSIPE en un entorno de mayor escala.
- Comparando la ejecución de aplicaciones en un entorno donde la arquitectura RISC-V esté siendo emulada con uno donde los aceleradores usen la arquitectura nativamente, para comprobar la pérdida de rendimiento esperada con el uso de QEMU al emularla.

13. Anexo A. Manual de instalación

En esta sección se detallarán los pasos para instalar y configurar el entorno de ejecución del proyecto. Se asume que se está usando un sistema tipo-UNIX (concretamente una distribución de Linux simiar a Debian o Ubuntu).
Las partes necesarias son:

- **Proto-SIPE**. El código del proyecto.
- **QEMU**. Incluido como un submódulo en el ćodigo del proyecto (ver más adelante).
- **Linux**. El kernel de Linux.
- **BusyBox**[2]. Utilidades básicas de UNIX para su uso en la VM. Incluidas en la imagen del disco que proporciono (ver más adelante).

13.1. Dependencias adicionales

Los paquetes necesarios para la instalación del proyecto (y el comando para instalarlos) son los siguientes:

```
sudo apt install git gcc make opensbi u-boot-qemu libglib2.0-dev \
    libfdt-dev libpixman-1-dev zlib1g-dev ninja-build netcat \
    libssh-dev libvde-dev libvdeplug-dev libcap-ng-dev \
    libattr1-dev libslirp-dev gcc-riscv64-linux-gnu
```

Junto con estos paquetes se instalarán también el resto de dependencias (como las binutils de RISC-V, que pueden ser útiles para depurar el código, si se desea).

13.2. QEMU

En primer lugar se debe clonar el repositorio del proyecto des de GitHub. QEMU está incluido como un submódulo de git. En este paso también se compilará el código del dispositivo de QEMU, que se incluye dentro de la estructura de directorios del mismo al ejecutar `setup.sh`.
El resto del proyecto será compilado una vez se haya instalado el kernel de Linux.

```
git clone https://github.com/Dorovich/psipe.git
cd psipe
./setup.sh
make -j $(nproc)
```

13.3. Kernel de Linux

Ahora compilaremos el kernel de Linux con su configuración por defecto. Si bien muchos de los módulos opcionales del kernel se pueden deshabilitar para este paso (pues en el proyecto no se usarán), los dejaremos para evitar añadir complejidad adicional a la instalación - además de que con la configuración por defecto, el tiempo de compilación no es muy elevado.
En el directorio `psipe` (clonado), ejecutar:

```
wget https://cdn.kernel.org/pub/linux/kernel/v6.x/linux-6.6.72.tar.xz
tar xz linux-6.6.72.tar.xz
cd linux-6.6.72
ARCH=riscv CROSS_COMPILE=riscv64-linux-gnu- make defconfig
ARCH=riscv CROSS_COMPILE=riscv64-linux-gnu- make -j $(nproc)
```

13.4. Driver y programas de usuario

Ahora que tenemos disponible el código del kernel, podemos proceder con la compilación del driver del proyecto. También aprovechamos para compilar los programas de usuario de muestra.
En el directorio `psipe/src/sw/kernel`:

```
make
```

Y en el directorio `psipe/src/sw/userspace`:

```
make
```

13.5. Máquina virtual

Dirigirse al directorio `psipe/vm`.
Todas las instancias de QEMU ejecutadas como muestra en el proyecto (ver Manual de uso) utilizan como disco la imagen `vda.img`, la cual pueden compartir al indicar que su uso es únicamente de lectura. En esta imagen ya he instalado de antemano BusyBox 1.36.0, junto con un sistema de ficheros ext2, por lo que es usable sin ninguna modificación adicional.
Para actualizar el driver y los programas de muestra que hay en el sistema de ficheros de la imagen, con los que hemos compilado en el paso anterior, ejecutar el siguiente comando en el directorio (requerirá permisos elevados):

```
./vm.sh -u
```

El programa `vm` (con la flag `-u`, de "update") que proporciono debe ser ejecutado tras cualquier modificación al código del driver o programa de usuario, con tal de que las copias en la imagen de disco estén actualizadas con los últimos cambios.

14. Anexo B. Manual de uso

Una vez el programa y todas sus dependencias han sido instaladas y configuradas correctamente (ver Manual de instalación), es posible ejecutar el sistema completo.

14.1. Línea de comandos

El dispositivo se puede añadir a una instancia de QEMU a través una opción adicional en sus parámetros de ejecución usuales[9]:

```
-device psipe[,server_mode=true|false[,port=port]]
```

- `server_mode`: Si el proxy del dispositivo debe actuar como servidor o como cliente. Para cada pareja de dispositivos, uno debe estar en modo servidor y el otro en modo cliente. Esta opción es independiente de qué instancia ejecute el programa de "master". Por defecto, esta opción es `true`.
- `port`: El puerto que el proxy utilizará para establecer una conexión con su pareja, ya sea para iniciar el servidor escuchando en ese puerto (en caso de que `server_mode` sea `true`), o bien para conectarse a través de él (en caso de que `server_mode` sea `false`). Por defecto, el puerto usado es 8987.

14.2. Programa `vm.sh`

Este programa se encuentra en el directorio `vm` del proyecto, junto con los ficheros de gestión de la máquina virtual (`manage-disk.sh`) y la imagen de disco de muestra (`vda.img`). Su principal función es la de ofrecer una forma sencilla de poner en ejecución una instancia de QEMU con uno o más dispositivos *Proto-SIPE* listos para ser conectados.
Las *flags* que soporta el programa son las siguientes:

- `-D`: Inicia la instancia de QEMU detenida, lista para ser controlada desde el depurador GDB a través del puerto 1234.
- `-s`: Configurar todos los dispositivos de la instancia como "servidores". Posteriormente, se debe iniciar otras instancias con disposividos en modo "cliente" (es decir, sin especificar esta opción).
- `-p base`: Configurar todos los dipositivos con puertos en orden ascendente, empezando por `base`. Por ejemplo, si se inicia una instancia con 4 dispositivos y se especifica la opción `-p 9990`, los dispositivos usarán los puertos `9990`, `9991`, `9992` y `9993`. `base` debe ser un entero mayor a 1024.
- `-n number`: Iniciar la instancia con `number` dispositivos conectados. Sus opciones pueden ser modificadas por otras *flags*. `number` debe ser un entero igual o mayor a 0.
- `-m`: Utilizar el monitor de QEMU en el terminal desde el que se ha creado la instancia. Por defecto se muestra la salida estándar de QEMU, pero con esta opción se puede cambiar.
- `-M`: Iniciar la instancia en modo "mantenimiento". Esta opción permite escribir en la imagen de disco desde la VM.

- `-u`: Actualiza la imagen de disco de la VM con el programa del driver PSIPE y los programas de usuario especificados en el script.

Referencias

[1] Broadcom. *VMWare*. URL: https://www.vmware.com/. (Último acceso: 10/3/2025).

[2] *BusyBox*. URL: https://busybox.net/. (Último acceso: 13/6/2025).

[3] *Digital Autonomy with RISC-V in Europe (DARE)*. URL: https://www.bsc.es/join-us/excellence-career-opportunities/dare. (Último acceso: 11/3/2025).

[4] The Linux Kernel docuementation. $pin_user_pages()and related calls$. URL: https://docs.kernel.org/core-api/pin_user_pages.html. (Último acceso: 16/6/2025).

[5] The Linux Kernel docuementation. *Proper Locking Under a Preemptible Kernel: Keeping Kernel Code Preempt-Safe*. URL: https://docs.kernel.org/locking/preempt-locking.html. (Último acceso: 17/6/2025).

[6] *Dynamic DMA mapping Guide*. URL: https://docs.kernel.org/core-api/dma-api-howto.html. (Último acceso: 16/6/2025).

[7] *Gantt Project*. URL: https://www.ganttproject.biz/. (Último acceso: 10/3/2025).

[8] *Inteligencia artificial*. URL: https://es.wikipedia.org/wiki/Inteligencia_artificial. (Último acceso: 22/6/2025).

[9] *Invocation - QEMU Documentation*. URL: https://qemu-project.gitlab.io/qemu/system/invocation.html. (Último acceso: 14/6/2025).

[10] Kevin Lawton. *Bochs*. URL: https://bochs.sourceforge.io/. (Último acceso: 10/3/2025).

[11] David Cañadas López. *pciemu (fork) - GitHub*. URL: https://github.com/Dorovich/pciemu. (Último acceso: 19/6/2025).

[12] David Cañadas López. *Proto-NVLink - GitLab BSC*. URL: https://gitlab.bsc.es/dcanadas/proto-nvlink. (Último acceso: 19/6/2025).

[13] David Cañadas López. *Proto-SIPE - GitHub*. URL: https://github.com/Dorovich/psipe. (Último acceso: 19/6/2025).

[14] Bram Moolenaar. *Vim*. URL: https://www.vim.org/. (Último acceso: 10/3/2025).

[15] Oracle. *VirtualBox*. URL: https://www.virtualbox.org/. (Último acceso: 10/3/2025).

[16] *Overleaf*. URL: https://www.overleaf.com/. (Último acceso: 11/3/2025).

[17] *Precioluzhora*. URL: https://tarifaluzhora.es/info/precio-kwh. (Último acceso: 10/3/2025).

[18] GNU Project. *GNU Compiler Collection*. URL: https://gcc.gnu.org/. (Último acceso: 10/3/2025).

[19] GNU Project. *GNU Debugger*. URL: http://gnu.org/software/gdb. (Último acceso: 10/3/2025).

[20] *QEMU*. URL: https://www.qemu.org/. (Último acceso: 10/3/2025).

[21] *Reentrancia (informática) - Wikipedia*. URL: https://es.wikipedia.org/wiki/Reentrancia_(inform%C3%A1tica). (Último acceso: 17/6/2025).

[22] Elixir Bootlin Cross Referencer. *list.h - Linux source code v6.6.72*. URL: https://elixir.bootlin.com/linux/v6.6.72/source/include/linux/list.h. (Último acceso: 17/6/2025).

[23] Elixir Bootlin Cross Referencer. *scatterlist.h - Linux source code v6.6.72*. URL: `https://elixir.bootlin.com/linux/v6.6.72/source/include/linux/scatterlist.h`. (Último acceso: 17/6/2025).

[24] Luiz Suraty. *pciemu*. URL: `https://github.com/luizinhosuraty/pciemu`. (Último acceso: 10/3/2025).

[25] Linus Torvalds. *Git*. URL: `https://git-scm.com/`. (Último acceso: 10/3/2025).