UC | Facultad de Ciencias

Demostrador de Caso de Uso de Aplicación de Conducción Autónoma – Particionado Temporal en Linux para Plataformas con GPUs

(Autonomous Driving Application Use Case – Time Partitioning in Linux for Platforms with GPUs)

Trabajo de Fin de Máster
para acceder al

MÁSTER EN INGENIERÍA INFORMÁTICA

Autor: Borja Cuevas Cuesta

Director\es: J. Javier Gutiérrez García y Iosu Gómez Iturrioz

Septiembre - 2024

Índice general

Índice de figuras

Resumen

El creciente interés de los últimos años en la conducción autónoma está llevando al desarrollo de aplicaciones cada vez más complejas que requieren mayor potencia computacional, impulsando consigo la evolución de las plataformas sobre las que se ejecutan hacia arquitecturas heterogéneas con procesadores y aceleradores específicos como GPUs y NPUs.

Las aplicaciones de conducción autónoma no solo requieren altas prestaciones de cómputo, sino que deben cumplir también unos requisitos de seguridad funcional y tiempo real estrictos, ya que cualquier retraso o fallo en algunas de sus tareas podría originar situaciones de riesgo para las personas cercanas al sistema (pasajeros, peatones, etc.). Para garantizar que estas tareas críticas cumplan sus respectivos plazos temporales, es esencial desarrollar el sistema de tal forma que se comporte de una manera predecible, permitiendo anticipar su comportamiento para poder asegurar una respuesta correcta ante un determinado estímulo dentro del tiempo acotado.

Sin embargo, la predictibilidad de los sistemas de tiempo real puede verse afectada en plataformas heterogéneas con GPUs integradas debido al uso intensivo de memoria y los mecanismos de planificación interna desconocidos, lo cual genera tiempos de respuesta impredecibles. Por tanto, ante la falta de soluciones que garanticen la aplicación de técnicas de tiempo real y seguridad funcional en este tipo de plataformas, en este trabajo se busca desarrollar un entorno de pruebas que facilite la experimentación en dicho campo.

Este entorno de pruebas, denominado Planificador mediante Ventanas Temporales sobre Linux (PVTL), emula un particionado temporal dividiendo el sistema en diferentes particiones garantizando el aislamiento entre ellas y proporcionando la capacidad de controlar la activación de las tareas del sistema para evitar accesos concurrentes a la GPU, mitigando así el efecto de la interferencia de memoria. Para validar su comportamiento, se integra un caso de uso de conducción autónoma en el que se ejecuta un algoritmo de detección de obstáculos dentro de este entorno de pruebas, permitiendo analizar el efecto de la configuración del particionado sobre sus tiempos de respuesta.

Palabras clave: Plataformas Heterogéneas, GPUs, Particionado, Hipervisor, Sistemas Críti-

cos, Tiempo Real.

Abstract

The growing interest in autonomous driving in recent years has led to the development of increasingly complex applications that require greater computational capabilities, driving the evolution of the platforms on which they run toward heterogeneous architectures with specialized processors and accelerators such as GPUs and NPUs.

Autonomous driving applications not only require high computational performance but must also meet strict functional safety and real-time requirements, as any delay or failure in some of their tasks could lead to risky situations for people near the system (such as assengers or pedestrians). To ensure that these critical tasks meet their respective deadlines, it is essential to develop the system in a way that ensures predictable behavior, allowing anticipation of its performance to guarantee a correct response to a given stimulus within a specified time.

However, the predictability of real-time systems can be compromised on heterogeneous platforms with integrated GPUs due to intensive memory usage and unknown internal scheduling mechanisms, which lead to unpredictable response times. Therefore, given the lack of solutions that guarantee the application of real-time and functional safety techniques on these platforms, this work aims to develop a testbed to facilitate experimentation in this field.

This testbed, called Time-Window Scheduler on Linux (PVTL, by its acronym in Spanish), emulates temporal partitioning by dividing the system into different partitions, ensuring isolation between them and providing the ability to control task activation to avoid concurrent GPU access, thus mitigating memory interference. To validate its behavior, an autonomous driving use case is integrated, where an obstacle detection algorithm is executed within this test environment, allowing the analysis of the effect of the partitioning configuration on response times.

Keywords: Heterogeneous Platforms, GPUs, Partitioning, Hypervisor, Critical Systems, Real-Time.

Capítulo 1

Introducción

En este primer capítulo introductorio, se proporcionará un contexto para el desarrollo de este trabajo que ayude a comprender las razones que han motivado su realización. Para ello, se expondrán los principales factores que han impulsado la exploración e investigación de este tema en particular. A continuación, se enumerarán los objetivos específicos del trabajo, indicando las metas que se pretenden alcanzar. Además, se describirá el proceso de realización del proyecto, explicando brevemente las fases seguidas. Por último, se resumirá la estructura del documento enumerando los capítulos por los que está formado y explicando brevemente el contenido de cada uno de ellos.

1.1. Contexto y motivación

La conducción autónoma se ha convertido en los últimos años en un sector que cada vez despierta un mayor interés en la industria, no sólo en el mundo de la automoción, sino también en el transporte de personas y mercancías por carretera o ferrocarril y en entornos de fabricación más controlados. Las aplicaciones desarrolladas en estos sectores son cada vez más complejas, y el aumento de la carga computacional que demandan ha provocado la evolución de las plataformas sobre las que se ejecutan hacia arquitecturas heterogéneas con varios procesadores y aceleradores hardware específicos, como GPUs (Graphics Processing Units) o NPUs (Neural Processing Units) [3] [4].

Por otro lado, además de unas altas prestaciones de cómputo, estas aplicaciones de conducción autónoma también exigen satisfacer ciertos requisitos de seguridad funcional y de tiempo real. Por lo tanto, no sólo será fundamental que los resultados del cómputo sean correctos, sino que también será de suma importancia que estos resultados se obtengan dentro

de unos plazos temporales impuestos. Por su impacto directo sobre la seguridad en la vida de las personas, las aplicaciones de conducción autónoma entran dentro de lo que se categoriza como sistemas de tiempo real críticos, en los que las respuestas del sistema se deben producir obligatoriamente dentro del plazo especificado. En caso contrario, se podrían originar situaciones en las que se pusiese en peligro la vida de las personas cercanas al sistema (pasajeros, peatones, etc.) y que pudiesen tener consecuencias catastróficas [5].

Una manera muy habitual de garantizar que las tareas críticas de tiempo real cumplan sus respectivos plazos temporales es desarrollar el sistema para que se comporte de una manera predecible. De esta forma, puede conocerse de antemano cuál será su comportamiento al ejecutar un determinado conjunto de aplicaciones, permitiendo comprobar si se va a poder asegurar una respuesta correcta ante un estímulo concreto dentro del tiempo acotado.

Sin embargo, la predictibilidad de un sistema de tiempo real puede verse comprometida cuando la plataforma hardware utilizada implementa una arquitectura heterogénea con una GPU integrada. En estos casos, el simple hecho de utilizar la GPU puede provocar interferencias debido a un uso intensivo de la memoria, degradando el comportamiento temporal del sistema [6]. Además, el acceso concurrente a la GPU entre diferentes aplicaciones que pueda darse dentro del sistema hace que aumente la impredictibilidad de los tiempos de respuesta de las tareas ejecutadas en la GPU debido a los mecanismos internos de planificación [7], los cuales son desconocidos.

Por estas razones, y ante la falta de soluciones disponibles actualmente en el mercado que permitan aplicar técnicas de tiempo real y seguridad funcional sobre plataformas heterogéneas con una GPU integrada, en este trabajo se pretende elaborar un entorno de ejecución que funcione como un campo de pruebas para la experimentación en la ejecución de tareas críticas de tiempo real sobre plataformas con este tipo de arquitecturas.

1.2. Objetivos del trabajo y fases de realización

El objetivo principal de este trabajo consiste en el desarrollo e implementación de un entorno de pruebas que permita ejecutar de manera predecible aplicaciones críticas con requisitos de tiempo real sobre una plataforma heterogénea con una GPU integrada, empleando para ello un particionado temporal. Este particionado va a permitir controlar, por un lado, los instantes de tiempo en los que se van a servir las activaciones de las tareas periódicas y, por otro lado, los periodos de tiempo durante los que se va a ejecutar cada tarea. Por tanto, mediante este mecanismo se proporcionará la posibilidad de adaptar la configuración de la ejecución de un conjunto de aplicaciones para mejorar la predictibilidad del sistema.

De esta manera, será posible tener control tanto sobre la interferencia de memoria como sobre

el acceso a la GPU, pudiendo evitar el acceso concurrente entre diferentes aplicaciones.

Posteriormente, se desarrollará un caso de uso de una aplicación de conducción autónoma basada en un algoritmo de detección de obstáculos que será integrada en este entorno de pruebas y ejecutada dentro del particionado diseñado. Esta ejecución servirá como demostrador para validar el correcto comportamiento del entorno desarrollado.

Para alcanzar estos dos objetivos principales, los retos específicos que se han planteado durante cada una de las fases de realización del proyecto han sido las siguientes:

- Estudio del estado del arte de los hipervisores disponibles para plataformas con GPUs con el propósito de analizar las características de cada uno de cara a su implementación sobre el entorno de pruebas de aplicaciones de tiempo real.
- Implementación de una capa de software sobre la plataforma de ejecución para la planificación de las tareas mediante el particionado temporal mencionado previamente.
- Definición e implementación de un caso de uso de una aplicación de conducción autónoma.
- Integración del caso de uso sobre la plataforma heterogénea, ejecutándose sobre el particionado temporal.
- Validación de la integración y comprobación del correcto funcionamiento del entorno desarrollado.

1.3. Descripción de la estructura del documento

Para proporcionar una visión clara del contenido desarrollado durante el proyecto, se proporciona a continuación un resumen de los 7 capítulos en los que se ha estructurado este documento:

- Capítulo 1: capítulo introductorio en el que se presenta la temática alrededor de la cual giran los objetivos del trabajo y se describe la motivación del trabajo.
- Capítulo 2: se explican conocimientos previos, tanto teóricos como relativos a la plataforma hardware utilizada, básicos para facilitar la comprensión del desarrollo del trabajo.

- Capítulo 3: se proporciona un estudio del estado del arte de los hipervisores orientados a sistemas de tiempo real compatibles con plataformas hardware heterogéneas con una GPU integrada como la que se utiliza en este trabajo.

- Capítulo 4: se describen las conclusiones extraídas tras el estudio del estado del arte del capítulo anterior, las cuales motivan el planteamiento de una nueva propuesta para una planificación que permita emular el comportamiento del particionado temporal sobre un Linux de tiempo real. Seguidamente, se detalla a nivel teórico el funcionamiento de dicha propuesta y se explican los detalles de implementación y las pruebas de validación realizadas.

- Capítulo 5: se describen, por un lado, las diferentes aplicaciones que forman el caso de uso implementado en el demostrador, detallando las adaptaciones necesarias en cada uno de los casos y, por otro lado, el proceso de integración del propio caso de uso en el entorno de pruebas desarrollado.

- Capítulo 6: en primer lugar, se explica cómo se han diseñado las pruebas experimentales para verificar el correcto funcionamiento del entorno de ejecución. En segundo lugar, se presentan y analizan los resultados obtenidos.

- Capítulo 7: se enumeran los logros que se han alcanzado tras el desarrollo del proyecto y se proponen una serie de líneas de trabajo abiertas para seguir investigando en este campo y profundizando en algunos puntos que se han quedado fuera del alcance de este trabajo.

Capítulo 2

Conocimientos previos

En este segundo capítulo, se comenzará explicando una serie de conceptos previos necesarios para comprender el desarrollo del trabajo. A continuación, se describirá detalladamente la plataforma hardware sobre la que se van a ejecutar las aplicaciones desarrolladas y, finalmente, se revisarán otros trabajos previos relacionados que permitan situar este trabajo de investigación en el marco del conocimiento existente.

2.1. Hipervisores y tipos

Un hipervisor es un software diseñado para gestionar la ejecución de múltiples entornos de computación (llamados máquinas virtuales o particiones) en un sólo dispositivo físico. Cada partición opera con su propio sistema operativo o bien directamente con aplicaciones que se ejecutan en modo *bare-metal*. El hipervisor es el encargado de garantizar el aislamiento entre particiones para que puedan ejecutarse de manera independiente y segura mientras gestiona y asigna los recursos físicos subyacentes (como las CPUs o los diferentes niveles de memoria) a las distintas particiones en función de las necesidades de cada una.

Existen dos tipos de hipervisores en función de la capa de abstracción sobre la que se ejecuten:

- Hipervisores de tipo 1: se implementan directamente sobre el hardware físico, sin la intermediación de un sistema operativo anfitrión (ver Figura 2.1). Por este motivo, también se denominan hipervisores *bare-metal*. En estos casos, el hipervisor actúa como un sistema operativo ligero, gestionando y asignando directamente los recursos hardware a las particiones.

  El hecho de prescindir de un sistema operativo anfitrión mejora la eficiencia ya que

permite administrar los recursos sin necesidad de pasar por capas software intermedias. Además de mejorar el rendimiento, también incrementa la seguridad, ya que se reducen los riesgos de inestabilidad debido a la ausencia de un sistema operativo anfitrión.

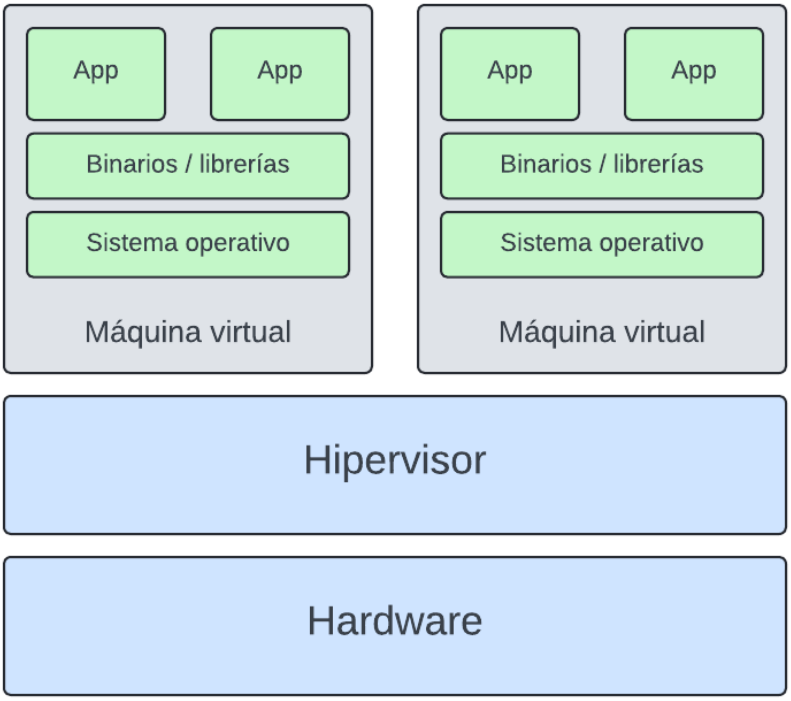

Figura 2.1: Arquitectura de un hipervisor de tipo 1.

- Hipervisores de tipo 2: se implementan sobre un sistema operativo anfitrión (ver Figura 2.2) y son conocidos como hipervisores *hosted*. A diferencia de los de tipo 1, estos hipervisores no controlan directamente los recursos hardware; en su lugar, dependen del sistema operativo anfitrión para la asignación de dichos recursos, que luego son asignados a las máquinas virtuales.

  La dependencia de un sistema operativo intermedio introduce una mayor latencia en el entorno virtualizado, ya que el hipervisor debe solicitar el acceso a los recursos hardware a través del sistema operativo. Además, la estabilidad y el rendimiento de las máquinas virtuales dependen en gran medida de la estabilidad del sistema operativo anfitrión.

A pesar de sus diferencias, ambos tipos de hipervisores son útiles en función del contexto en el que se utilicen. Los hipervisores de tipo 1 son preferidos en centros de datos empresariales y entornos de nube por su alta eficiencia, escalabilidad y seguridad. Además, son fundamentales en la industria para garantizar la seguridad mediante mecanismos de aislamiento entre

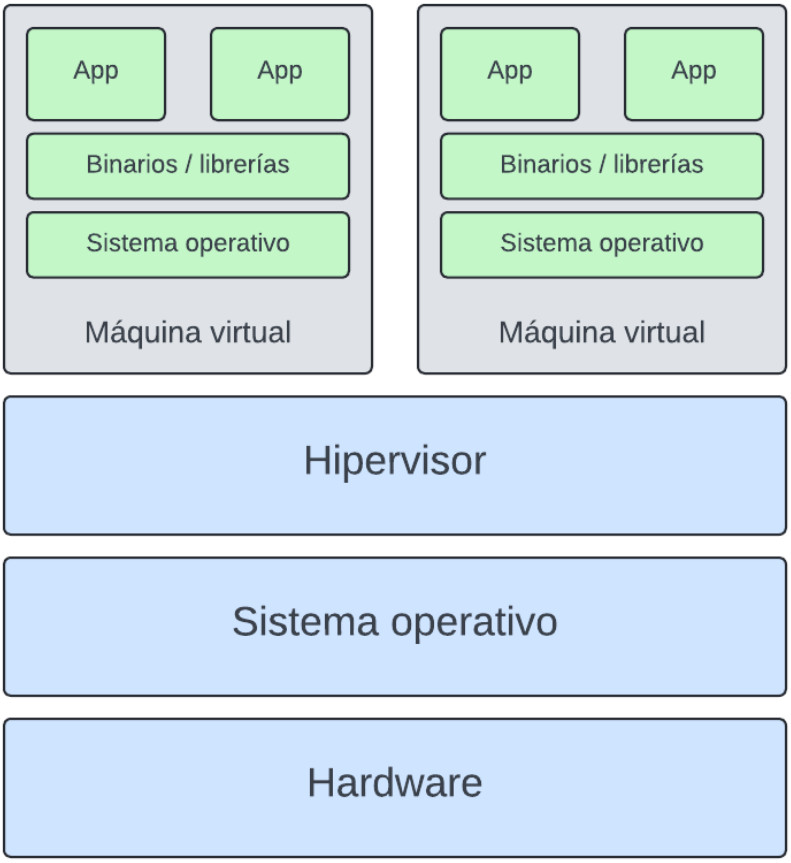

Figura 2.2: Arquitectura de un hipervisor de tipo 2.

particiones en sistemas críticos como la aviónica, ferrocarril, automoción, etc. En contraste, los hipervisores de tipo 2 son valorados por su facilidad de instalación, configuración y uso, lo que los convierte en una opción popular para entornos de desarrollo y aplicaciones de escritorio.

2.2. Estándar ARINC 653 y planificación *Time-Table-Driven*

ARINC 653 [8] es un estándar desarrollado por *Airlines Electronic Engineering Committee* (AEEC). Este estándar define una especificación para el desarrollo de aplicaciones software para aviónica con requisitos estrictos de seguridad y de tiempo real que deben funcionar correctamente incluso en presencia de fallos o condiciones anómalas.

Por lo tanto, una aplicación cuya seguridad sea crítica conforme a ARINC 653 es aquella que sigue las directrices y requisitos establecidos en su documento de especificaciones. Este documento define un entorno de sistema operativo particionado para la ejecución de múltiples aplicaciones software (denominadas particiones) en una única plataforma de hardware. Cada partición está aislada del resto, garantizando que cualquier fallo o problema que ocurra en una partición no afectará al funcionamiento de las demás.

Algunos de los puntos clave definidos en el estándar ARINC 653 son los siguientes:

- Particionado: cada partición opera como una entidad distinta, encapsulando una aplicación software concreta. El aislamiento entre particiones impide que se produzcan interferencias entre sus ejecuciones, garantizando la integridad de los sistemas críticos.

  El particionado se puede llevar a cabo en tiempo, en espacio o en ambas, asignando franjas temporales (tiempo) y regiones de memoria (espacio) específicas a cada partición. Este mecanismo contribuye a acotar los tiempos de respuesta de cada tarea y, por lo tanto, a mejorar la predictibilidad del sistema.

- Mecanismos de comunicación: además del aislamiento, la comunicación entre particiones es igualmente crítica en sistemas complejos. Este estándar define mecanismos de comunicación que permiten un intercambio de datos controlado y determinista entre particiones, manteniendo estrictas restricciones de temporización.

- Mecanismos de monitorización y manejo de errores: ARINC 653 incorpora mecanismos que ayudan al sistema a detectar anomalías en las particiones y a responder a los errores de manera rápida, favoreciendo una recuperación ágil.

Para garantizar el aislamiento temporal entre particiones, el estándar ARINC 653 define una política de planificación *time-table driven*, también conocida como planificación estática. Este tipo de planificación consiste en crear una tabla predefinida que especifica las ventanas temporales en las que se ejecutará cada partición y que están caracterizadas por su instante de comienzo y su duración. El planificador sigue esta tabla para determinar qué partición debe ejecutarse en cada momento.

La principal ventaja de este tipo de planificación es que, al tratarse de una tabla estática y predefinida, se consigue una ejecución determinista del sistema al conocer a priori los tiempos configurados para cada ventana. De esta manera, limitando mediante ventanas temporales el tiempo máximo durante el cual se va a ejecutar cada aplicación, se puede conseguir un mayor control sobre los periodos de tiempo durante los que se ejecutará cada tarea y sobre los tiempos de respuesta de cada una.

2.3. Descripción de la plataforma hardware utilizada: F1-Tenth

En el desarrollo de sistemas dentro del contexto de la conducción autónoma, el entorno hardware utilizado desempeña un papel fundamental para asegurar la correcta ejecución y validación de los algoritmos implementados. En este trabajo, se ha escogido el prototipo de vehículo de conducción autónoma F1Tenth como plataforma de ejecución (ver Figura 2.3).

Figura 2.3: Imagen del prototipo F1Tenth utilizado en el proyecto [1].

El F1Tenth [1] es un modelo a escala 1:10 de un automóvil de carreras, ampliamente utilizado en investigación dentro del mundo de la robótica y los sistemas autónomos. Se trata de una plataforma diseñada específicamente para experimentar y desarrollar algoritmos de conducción autónoma en un entorno controlado. Está equipado con diversos sensores y cámaras, así como una unidad de procesamiento de alto rendimiento, permitiendo de esta manera la ejecución de algoritmos complejos de manera eficiente y proporcionando un entorno adecuado para llevar a cabo pruebas de validación de aplicaciones de conducción autónoma.

Además, cuenta también con otros elementos como una placa de alimentación para gestionar

y distribuir con eficiencia la potencia suministrada, un adaptador Wi-Fi para las comunicaciones y transferencias de datos, una unidad *OBU-301U V2X On-Board Unit* para la comunicación inálambrica en tiempo real con todos los elementos de comunicación posibles del entorno (otros vehículos, peatones, ciclistas e infraestructuras inteligentes) y, por último, un Controlador Electrónico de Velocidad (VESC) responsable de controlar el motor del vehículo y regular la velocidad.

Para el desarrollo de este trabajo, los componentes hardware que se han utilizado han sido la unidad de procesamiento (concretamente, una NVIDIA Jetson Orin NX) y la cámara de profundidad (ambos detallados en los subapartados 2.3.1 y 2.3.2).

La cámara RGB-D se ha utilizado para la percepción del entorno a la hora de ejecutar el caso de uso de conducción autónoma. Este tipo de aplicaciones trabaja con una gran cantidad de puntos cuyo procesamiento supone una alta carga computacional, lo cual motiva la utilización de una unidad de procesamiento con una GPU integrada, como es el caso de la la Jetson Orin NX.

2.3.1. Unidad de procesamiento: Jetson Orin NX

Como unidad de procesamiento se ha integrado en el prototipo F1Tenth el dispositivo Jetson Orin NX, desarrollado por NVIDIA. Se trata de una plataforma de computación de alto rendimiento diseñada específicamente para aplicaciones de inteligencia artificial y robótica, lo cual le convierte en una buena elección capaz de llevar a cabo las complejas tareas de percepción, planificación y control maximizando el rendimiento y optimizando el uso de los recursos.

Esta unidad se distingue por su capacidad para realizar cálculos intensivos de manera eficiente. Sus principales características se resumen en la tabla del Cuadro 2.1.

Por otro lado, se adjunta también un diagrama (ver Figura 2.4) que esquematiza la arquitectura de la Jetson Orin NX. En él, se puede apreciar cómo se organizan e interconectan entre sí los diferentes elementos de la plataforma. Como se observa, se utiliza el *Memory Controller Fabric* como elemento de interconexión, permitiendo al resto de elementos intercambiar datos sin generar cuellos de botella que afecten al rendimiento del sistema.

Por último, cabe destacar que el sistema operativo que se ejecuta sobre la plataforma se trata de un *Linux-4-Tegra* (diseñado específicamente para la familia Jetson de NVIDIA), que lleva integrado el *kernel 5.10.104-rt63-tegra*. Esta versión corresponde a un *kernel* al que se le ha implementado el parche *PREEMPT-RT*, el cual dota al sistema de capacidades de tiempo real. Esta característica es esencial para este trabajo, ya que permite al planificador de Linux expulsar tareas en ejecución cuando llega al sistema una tarea más prioritaria.

JETSON ORIN NX	
Procesador	ARM Cortex-A78AE 8 cores
Frecuencia máxima de la CPU	2 MHz
GPU	Arquitectura NVIDIA Ampere 1024 CUDA cores 32 tensor cores
Frecuencia máxima de la GPU	918 MHz
Memoria	16 GB
Rendimiento de IA	100 TOPS

Cuadro 2.1: Características principales de la NVIDIA Jetson Orin NX.

2.3.2. Cámara de profundidad RGB-D

Los sensores que incorpora el F1Tenth se encargan de percibir y proporcionar la información necesaria del entorno del vehículo para poder llevar a cabo su actividad de manera segura. La recopilación de datos precisos y en tiempo real permite que los algoritmos de las aplicaciones ejecutadas puedan procesar y analizar la información para poder tomar decisiones a tiempo durante la conducción autónoma.

En concreto, la cámara RGB-D (*RGB-Depth*) es un sensor que captura imágenes en color (RGB) junto con información de profundidad (D). Este tipo de cámara permite al sistema autónomo obtener una representación tridimensional del entorno, lo cual es utilizado para tareas como la detección de obstáculos durante la navegación y el reconocimiento de objetos.

El F1Tenth equipa una cámara Intel RealSense D435i, que además de proporcionar las características de una cámara RGB-D, incorpora una IMU (*Interial Measurement Unit*) que permite que las tareas de percepción mediante la cámara funcionen correctamente incluso en situaciones en las que la cámara esté en movimiento.

2.4. Trabajos previos relacionados

La conducción autónoma despierta cada vez un interés mayor. El desarrollo de sistemas complejos equipados con un gran número de sensores ha provocado que los sistemas ciber-físicos propuestos para este tipo de aplicaciones requieran una gran capacidad de cómputo (arquitecturas heterogéneas con varios núcleos, GPUs, NPUs...). Además, este tipo de sistemas

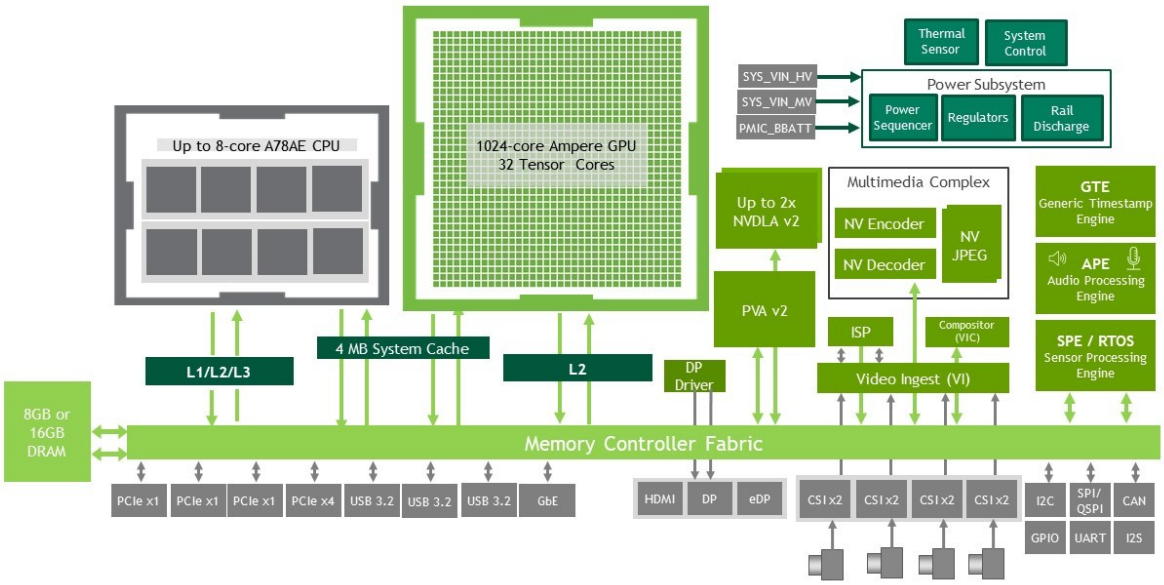

Figura 2.4: Diagrama de bloques de la arquitectura de la NVIDIA Jetson Orin NX [2].

debe satisfacer tanto requisitos de seguridad funcional como de tiempo real, lo cual plantea retos en los que se está trabajando intensamente y en los que aún quedan muchas cuestiones por resolver, lo cual explica que se existan numerosos trabajos de investigación en los que se aborden temas relacionados.

En el artículo [5], por ejemplo, se hace una revisión de la literatura más reciente en donde se estudia el uso de arquitecturas heterogéneas con una GPU integrada en aplicaciones de tiempo real. Los trabajos que se describen proponen diferentes soluciones para la estimación de cotas tanto para los tiempos de ejecución como para los tiempos de respuesta, proponiendo diferentes estrategias de optimización, entre las que destaca la mitigación de interferencias en la memoria.

Por su parte, en el artículo de investigación [9], se analizan los desafíos que supone la integración de aplicaciones complejas, paralelas, con una gran demanda computacional y varios niveles de criticidad en plataformas con GPUs con recursos hardware compartidos. En el artículo se proporciona una visión general de las contribuciones aportadas por diferentes trabajos de investigación respecto a los fallos hardware aleatorios, los fallos sistemáticos y la independencia de la ejecución.

Otro ejemplo es el artículo [10], publicado en el año 2023. En este trabajo de investigación se presenta una solución arquitectónica que aprovecha tecnologías de virtualización y los aceleradores hardware disponibles para apoyar la ejecución de aplicaciones de inteligencia artificial. Los requisitos de seguridad funcional y predictibilidad se satisfacen en este trabajo creando dos entornos de ejecución aislados: un dominio de alto rendimiento que ejecuta

algoritmos de aprendizaje profundo bajo el sistema operativo Linux y otro dominio crítico para la seguridad que ejecuta funciones de control y monitorización bajo el sistema operativo freeRTOS de tiempo real. El enfoque propuesto se valida mediante un caso de uso que consiste en un vehículo aéreo no tripulado capaz de rastrear objetivos en movimiento.

Por último, en el artículo [11], se presenta SPHERE, un framework destinado a simplificar la gestión de los recursos computacionales de una plataforma heterogénea. Esta propuesta aprovecha un hipervisor para virtualizar los recursos computacionales de la plataforma y aislar el comportamiento de diferentes subsistemas que se ejecutan en la misma plataforma, al tiempo que proporciona mecanismos de seguridad, protección y comunicación en tiempo real. Los principales desafíos abordados incluyen mecanismos de aislamiento para aplicaciones con criticidades mixtas, virtualización de una entrada/salida predecible y la gestión de redes sensibles al tiempo con flujos de tráfico heterogéneo. Esta arquitectura se valida mediante un caso de uso de conducción autónoma.

Capítulo 3

Estado del arte de los hipervisores para arquitecturas heterogéneas

En este capítulo, se presentará un análisis de los principales hipervisores disponibles actualmente orientados a la ejecución de tareas críticas de tiempo real que sean capaces de gestionar y administrar los recursos en plataformas heterogéneas con una GPU integrada (sin virtualizar la propia GPU). Para ello, se han explorado diversas opciones, considerando sus principales características y haciendo énfasis en aquellos aspectos concretos más relevantes de cara al desarrollo del proyecto. Para finalizar, se resumen las principales conclusiones que se extraen tras este estudio de cara al desarrollo de este proyecto.

3.1. Descripción de los hipervisores disponibles

- Jailhouse

  Es un hipervisor de tipo 1 implementado como un módulo del *kernel* de Linux capaz de ejecutar aplicaciones bare-metal u otros sistemas operativos adaptados [12]. Con este objetivo, Jailhouse configura las características de virtualización de la CPU y de otros recursos de la plataforma de ejecución de manera que ninguno de estos dominios, denominados "celdas", pueda interferir en el resto de manera no controlada. Además, Jailhouse no emula elementos de los que no disponga la plataforma hardware, por lo que no permite la sobreasignación de recursos. Solamente son virtualizados aquellos recursos software esenciales para la plataforma que no puedan ser particionados en hardware.

  La ventaja de este enfoque está en que, al no permitir la sobreasignación de recursos y proporcionar un acceso garantizado, Jailhouse ofrece un rendimiento predecible que

lo hace una solución adecuada para sistemas de tiempo real.

En cuanto al proceso de planificación, se lleva a cabo a nivel de celda, haciendo que cada una opere de manera independiente al resto con los recursos que le han sido asignados. Por último, respecto a su soporte para arquitecturas heterogéneas con una GPU integrada, algunas de las plataformas compatibles son las siguientes [12]:

- NVIDIA Jetson TX1/TX2.
- NXP MCIMX8M-EVK.
- LeMaker HiKey.

- PikeOS

  Es un hipervisor de tipo 1 diseñado para sistemas en los que se ejecuten aplicaciones críticas de tiempo real [13]. Una de sus principales características es su capacidad para ejecutar concurrentemente y de forma segura aplicaciones con diferentes niveles de criticidad y protección. Esto lo consigue mediante una estricta separación espacial y temporal entre estas aplicaciones a través de particiones software, basándose en el modelo de planificación definido en el estándar ARINC 653.

  Respecto a su implementación sobre arquitecturas hardware con una GPU integrada, se puede consultar en la página oficial de PikeOS un listado de las plataformas compatibles con este hipervisor [14]. En esta lista, se incluyen numerosos ejemplos que implementan arquitecturas heterogéneas como las plataformas Renesas R-Car H3, la Xilinx Zynq UltraScale+ MPSoC o la Jacinto 7 (J721S2) EVM, de Texas Instruments.

  Sin embargo, cabe destacar que PikeOS se trata de un hipervisor de carácter cerrado, por lo que sería necesario adquirir una licencia comercial que conlleva gastos económicos.

- XtratuM

  Es un hipervisor de tipo 1 desarrollado inicialmente por la Universidad Politécnica de Valencia y diseñado específicamente para sistemas empotrados críticos de tiempo real [15]. Sin embargo, las versiones posteriores han sido desarrolladas y comercializadas por fentISS bajo una licencia de uso comercial.

  Para cumplir los requisitos de tiempo real y seguridad funcional para la ejecución de este tipo de tareas, XtratuM proporciona un entorno de particionado robusto que permite ejecutar varias particiones de manera concurrente sobre la misma plataforma de ejecución. Además del particionado, también proporciona mecanismos de comunicación definidos en el estándar ARINC 653 entre las particiones.

  En cuanto a su compatibilidad con plataformas con una GPU integrada, existen actualmente proyectos en marcha para implementar este hipervisor sobre este tipo de plataformas. Es el caso del proyecto europeo METASAT [16], una iniciativa que tiene como objetivo diseñar una plataforma de procesamiento de alto rendimiento (que

incluye GPUs y otros aceleradores) para aplicaciones espaciales en la que se utiliza XtratuM para la correcta administración y gestión de los recursos. Por lo tanto, no existe todavía una versión disponible que soporte este tipo de arquitecturas.

Por otro lado, XtratuM es de carácter cerrado y su uso está restringido a la adquisición de una licencia comercial.

- Xen

  Xen [17] es un hipervisor de tipo 1 de código abierto desarrollado por la Universidad de Cambrdige cuyas principales características tienen como objetivo proporcionar a los sistemas un aislamiento seguro, un control de recursos, unas garantías de calidad de servicio y capacidad para realizar migración de máquinas virtuales de un servidor físico a otro sin necesidad de interrumpir su funcionamiento ni causar periodos de inactividad. Para ello, el hipervisor consiste en una capa de software que proporciona al usuario una abstracción del hardware subyacente permitiendo que varias máquinas virtuales se ejecuten simultáneamente. Estas máquinas virtuales reciben el nombre de dominios.

  Además, Xen cuenta con extensiones y configuraciones adicionales, como Xen-RT, diseñadas para mejorar sus prestaciones en contextos de tiempo real. Estas extensiones incluyen algoritmos de planificación que permiten desarrollar una planificación particionada estilo *time-table driven* como la utilizada por el estándar ARINC 653, más apropiada para la ejecución de sistemas con tareas críticas [18].

  Por otro lado, se han encontrado trabajos en los que se implementa este hipervisor en sistemas embebidos críticos. En el trabajo de investigación desarrollado por VanderLeest y White y descrito en el artículo [19], se implementa Xen sobre una Xilinx Zynq UltraScale+ MPSoC, plataforma hardware heterogénea que cuenta con una GPU integrada [20].

  Sin embargo, no se ha encontrado ningún trabajo en el que valide la extensión de Xen-RT sobre una plataforma heterogénea.

- CLARE-Hypervisor

  Es un hipervisor bare-metal de tipo 1 pensado para simplificar el desarrollo de sistemas ciber-físicos ofreciendo compatibilidad con plataformas de computación heterogéneas [21]. Proporciona la capacidad de definir diferentes niveles de criticidad para las tareas, brindando un fuerte aislamiento entre los diferentes dominios de ejecución e integrando mecanismos avanzados de seguridad, protección y gestión de recursos en tiempo real.

  En cuanto a la planificación de tareas, implementa políticas basadas en la asignación de prioridades fijas o en la planificación EDF (*Earliest Deadline First*), en la que las tareas más prioritarias son aquellas más próximas al vencimiento de su plazo.

  Por último, cabe destacar que, a pesar de ser un hipervisor reciente, ya se han desarrollado proyectos de investigación, como [22], en el que se implementa CLARE-Hypervisor

sobre la plataforma heterogénea Xilinx ZCU104 MPSoCs con el objetivo de dar soporte a sistemas ciber-físicos de tiempo real potenciados mediante inteligencia artificial. Por otro lado, en el proyecto SPHERE [11], se diseña una arquitectura Multi-SoC para sistemas ciber-físicos de próxima generación basado en plataformas heterogéneas en las que se implementa el hipervisor CLARE sobre una Xilinx Zynq Ultrascale+ MPSoC.

- KVM

  Es un hipervisor de tipo 2 de código abierto integrado en el *kernel* de Linux [23], del cual aprovecha gran parte de su funcionalidad, como la gestión de la memoria o sus algoritmos para la planificación de tareas en la CPU. KVM no ofrece servicios de virtualización de dispositivos hardware, sino que se apoya en herramientas externas que se ejecutan en el espacio de usuario, como QEMU, que le permiten ejecutar otros sistemas operativos sin necesidad de adaptarlos. De esta forma, permitiría transformar Linux en un hipervisor en el que cada máquina virtual que se ejecute sobre él se implementaría como un proceso habitual de Linux.

  Además, existe la posibilidad de integrarlo con PREEMPT-RT [24], un conjunto de parches para el *kernel* de Linux que proporciona capacidades de tiempo real mejoradas orientadas a la reducción de la latencia y a la obtención de unos tiempos de respuesta predecibles para las tareas.

  En cuanto a su soporte para el uso de GPUs, existe compatibilidad con plataformas de la familia Jetson de NVIDIA, como la Jetson Nano [25] y la Jetson Xavier [26], y plataformas como la Renesas R-Car M3 [27].

- NVIDIA DRIVE OS

  Se trata de un sistema operativo desarrollado por NVIDIA para sus propias placas. Está específicamente diseñado para plataformas de conducción autónoma y vehículos inteligentes que incorpora capacidades avanzadas de virtualización. Basado en la arquitectura NVIDIA DRIVE [28], proporciona una infraestructura de software robusta que incluye soporte para la planificación con particionado temporal, acceso a memoria con particionado espacial, mecanismos para la comunicación entre procesos de diferentes sistemas operativos, soporte para aplicaciones AUTOSAR [29] de tiempo real y uso compartido de la GPU en tiempo real.

  Al ser un sistema operativo propietario desarrollado por NVIDIA, es de carácter cerrado y requiere la adquisición de una licencia comercial para su utilización.

- Rust-Shyper

  Es un hipervisor reciente de tipo 1 construido con Rust y diseñado para ofrecer un alto rendimiento y una alta fiabilidad. Está pensado para plataformas integradas, en las que asegura un aislamiento entre las diferentes máquinas virtuales que estén ejecutándose. Además, ofrece servicios diferenciados para cada máquina virtual, pudiendo de esta manera proporcionar mecanismos de tiempo real específicos a aquellas máquinas virtuales que sean críticas.

Algunas de las plataformas heterogéneas con GPU integrada con las que es compatible el hipervisor Rust-Shyper son las siguientes:

- NVIDIA Jetson TX2.
- Raspberry Pi 4 Model B.
- Firefly ROC-RK3588S-PC.

3.2. Conclusiones

Tras finalizar la fase de exploración en la que se ha realizado una revisión de los principales hipervisores orientados a la ejecución de tareas críticas sobre plataformas heterogéneas con GPUs integradas, se proporciona la tabla del Cuadro 3.1 a modo de resumen en la que se recogen aquellas características que se han considerado más relevantes de cara a la realización de este proyecto y que ayudarán a concluir si alguno de los hipervisores que se han analizado satisface los requerimiento de este proyecto.

Como puede observarse, todavía no existe una solución de código abierto para las plataformas heterogéneas que proporcione un aislamiento espaciotemporal y que soporte una planificación *time-table driven*. Además, el estudio de la utilización de hipervisores para la ejecución de sistemas críticos de tiempo real sobre este tipo de arquitecturas es una tendencia que está en fase de investigación, como puede apreciarse mediante varios estudios recientes como [22] o [30].

Hipervisor	Tipo de licencia	Aislamiento	Planificación time-table driven	Arquitecturas CPU-GPU soportadas
Jailhouse	Código abierto	Espacial	No	NVIDIA Jetson TX1/TX2 Xilinx Zynq UltraScale+ NXP MCIMX8M-EVK LeMaker HiKey.
PikeOS	Comercial	Espaciotemporal	Sí	Renesas R-Car H3 Xilinx Zynq UltraScale+ Jacinto 7 (J721S2) EVM ...
XtratuM	Comercial	Espaciotemporal	Sí	Xilinx Zynq UltraScale+
Xen	Código abierto	Espacial	No	Xilinx Zynq UltraScale+
CLARE-Hypervisor	Comercial	Espaciotemporal	No	Xilinx ZCU104 Xilinx Zynq Ultrascale+
KVM	Código abierto	Espacial	No	NVIDIA Jetson Nano NVIDIA Jetson Xavier Renesas R-Car M3
NVIDIA DRIVE OS	Comercial	Espaciotemporal	Sí	Familia NVIDIA DRIVE
Rust-Shyper	Código abierto	Espaciotemporal	No	NVIDIA Jetson TX2 Raspberry Pi 4 Model B Firefly ROC-RK3588S-PC

Cuadro 3.1: Tabla resumen de las características de los hipervisores explorados.

Capítulo 4

Diseño e implementación de un entorno de pruebas sobre plataformas heterogéneas

En primer lugar, en este capítulo se realizará un análisis detallado de los requisitos del entorno que se va a desarrollar con el objetivo de compararlos con las conclusiones extraídas tras el estudio del estado del arte del capítulo anterior. Posteriormente, ante la falta de una solución adecuada dadas las necesidades, se describirá una propuesta alternativa para la planificación mediante un particionado temporal que permita gestionar la ejecución de aplicaciones críticas de tiempo real sobre la plataforma hardware. Para ello, primero se explicará de manera genérica la idea enumerando las razones que han motivado su propuesta y, a continuación, se profundizará en su funcionamiento y en cómo soluciona los problemas planteados. Finalmente, describen las pruebas desarrolladas para validar su comportamiento.

4.1. Análisis de requisitos del entorno de ejecución utilizado en el contexto del estado del arte

Como se ha descrito en el apartado 2.3.1, la unidad de procesamiento del entorno de ejecución se trata de una Jetson Orin NX, una plataforma heterogénea cuya arquitectura se describe en ese mismo apartado. Por lo tanto, es imprescindible que la capa de software que gestione los recursos hardware de la plataforma soporte este tipo de arquitecturas con una GPU integrada.

Por otro lado, y teniendo en cuenta que se van a ejecutar aplicaciones de conducción autóno-

ma críticas de tiempo real, se deberá ofrecer también un soporte para la planificación con particionado temporal estilo *time-table driven* con el objetivo de tomar el control sobre la activación de las aplicaciones para poder determinar el comportamiento temporal del sistema. Este tipo de planificación garantizará el aislamiento entre particiones y posibilitará el establecimiento de cotas temporales ayudando a mejorar la predictibilidad del sistema.

Sin embargo, tras el estudio del estado del arte desarrollado en el apartado anterior, se ha podido observar que todavía no existe una opción adecuada disponible que satisfaga los requisitos planteados previamente cuyo proceso de implantación en el entorno de ejecución utilizado pueda suponer un tiempo razonable. Dados los objetivos de este trabajo (descritos en el apartado 1.2) y el carácter cerrado de algunos de los hipervisores, la implantación de un hipervisor en la plataforma de ejecución queda fuera del alcance del proyecto.

Por lo tanto, ante la falta de una opción adecuada para satisfacer los requisitos planteados, se ha desarrollado una solución propia para emular el comportamiento del hipervisor respecto a la planificación de tareas que va a servir para poder desarrollar el entorno de pruebas descrito cumpliendo los requisitos establecidos.

4.2. Introducción a la solución propuesta

La alternativa que se propone consiste en el desarrollo e implementación de una capa de software que emule el comportamiento del hipervisor a la hora de realizar la planificación de las particiones y de las tareas ejecutadas en cada partición. El hecho de desarrollar esta solución propia permitirá personalizar la fase de diseño de manera que se puedan implementarse todas las funcionalidades descritas en el apartado anterior.

Cabe destacar que, mediante el desarrollo de esta alternativa, no se conseguiría sustituir el hipervisor como tal, por lo que el sistema no serviría como producto final, sino que esta solución se plantea para utilizarse como base para el desarrollo de un entorno de pruebas que facilite llevar a cabo estudios de modelado sobre sistemas particionados.

A esta propuesta propia para la planificación de tareas emulando un particionado temporal se le ha bautizado *Planificador mediante Ventanas Temporales sobre Linux*, por sus siglas, *PVTL*, nombre que será utilizado durante el resto del documento para referirse a él.

Siguiendo con la explicación, este planificador PVTL se ejecutará sobre el sistema operativo Linux de tiempo real descrito en el apartado 2.3.1 (ver Figura 4.1). El planificador PVTL será el encargado de ejecutar las aplicaciones críticas bajo un particionado temporal de manera que sea posible garantizar los requisitos de tiempo real correspondientes. Por lo tanto, esta capa de software actuará como un gestor de grupos de hilos que emulará el particionado

deseado aprovechándose de la planificación mediante prioridades fijas de Linux, que serán modificadas dinámicamente durante la ejecución del sistema.

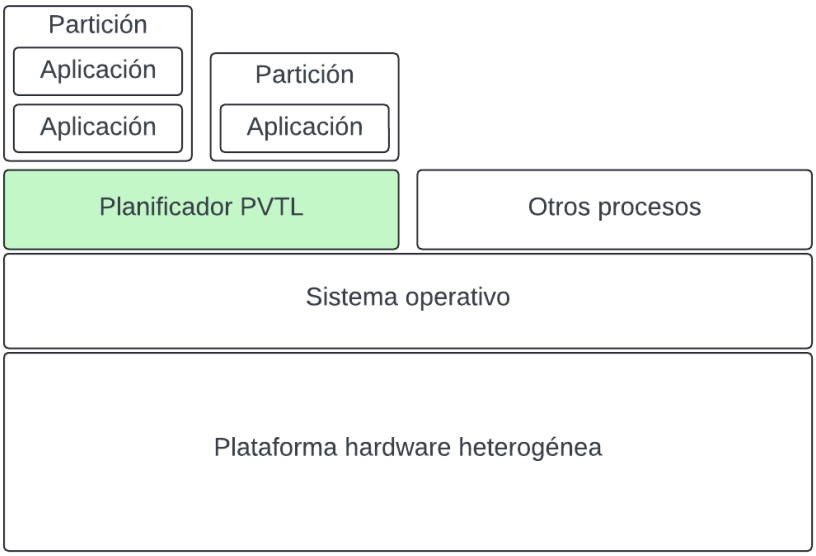

Figura 4.1: Arquitectura del planificador propuesto.

4.3. Descripción del gestor de hilos

Como se ha mencionado previamente, se debe garantizar que la ejecución de las tareas del sistema se realice de forma segura. Para ello, la manera de garantizar esta seguridad será aislando entre sí cada una de las diferentes tareas con el objetivo de evitar interferencias no acotadas al hacer uso de los recursos hardware compartidos.

Para lograr este aislamiento, se llevará a cabo una planificación particionada del estilo *time-table driven* similar a la definida en el estándar ARINC 653. Para ello, se dividirá la línea temporal del flujo de la ejecución en ventanas temporales, cada una de una determinada duración de acuerdo a las siguientes características:

- El número de ventanas será fijo.
- Cada ventana temporal tendrá asociada una partición y dentro de cada partición se podrán ejecutar una o varias tareas que serán planificadas en base a prioridades fijas. La misión de la ventana es limitar el tiempo durante el cual se podrán ejecutar las tareas de su partición asociada. Por lo tanto, estas tareas no podrán ejecutarse fuera de la ventana temporal asociada a la partición a la que pertenecen.

- El orden de ejecución de las ventanas estará predefinido mediante una tabla en la que se indicará también la duración de cada ventana.
- La ejecución de las ventanas temporales se repetirá cíclicamente de manera indefinida.

De esta manera, mediante este mecanismo de ventanas temporales, se consigue acotar el tiempo durante el cual cada tarea puede hacer uso de los recursos hardware. En el instante en el que se agote la duración de una determinada ventana, su partición asociada sería expulsada de la CPU, comenzaría la siguiente ventana temporal definida en la tabla y entraría a ejecutarse la nueva partición correspondiente.

A la partición que se encuentre ejecutándose en un determinado instante de tiempo se le llamará partición activa, mientras que al conjunto del resto de particiones se les denominará particiones en espera. Cada partición podrá estar formada por numerosos hilos de ejecución, que serán los encargados de ejecutar la aplicación en cuestión correspondiente a la partición.

El comportamiento del planificador PVTL se ilustra en el diagrama de la Figura 4.2. En él, se representa de manera simplificada la alternancia ordenada entre las particiones en función del instante de tiempo actual de la ejecución. Durante cada ventana, se ejecuta únicamente su partición asociada (correlación representada utilizando el mismo número identificativo para cada una) y, cuando finalizase la ejecución de la tercera ventana, se repetiría en bucle la ejecución de todo el bloque comenzando de nuevo desde la primera.

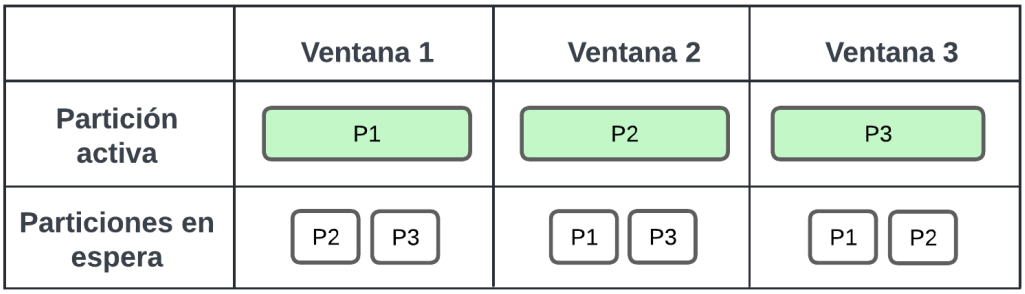

Figura 4.2: Modelo de planificación de particiones.

4.4. Detalles de implementación

El PVTL se ha desarrollado en lenguaje C y se ha implementado sobre el sistema ejecutándose como un proceso de Linux. Por tanto, las aplicaciones que se quieran ejecutar sobre este

entorno particionado serán lanzadas sobre él de manera monolítica y no como procesos de Linux independientes.

En cuanto a su implementación, el PVTL va a consistir en un proceso formado por varios conjuntos de hilos, cada uno de los cuales se comportará como una partición diferente. Los hilos que pertenezcan a cada uno de estos conjuntos serán los encargados de ejecutar las tareas que correspondan a esa partición.

A la hora de implementar el PVTL, se ha planteado su diseño de manera que sea capaz de abordar los retos que se derivan de sus requisitos funcionales descritos en el apartado anterior. Estos retos son, principalmente, los siguientes:

- En primer lugar, debe llevar a cabo una política de planificación *time-table driven* para las particiones en la que, de manera cíclica, planificará la partición activa correspondiente a cada ventana temporal manteniendo el resto de particiones a la espera.

  Esto implica que tendrá que ser capaz de realizar cambios de contexto entre particiones, estando cada una de ellas formada por varios hilos de ejecución.

- En segundo lugar, deberá garantizar el aislamiento entre particiones asegurando que cada una de ellas se ejecuta únicamente durante el periodo de tiempo definido por su ventana temporal asociada. Esto implica manejar dos tipos de situaciones:

  - Se deberá asegurar que durante los periodos de tiempo en la ejecución de una partición en los que no haya ningún hilo activo, no deberá entrar a ejecutarse ningún otro hilo de ninguna otra partición a pesar de estar la CPU libre.
  - Por otro lado, deberá garantizarse que en los instantes de tiempo en los que haya un hilo ejecutándose y se agote su ventana temporal correspondiente, ese hilo es forzado a liberar la CPU aún no habiendo finalizado su ejecución para dejar paso al hilo que corresponda de la siguiente partición que entra a la CPU.

Se puede apreciar, por tanto, que debe llevarse a cabo una planificación a dos niveles:

- Un primer nivel, en el que se planifiquen las diferentes particiones atendiendo, por un lado, a la tabla que define su orden de ejecución y, por otro lado, a la duración de sus ventanas temporales asociadas.

- Un segundo nivel, en el que se planifiquen los diferentes hilos dentro de cada partición activa atendiendo a la prioridad de cada uno.

De esta manera, para comenzar a entender cómo se han abordado estos retos, se va a comenzar por describir, primero, los diferentes elementos que forman la estructura del planificador PVTL y, segundo, una división del rango de prioridades disponible en Linux necesaria para poder llevar a cabo la planificación multinivel que se plantea.

4.4.1. Elementos del planificador PVTL

El PVTL se estructura en base a los siguientes elementos:

- Hilo asignador de prioridades: hilo que se encarga de la planificación de las particiones modificando las prioridades de los hilos por los que están formadas. Este hilo se ejecutará a la máxima prioridad del sistema (detallado en el apartado 4.4.2).
- Partición activa: partición en ejecución que podrá hacer uso de los recursos de procesamiento durante la ventana temporal actual. En el momento en el que comience la ventana temporal asociada a la partición activa, los hilos de dicha partición adquirirán una prioridad dentro rango alto (desarrollado en el apartado 4.4.2).
- Particiones en espera: conjunto de particiones del sistema que están a la espera de ser planificadas en futuras ventanas. Los hilos que formen parte de este conjunto de particiones tendrán una prioridad dentro del rango bajo (explicado en el apartado 4.4.2).
- Hilo *dummy*: hilo cuyo único objetivo será mantener a la CPU ocupada durante los periodos de tiempo en los que no se esté ejecutando ningún hilo de la partición activa. De este modo, se evita que se ejecuten hilos de otras particiones durante estos tiempos muertos, lo cual rompería el aislamiento. Para ello, este hilo estará continuamente ejecutándose comiendo tiempo a una prioridad intermedia, por debajo de la partición activa pero por encima de las particiones en espera, de tal manera que cuando no haya activo ningún hilo de la partición activa, el siguiente hilo más prioritario del sistema será este hilo *dummy*. En estas situaciones, el hilo *dummy* comenzará a comer tiempo evitando que entre a ejecutarse el hilo más prioritario de las particiones en espera y haciendo que se cumpla el aislamiento entre particiones.
- Estructura de datos en la que se define:
  - La duración de cada una de las ventanas temporales del sistema.
  - El orden en el que se ejecutan.
  - Los hilos por los que está formada cada una, lo cual permite saber a qué hilos se deberá modificar la prioridad en los cambios de contexto entre particiones.

4.4.2. División del rango de prioridades disponibles en Linux

Para llevar a cabo la planificación multinivel que se ha descrito en el apartado 4.4, se ha decidido dividir el rango de prioridades disponibles en Linux (de la prioridad 1 a la 99) de la siguiente manera:

- Prioridad 99: prioridad a la que se ejecutará el hilo asignador de prioridades.
- Rango de prioridades altas ([50, 97]): prioridades disponibles únicamente para los hilos que formen parte de la partición activa. Asignándoles una prioridad mayor que al resto de hilos de otras particiones se consigue que, durante la ventana correspondiente, solamente se ejecuten hilos de la partición activa.
- Rango de prioridades bajas ([1, 48]): rango utilizado para todo el resto de hilos que formen parte del conjunto de particiones que estén en espera.
- Prioridad 49: prioridad a la que se ejecutará el hilo *dummy* que se describía en el apartado anterior. Se trata de una prioridad intermedia entre los rangos definidos previamente.

  De esta manera, cuando se den periodos de tiempo durante la ejecución de la partición activa en los que no haya trabajo que realizar y la CPU se libere, este hilo *dummy* se convertirá automáticamente en el hilo activo más prioritario del sistema, por lo que se comenzará a ejecutar e impedirá que se ejecuten hilos de otras particiones, evitando así ejecuciones fuera de las ventanas temporales asignadas a las particiones y cumpliendo el aislamiento. Para ello, este hilo estará permanentemente activo.

Mediante la Figura 4.3 se aporta un diagrama de estados que esquematiza el comportamiento de la ejecución del sistema. Se observa cómo una partición pasa del estado *Espera* al estado *Ejecución PA* al iniciarse la ventana temporal a la que pertenece. Al comenzar a ejecutarse, se debe modificar la prioridad de sus hilos y asignarles un valor dentro del rango alto (*entry action*) y reducirla a la prioridad original dentro del rango bajo al finalizar su ventana temporal (*exit action*). Por último, cuando no haya ningún hilo activo de la partición en ejecución, se comenzará a ejecutar el hilo *dummy* hasta que vuelva a activarse un hilo de la partición activa.

4.4.3. Configuración del planificador PVTL

Para llevar a cabo la configuración del PVTL, se desarrollaron dos ficheros de configuración, *config.txt* y *tasks.txt*, utilizados para definir diferentes aspectos estructurales.

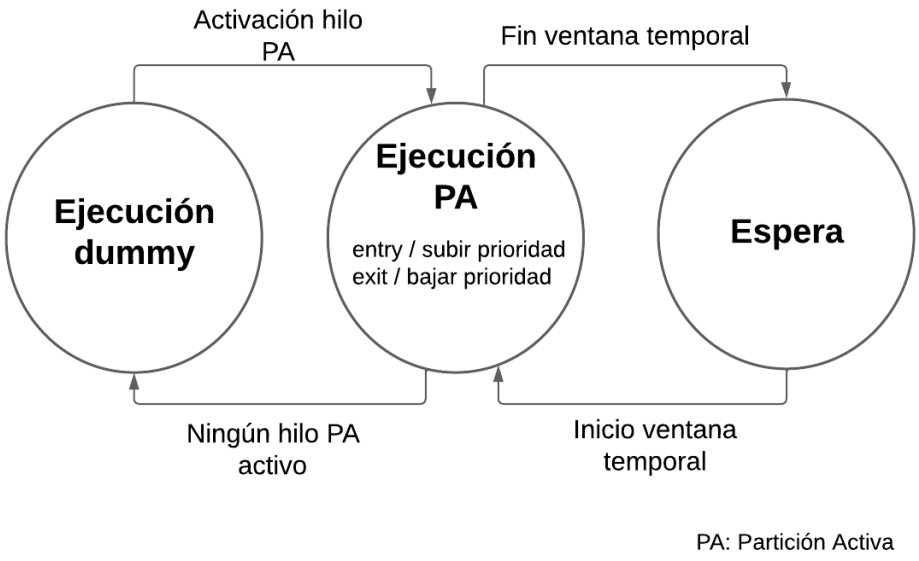

Figura 4.3: Diagrama de estados de las particiones del sistema.

4.4.3.1. Fichero de configuración *config.txt*

Este fichero está diseñado para establecer parámetros de configuración relativos a la estructura del PVTL. Estos parámetros son el número total de ventanas temporales que conforman la ejecución del sistema y la duración de cada una de ellas definida con precisión de nanosegundos.

A la hora de introducir estos datos, habrá que tener en cuenta que el número total de ventanas indicado deberá ser igual al número de ventanas para las que se defina un determinado tiempo.

4.4.3.2. Fichero de configuración *tasks.txt*

Este fichero es utilizado para definir diferentes parámetros que serán utilizados durante el proceso de validación del funcionamiento del PVTL para lanzar tareas que coman tiempo durante un determinado periodo simulando así su ejecución.

Este mecanismo servirá como proceso de validación inicial previa a la integración con algoritmos reales, permitiendo personalizar las características de las aplicaciones según convenga. De esta manera, se consigue proporcionar un mayor control sobre la ejecución (simulada) de las aplicaciones facilitando así el proceso de pruebas.

Los parámetros que se definen en este fichero son, en primer lugar, el número total de hilos que ejecutarán la aplicación simulada que se va a lanzar para realizar las pruebas. En segundo lugar, se define, para cada una de estos hilos, los siguientes datos:

- Un identificador para distinguir y gestionar individualmente cada uno de los hilos.
- El identificador de la partición a la que pertenecen, que se corresponde a su vez con el identificador de la ventana temporal correspondiente.
- El periodo de activación del hilo.
- Su tiempo de ejecución de peor caso.
- La fase, lo cual determina su instante inicial de ejecución en relación al instante de ejecución inicial del sistema.

En este caso, habrá que tener en cuenta que no será posible definir un número de tareas diferente al que se ha especificado en el parámetro de configuración del número total de tareas y, además, no se podrá asignar una tarea a una partición que se corresponda con una ventana temporal que no haya sido definida previamente en el fichero *config.txt*.

Por otro lado, será necesario también tener en cuenta que el número de prioridades a utilizar por cada una de las aplicaciones a ejecutar está limitado por el tamaño de los rangos en los que se habían dividido las prioridades disponibles de Linux. Recordando lo que se explicaba en el apartado 4.4.2, los rangos de prioridades altas y bajas van desde la prioridad 50 hasta la 97 y desde la prioridad 1 hasta la 48 respectivamente. Por tanto, una aplicación que se vaya a ejecutar sobre este pseudo-planificador no podrá utilizar más de 48 prioridades diferentes, valor que se corresponde con el tamaño de los rangos de prioridades disponibles.

Finalmente, cabe destacar que las prioridades originales asignadas mediante este fichero de configuración deberán estar dentro del rango de prioridades bajas. Será el hilo asignador de prioridades (el que se ejecutaba a la máxima prioridad del sistema) el encargado de elevar la prioridad de los hilos de la partición activa cuando sea necesario y reducirla a su valor original cuando se agote su correspondiente ventana temporal. Esta conversión de prioridad alta a prioridad baja y viceversa se llevará a cabo simplemente sumando o restando a cada prioridad el valor de la prioridad del hilo *dummy* (49). Al haber diseñado ambos conjuntos de prioridades con el mismo número de elementos y existir una correspondencia uno a uno entre ellos, se simplifica este proceso aplicando esta función biyectiva.

4.5. Pruebas de validación realizadas

Por último, se procederá a detallar las pruebas de validación realizadas para el planificador PVTL. En primer lugar, se ha realizado un análisis de los tiempos de cambio de contexto entre particiones. En segundo lugar, se ha diseñado un escenario de pruebas que servirá para verificar el correcto funcionamiento del planificador PVTL. Para explicar el desarrollo de este proceso de validación, primero se describirán los parámetros de configuración tanto del PVTL como de las tareas que se simulen; a continuación, se proporcionará un diagrama a modo de esquema que facilite entender cómo debería comportarse la ejecución y, por último, se presentarán los resultados obtenidos tras la ejecución del sistema.

4.5.1. Análisis de los tiempos de cambio de contexto

Para analizar los tiempos de cambio de contexto entre particiones introducidos por la gestión de las prioridades de los hilos por parte del planificador PVTL, se ha diseñado una prueba modificando ligeramente el código del PVTL para calcular el tiempo que transcurre desde que se activa el hilo asignador de prioridades hasta que comienza a ejecutarse la primera tarea de la partición que entra a la CPU. Tras ejecutar esta prueba, los resultados que se han obtenido se representan en el gráfico de la Figura 4.4.

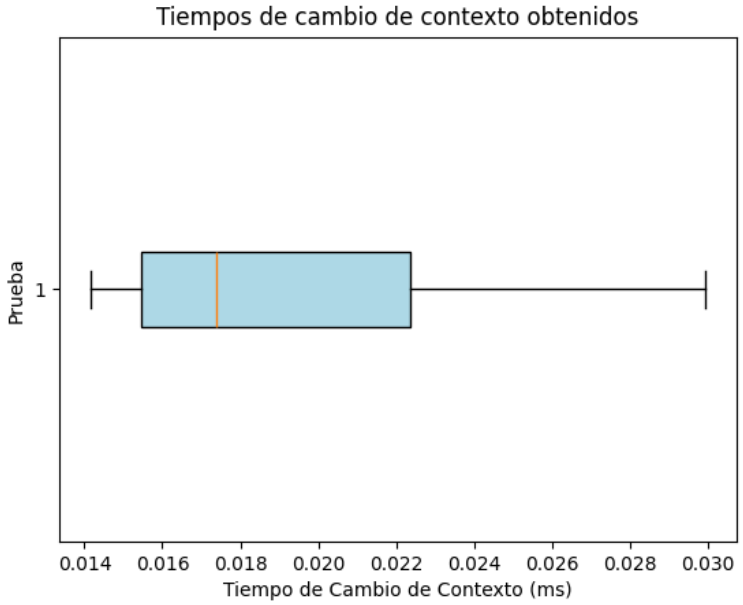

Figura 4.4: Resultados de los tiempos de cambio de contexto obtenidos introducidos por la gestión del planificador PVTL.

Este gráfico es un diagrama de cajas que muestra la distribución de los tiempos obtenidos y es útil para visualizar la dispersión de los resultados. Dentro de la caja se encuentran el 50 % de los datos registrados; la línea de dentro de la caja representa la mediana y las líneas que se extienden a ambos lados llegan hasta los valores mínimo y máximo obtenidos. Por tanto, se ve cómo el tiempo del cambio de contexto suele estar en torno a los 0.017 ms aproximadamente, pudiendo llegar hasta un máximo de unos 0.03 ms según los resultados obtenidos.

Es interesante conocer este dato porque es posible que existan situaciones en las que tenga un impacto alto. El tiempo del cambio de contexto se va a mantener siempre fijo (respetando la variabilidad recogida en el gráfico de la Figura 4.4) independientemente del tamaño de las ventanas temporales de las particiones que se definan en el sistema. Por lo tanto, cuanto menor sean los tamaños de las ventanas, mayor será, en proporción, el tiempo de los cambios de contexto, lo cual habrá que tener en cuenta a la hora de diseñar la configuración de las particiones.

4.5.2. Parámetros de configuración del escenario de pruebas

Para diseñar el escenario de pruebas que sirva para verificar el correcto comportamiento de las funcionalidades descritas a lo largo de este capítulo, se han configurado los ficheros *config.txt* y *tasks.txt* de manera que las tareas ejecutadas se lanzan de acuerdo a las características que se recogen en la tabla del Cuadro 4.1. Por su parte, en la Figura 4.5 se muestra el diseño realizado de la estructura del MAF (*Major Frame*), es decir, la ventana global del sistema que está formada por el conjunto de todas las ventanas temporales definidas y cuya ejecución se repite cíclicamente.

Se ha configurado el escenario de pruebas para que todas las tareas se activen en el instante inicial del sistema, por lo que el valor de la fase de cada tarea se ha dejado a valor 0.

4.5.2.1. Diagrama de la ejecución del sistema

Dadas las características del escenario configurado, el comportamiento esperado del sistema se muestra en los diagramas que se presentan a continuación. El primer diagrama (Figura 4.6) representa el primera ciclo de ejecución del MAF, mientras que el segundo (Figura 4.7) representa el segundo ciclo.

A través de estos dos diagramas, se observan las siguientes situaciones respecto a cada una de las particiones:

ESCENARIO DE PRUEBAS				
		WCET (s)	T (s)	Prioridad
Partición 0	Tarea 0	0.100	0.9	46
	Tarea 1	0.025	8	26
Partición 1	Tarea 2	0.200	6	45
	Tarea 3	0.150	7	41
Partición 2	Tarea 4	0.075	3	32
	Tarea 5	0.100	4	35
	Tarea 6	0.025	3	27
Partición 3	Tarea 7	0.050	6	17
	Tarea 8	0.175	4	6
	Tarea 9	0.100	4	38

Cuadro 4.1: Parámetros de las tareas a ejecutar en el escenario de pruebas.

- Respecto a la partición 0, la suma de los tiempos de ejecución de sus tareas (100 ms la tarea 0 y 25 ms la tarea 1) es menor que la duración de su correspondiente ventana temporal (150 ms). Por lo tanto, se ve cómo durante este periodo de tiempo en el que no hay activa ninguna tarea, se ejecuta el hilo *dummy* hasta que finalice la duración de la ventana actual y se produzca el cambio de contexto entre particiones. De esta manera, se evita que se ejecute el siguiente hilo más prioritario de otra partición en espera que, en este caso, sería la tarea 2.

  Además, durante la ventana 3, la tarea 0 se activa y se observa cómo es atendida en la siguiente ventana correspondiente a la partición 0 (en el segundo ciclo del MAF).

- En cuanto a la partición 1, el tiempo total de ejecución de sus tareas (350 ms) sí que supera en este caso la duración de su ventana (300 ms). Por tanto, el comportamiento esperado es que la tarea 3 sea expulsada de la CPU en el instante en el que se agote su ventana, antes de que haya podido finalizar su ejecución. Ante esta situación, quedarían pendientes 50 milisegundos de ejecución que serían ejecutados en el siguiente ciclo del MAF, cuando vuelva a planificarse la partición 1.

- En el caso de la partición 2 sucede lo mismo que ocurría al ejecutar la partición 1: el tiempo de ejecución de sus tareas (200 ms en total) no supera la duración de su ventana (250 ms). Por lo tanto, al terminar de ejecutar todas las tareas, se ejecutará el hilo *dummy* durante otros 50 ms para mantener el aislamiento.

- Por último, respecto a la partición 3, la situación es similar a la de la partición 1:

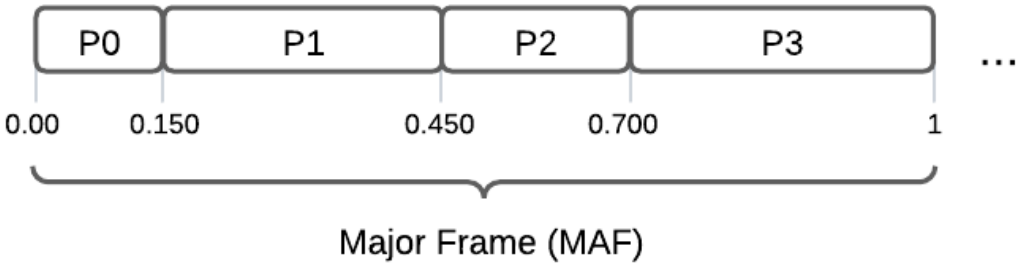

Figura 4.5: Diagrama de la estructura del MAF del escenario de pruebas.

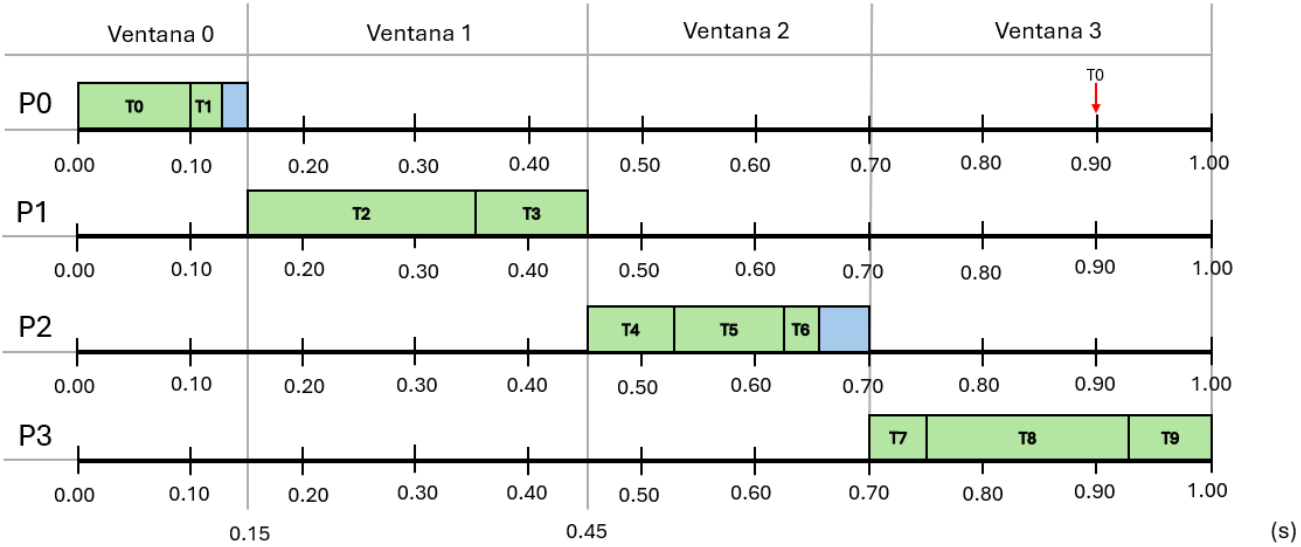

Figura 4.6: Diagrama que representa la primera parte de la ejecución del sistema.

la suma de los tiempos de ejecución de sus tareas (325 ms) supera la duración de su ventana temporal (300 ms). Por lo tanto, la tarea 9 es expulsada en el instante en el que finaliza la ventana y terminará su ejecución (25 ms restantes) cuando vuelva a planificarse la partición 3 en el siguiente ciclo del MAF.

4.5.2.2. Resultados obtenidos

Finalmente, tras haber ejecutado el sistema descrito en el en el entorno desarrollado, los resultados que se han obtenido se presentan a continuación.

En primer lugar, cabe destacar que, para poder analizar el comportamiento del sistema, se ha modificado el código original del planificador PVTL para que se muestre mediante *logs*

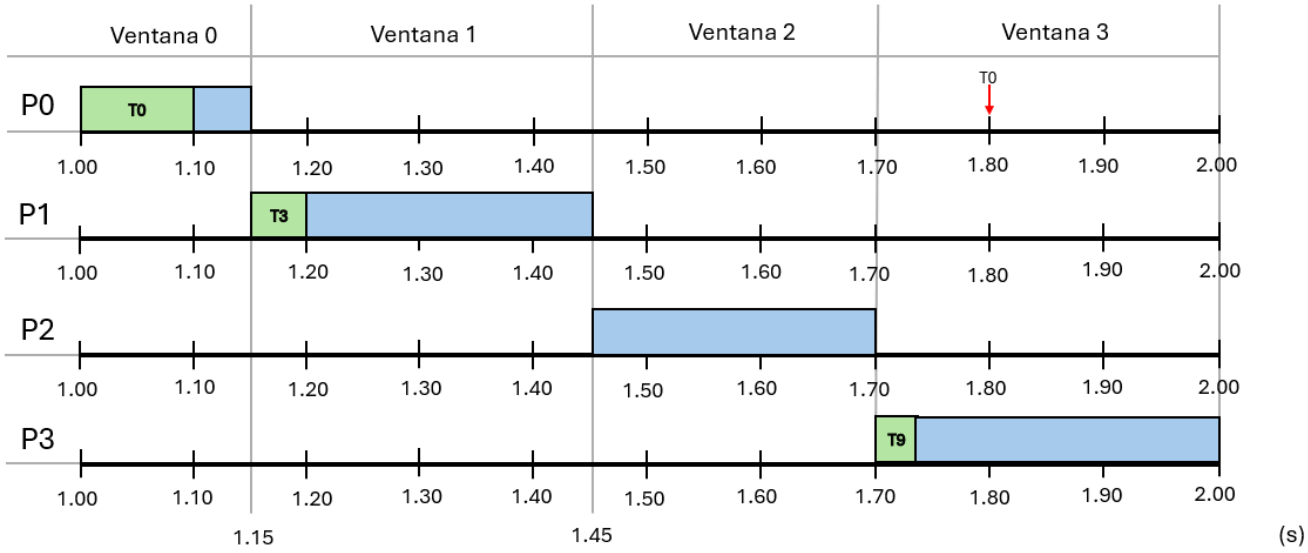

Figura 4.7: Diagrama que representa la segunda parte de la ejecución del sistema.

lo que vaya ocurriendo durante la ejecución junto al instante de tiempo en el que tiene lugar cada suceso (inicio y fin de las ejecuciones de las tareas y del hilo *dummy* y los cambios de contexto entre particiones).

Los mensajes de *log* obtenidos tras la ejecución del escenario de prueba se adjuntan en el Listing A.1 del apéndice A del anexo de este documento.

En ellos, puede verse cómo se comienza reportando el instante en el que se activa el hilo asignador de prioridades para planificar una nueva partición. A continuación, se informa de la partición que ha sido planificada, los instantes de inicio y fin (en caso de que finalice) de las tareas que se ejecuten dentro de esa partición y del inicio del hilo *dummy* (si procede).

Mediante la lectura detenida de estos mensajes se comprueba que el comportamiento que describen coincide con el que se representaba en los diagramas temporales de la Figura 4.6 y de la Figura 4.7. Estos mensajes muestran el siguiente funcionamiento:

- Los hilos pertenecientes a cada partición se ejecutan únicamente durante su ventana temporal correspondiente, respetando así el particionado.
- Una tarea en ejecución es expulsada de la CPU en el instante en el que se agote su ventana temporal aunque no haya finalizado su ejecución. Esto se puede observar con las tareas 3 y 9, en las que se ve cómo se reporta el inicio de sus ejecuciones durante sus correspondientes ventanas en el primer ciclo del MAF pero no se informa del instante de finalización hasta el segundo ciclo, en el que completan el cómputo que les quedaba pendiente.

- El hilo *dummy* se ejecuta durante los periodos de tiempo en los que no se ejecute ningún otro hilo de la partición activa para asegurar el aislamiento. Esto se ve en las particiones 0 y 2 del primer ciclo del MAF, en donde se reporta el instante del inicio de la ejecución del hilo *dummy* al finalizar las tareas de cada una de las particiones, y en todas las particiones en el segundo ciclo del MAF.
- Se observa también cómo una activación de una tarea que se produce mientras la partición a la que corresponde está en espera (como es el caso de la tarea 0) se atiende en su siguiente ventana temporal correspondiente, en el siguiente ciclo del MAF.

Por último, al haber coincidido los instantes de inicio y fin de ejecución obtenidos para cada tarea con los esperados mediante el análisis teórico, se entiende que, dado un tamaño de las ventanas de las particiones del orden de 10^{-2} segundos, los cambios de contexto no han tenido un impacto notable en los tiempos de respuesta de cada tarea.

Capítulo 5

Definición, implementación e integración del caso de uso

En este capítulo, se comenzará definiendo de manera genérica la estructura del caso de uso elaborado basado en una aplicación de conducción autónoma. A continuación, se detallará cada una de las partes por las que está formado y se profundizará en el proceso de implementación. Finalmente, se describirá el proceso de integración, en el que se ha implementado el caso de uso sobre el planificador PVTL con el objetivo de analizar el comportamiento del entorno de ejecución desarrollado al ejecutar una aplicación real de conducción autónoma.

5.1. Definición general del caso de uso

El caso de uso que se ha desarrollado para validar el entorno diseñado consiste en la aplicación de un algoritmo de detección de obstáculos sobre una imagen capturada mediante la cámara RGBD del F1Tenth. Para ello, puede hacerse una división conceptual del caso de uso en tres partes diferentes representadas en el diagrama de la Figura 5.1. Estas tres partes son la captura de la imagen sobre la que se va a realizar la detección de obstáculos tomada mediante la cámara RGBD a bordo del F1Tenth, el preprocesamiento de dicha imagen necesario para poder trabajar con ella de una manera más eficiente y con un formato compatible al necesario en etapas posteriores y, finalmente, la aplicación del algoritmo DBSCAN (*Density-Based Spatial Clustering of Applications with Noise*) [31]: un algoritmo de detección de obstáculos que facilitará un posterior análisis para poder tomar las decisiones que correspondan para una conducción autónoma segura.

Sin embargo, a la hora de realizar la implementación del algoritmo de detección de obstáculos,

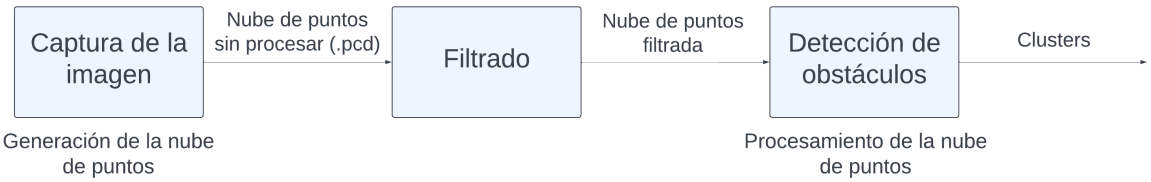

Figura 5.1: Esquema del comportamiento del caso de uso.

se ha tenido que separar la parte de la captura de la parte de detección por cuestiones de tiempo. A pesar de que se realiza el ciclo completo descrito en el diagrama de la Figura 5.1, en la implementación final del algoritmo de detección de obstáculos se lleva a cabo el proceso de detección sobre una nube de puntos generada con anterioridad. Es decir, primero se ha de tomar una captura y guardar los datos relativos a la imagen en un fichero para, posteriormente, obtener la nube de puntos en el algoritmo de detección de obstáculos a partir de este fichero.

A continuación, se describirá de manera más detallada cada una de las partes que forman el caso de uso descrito.

5.2. Implementación del caso de uso

Como se ha mencionado en el apartado anterior, el caso de uso puede dividirse en tres partes diferentes, las cuales se detallan a continuación.

5.2.1. Captura de la imagen mediante la cámara RGBD

Como se ha descrito en el apartado 2.3.2, del documento, la cámara RGBD permite al sistema autónomo (en este caso, al vehículo autónomo F1Tenth) obtener una representación tridimensional del entorno mediante un sensor que captura imágenes en color (RGB) junto con información de profundidad (D).

La cámara RGBD utilizada en este proyecto, la Intel RealSense D435i, dispone de un repositorio oficial [32] en el que se proporciona todo el código fuente de los drivers y las librerías necesarias para poder utilizar toda la gama de cámaras de profundidad de Intel junto a documentación y programas de ejemplo que ilustran cómo manejarla.

Para elaborar la primera parte del caso de uso diseñado, se tomó como programa de base la

aplicación *rs-distance.c*, ubicada en la ruta */librealsense/examples/C/distance* del repositorio. Este programa de prueba demuestra cómo utilizar la API de C para recoger los datos relativos a la profundidad tomados por la cámara y obtener la distancia desde la cámara hasta el objeto ubicado en el centro de la imagen, de tamaño 640x480 píxeles. Sobre él, se realizaron una serie de modificaciones para conseguir volcar los datos de profundidad recogidos para cada uno de los píxeles a un fichero externo en formato PCD (Point Cloud Data) [33].

El formato PCD es un estándar de archivo utilizado para almacenar nubes de puntos tridimensionales. Este formato fue desarrollado como parte de la librería Point Cloud Library (PCL) [34] y está diseñado para almacenar datos espaciales de manera eficiente. Los ficheros PCD pueden contener diferente tipo de información de cada uno de los puntos, como sus coordenadas en el espacio (X, Y, Z), colores (RGB) y otras propiedades en función de la aplicación. En este caso, se ha utilizado este formato de fichero para aprovechar la información de profundidad tomada para cada píxel de la imagen y convertir los píxeles en puntos, ya que el formato PCD permite almacenar para cada punto sus coordenadas (X, Y, Z) junto a la distancia hasta él. De esta manera, se consigue transformar la imagen en una nube de puntos sobre la que luego se podrá aplicar un algoritmo de detección de obstáculos.

Por lo tanto, en lugar de, como hace el programa de ejemplo, filtrar todos los píxeles de la imagen para quedarse solo con el píxel central y calcular la distancia al objeto representado en dicho píxel mediante su información de profundidad, lo que se hace ahora es almacenar esa información de profundidad relativa a cada uno de los píxeles en un fichero aparte en formato PCD, que será utilizado posteriormente como entrada para el algoritmo de detección de obstáculos aprovechando su compatibilidad con la Point Cloud Library.

De esta manera, el comportamiento de esta parte de la aplicación se resume en el Listing 5.1:

```
Inicializar el sistema y las variables necesarias.
Establecer una conexion con la camara y configurar el dispositivo
   para iniciar la captura y transmision de datos.
Recibir la informacion correspondiente a una imagen capturado por la
    camara.
Almacenar en un array los datos de profundidad para cada uno de los
   pixeles de la imagen.
Volcar los datos de profundidad junto a las coordenadas de cada
   pixel en un fichero con formato PCD.
Liberar los recursos y finalizar el programa.
```

Listing 5.1: Psuedocódigo del programa encargado de capturar una imagen mediante la cámara RGB-D

5.2.2. Filtrado de la nube de puntos generada

Una vez generada la nube de puntos en formato PCD a raíz de la imagen tomada mediante la cámara RGB-D, se procedió a realizar una serie de pruebas de visualización de la nube de puntos obtenida para comprobar que la detección se estaba realizando correctamente. Para la visualización, se ha utilizado una herramienta proporcionada en el propio repositorio del algoritmo DBSCAN [35] basada en la librería VTK [36].

Sin embargo, al ejecutar la aplicación DBSCAN sobre la GPU, se obtenía el siguiente error:

```
Terminate called after throwing an instance of
'thrust::system::detail::bad_alloc'
  what(): std::bad_alloc: cudaErrorMemoryAllocation: out of memory
Abortado ('core' generado)
```

Este mensaje de error indica que se ha agotado la memoria de la GPU durante la ejecución del algoritmo de agrupamiento. Esto ocurre cuando el programa intenta asignar más memoria en la GPU de la que está disponible.

Por lo tanto, esta situación obligaba a aplicar una serie de filtros sobre la nube original generada a partir de la imagen capturada para reducir su número de puntos. Además, estos filtros también ayudarían a mejorar el proceso de detección de obstáculos, ya que mejorarían la calidad y la precisión del agrupamiento reduciendo el ruido de la imagen y mejorando la densidad de puntos.

A continuación, se detallan los filtros utilizados sobre la nube original, de 640x480 píxeles y, por tanto, de 307.200 puntos, y su evolución tras la aplicación de cada uno de ellos.

- Voxel grid

  El filtro voxel es un método utilizado en el procesamiento de nubes de puntos para reducir su densidad y, por tanto, su número de puntos. Mediante la aplicación de este filtro, se divide el espacio correspondiente a la nube de puntos en cuadrículas tridimensionales con forma de cubos denominadas *voxels*. De esta manera, cada voxel agrupa todos los puntos que caigan dentro de sus límites en un único punto, que generalmente se corresponde con el centroide de los puntos contenidos en dicho voxel. Así, se consigue simplificar las nubes de puntos reduciendo la cantidad de datos a procesar sin perder significativamente la estructura espacial de la información original.

  Tras la aplicación de este filtro con un tamaño de voxel de 3 cm^3, se consiguió reducir el número de puntos a 36.217, quedando la nube como se muestra en la Figura 5.2.

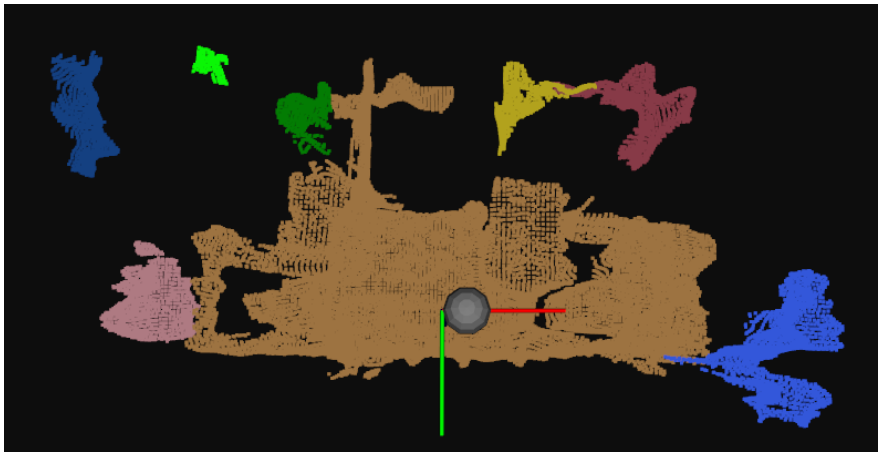

Figura 5.2: Representación de la nube de puntos detectada tras la aplicación del filtro voxel.

Como se puede observar, se encuentran numerosos objetos. Sin embargo, la detección no es del todo precisa, pues gran cantidad de los elementos que se perciben se están clasificando como parte del mismo objeto (representado en color marrón). La explicación de este problema y cómo solucionarlo se explica en el siguiente punto.

- Algoritmo RANSAC (Random Sample Consensus):

  En el contexto del procesamiento de nubes de puntos, el filtro RANSAC se utiliza para identificar y filtrar puntos que se ajustan a una determinada forma o modelo geométrico mientras ignora aquellos puntos que no se ajustan al modelo. En este caso, debido a que la captura de imágenes se va a realizar dentro de un caso de uso de conducción autónoma, el filtro RANSAC se ha utilizado para eliminar de la nube de puntos original aquellos puntos que correspondan al suelo.

  Para ello, se selecciona aleatoriamente un subconjunto de puntos para proponer un modelo inicial, calcular cuántos puntos de la nube se ajustan a este modelo dentro de un determinado umbral de tolerancia y repetir este proceso varias veces para encontrar el modelo que mejor se ajuste a la mayor cantidad de puntos de la nube.

  Tras su aplicación, se consiguió reducir la cantidad de puntos a 28.198, siendo la nube resultante la que se representa en la Figura 5.3.

  Como puede observarse, se ha conseguido eliminar el problema que sucedía en el caso anterior, en el que gran parte de los objetos detectados se clasificaban como parte del mismo obstáculo. Este problema ocurría porque los puntos que pertenecían al suelo interconectaban el resto de objetos de la captura entre sí, haciendo que se detectasen como un único objeto unido mediante el suelo. Sin embargo, al haber eliminado ahora estos puntos, los diferentes objetos que se capturaban en la imagen comienzan a ser detectados como obstáculos independientes.

- Filtro Cropbox

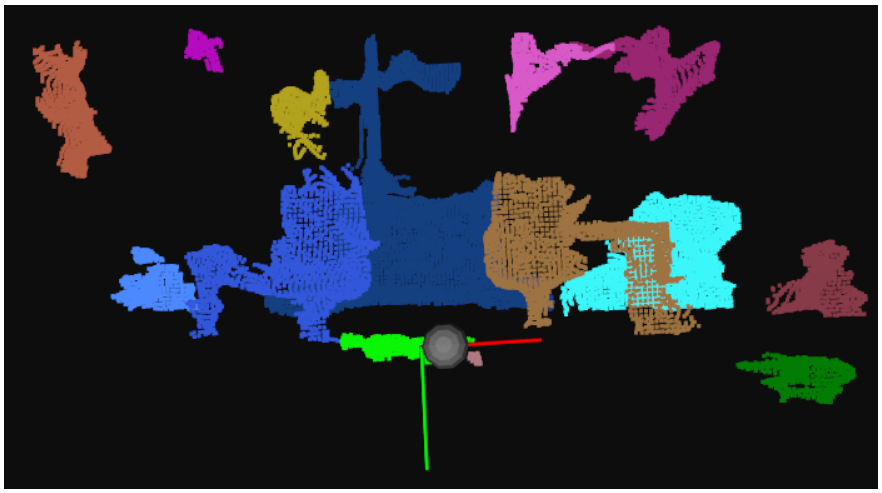

Figura 5.3: Representación de la nube de puntos detectada tras la aplicación del filtro RANSAC.

Esta técnica de filtrado se utiliza para seleccionar un subconjunto de puntos que se encuentran dentro de un volumen definido espacialmente. Este filtro permite definir un volumen tridimensional con forma rectangular o cúbica mediante límites máximos y mínimos en los ejes X, Y y Z, y posteriormente filtrar los puntos de la nube que se encuentren fuera de ese volumen.

Este filtro va a permitir evitar el procesamiento de puntos que se encuentran muy alejados del vehículo (en cualquiera de los tres ejes) y que, dadas las dimensiones del propio vehículo, no suponen ningún peligro a pesar de haber sido detectados por la cámara.

En este caso, se consigue reducir el número de puntos a 11.243 y se realiza una detección correcta, generando una versión final de la nube como la representada en la Figura 5.4.

5.2.3. Algoritmo de detección de obstáculos

Como se describía en el diagrama de la Figura 5.1, la nube de puntos filtrada constituye la entrada del algoritmo de detección de obstáculos. El algoritmo concreto que se ha escogido ha sido el DBSCAN (*Density-Based Spatial Clustering of Applications with Noise*), cuya implementación se ha extraído del repositorio [35].

A continuación, se describirá brevemente en qué consiste este algoritmo y se detallarán los aspectos más relevantes en cuanto a la implementación.

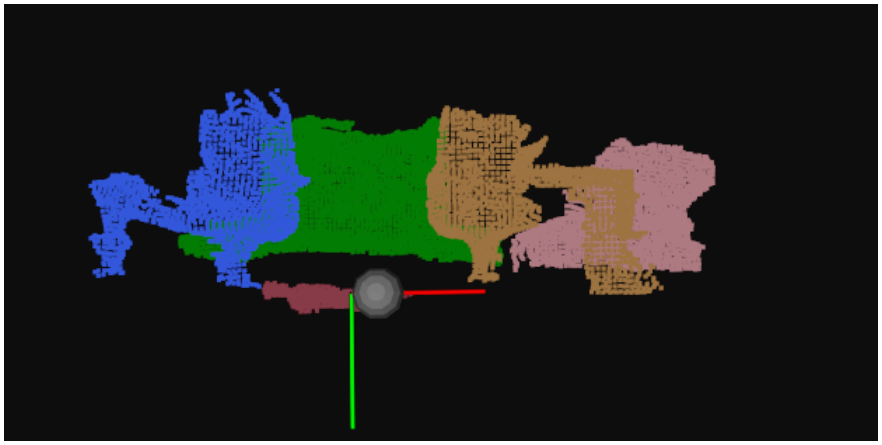

Figura 5.4: Representación de la nube de puntos detectada tras la aplicación del filtro Cropbox.

5.2.3.1. Descripción del algoritmo DBSCAN

El algoritmo DBSCAN se basa en el agrupamiento de datos con el objetivo de aislar e identificar estructuras dentro de una nube de puntos tridimensional, por lo que es ampliamente utilizado en el campo del aprendizaje automático y el análisis de datos.

Para identificar los diferentes obstáculos, el algoritmo se fundamenta en la idea de que aquellos puntos de la nube que estén densamente agrupados por encima de cierto valor umbral, se consideran puntos pertenecientes al mismo objeto y, por lo tanto, se agrupan dentro del mismo *cluster*. Por el contrario, aquellos puntos que estén situados en regiones de baja densidad, se considerarán ruido y no pertenecerán a ningún *cluster*, siendo eliminados de la nube de puntos.

De esta manera, el resultado final tras la aplicación del algoritmo DBSCAN consiste en un conjunto de *clusters*, cada uno de los cuales representa un objeto diferente de la imagen original capturada.

5.2.3.2. Detalles de implementación del algoritmo DBSCAN

Como se ha mencionado en la parte introductoria de la sección, el algoritmo de detección de obstáculos toma como entrada la nube de puntos generada a partir de la imagen capturada mediante la cámara RGBD y adecuadamente filtrada, y genera como salida el número de *clusters* que ha sido capaz de agrupar (correspondiéndose cada *cluster* con un obstáculo diferente) y el tiempo de procesamiento. Además, ofrece la posibilidad de visualizar la salida generada en la que se muestran todos los objetos detectados, agrupando los puntos que

pertenezcan a cada uno de ellos pintándolos del mismo color según se muestra en las imágenes del apartado 5.2.2.

En el repositorio utilizado [35], existen disponibles una implementación del algoritmo DBSCAN para ejecutar en la CPU y otra para ejecutar en la GPU (denominada G-DBSCAN y basada en el artículo [37]). Dadas las características del problema que se plantea, en el que se van a procesar miles de puntos para llevar a cabo la detección de obstáculos mediante un algoritmo de agrupamiento, se ha utilizado la implementación del algoritmo DBSCAN sobre la GPU. La preferencia por este tipo de implementación está justificada, entre otras razones, por el gran paralelismo que es capaz de ofrecer la GPU a la hora de procesar datos, lo cual será aprovechado a la hora de realizar operaciones sobre cada punto con un rendimiento global significativamente mayor. Además, analizando y comparando el tiempo de cómputo que supone la ejecución del algoritmo en cada una de sus dos implementaciones, se ha observado que el tiempo de ejecución en su versión para GPU llega a ser un orden de hasta 10 veces más rápido aproximadamente, aunque este valor varía en función del número de puntos por el que esté formada la nube sobre la que se aplique la detección.

Por último, en cuanto a la implementación G-DBSCAN, cabe destacar que se caracteriza por su simplicidad a la hora de indexar los datos, utilizando grafos para aplicar un enfoque basado en BFS (Breadth First Search) sobre los nodos de los puntos, lo que permite una paralelización efectiva del algoritmo y, en consecuencia, una aceleración significativa del tiempo de procesamiento.

En la Figura 5.5 se muestra un diagrama que representa el flujo de ejecución de la aplicación DBSCAN de detección de obstáculos y cómo hace uso de los recursos de procesamiento disponibles.

Como se observa en el diagrama, unas partes de la aplicación se ejecuta en la CPU (bloques de color blanco) y otras en la GPU (bloques de color verde). A continuación, se describirán las tareas realizadas en cada uno de los bloques representados en el diagrama. Sin embargo, cabe destacar que se han obviado de esta explicación muchas partes de la aplicación dedicadas a la copia de datos entre CPU y GPU y viceversa para que no resulte repetitivo. Como se refleja en la Figura 5.5, la copia de datos y de resultados entre CPU y GPU (o al revés) la lleva a cabo la GPU previa orden ejecutada en la CPU. Por su parte, antes de ejecutar cada uno de los *kernels* en la GPU, tendrá que ser lanzada la orden previamente también desde la CPU.

Por lo tanto, se describen a continuación las diferentes partes de la aplicación DBSCAN que resultan de interés para comprender el funcionamiento del algoritmo y la unidad de procesamiento en la que se ejecutan.

- En la CPU:

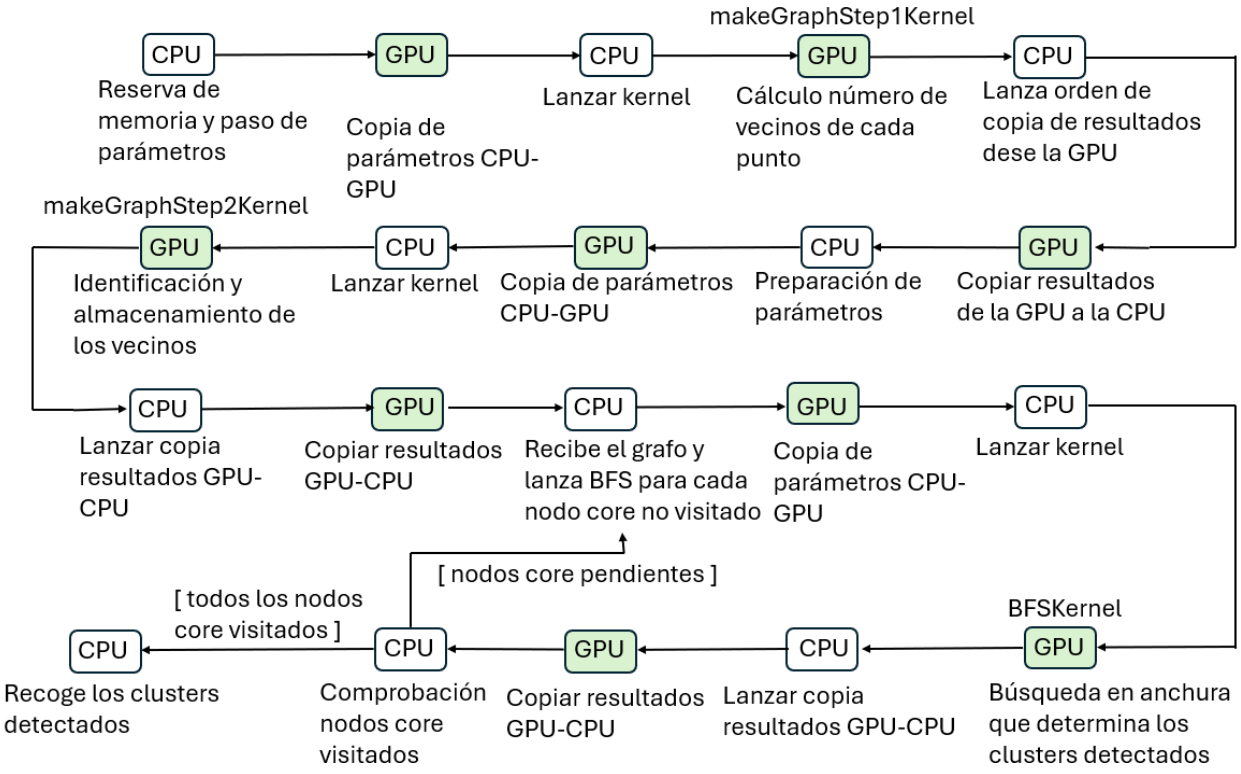

Figura 5.5: Diagrama del flujo de ejecución del algoritmo DBSCAN.

Reserva de un espacio de memoria en la GPU en el que almacenar los parámetros que el *kernel* necesita para realizar su función y transferencia de estos parámetros desde la memoria principal a este espacio de memoria de la GPU. Este proceso se lleva a cabo para la preparación tanto de la primera parte del grafo (función *makeGraphStep1Kernel*) como de la segunda (*makeGraphStep2Kernel*).

- En la GPU:

  La función *makeGraphStep1Kernel* realiza la primera fase del algoritmo DBSCAN, en la que se calcula el número de vecinos para cada punto de la nube tridimensional. Para ello, se itera sobre todos los puntos de la nube calculando, para cada uno de ellos, la distancia al resto de puntos dentro de un radio *eps* (pasado como parámetro), clasificando como vecinos aquellos nodos que estén a una distancia por debajo de un valor umbral.

  Además, también se encarga de determinar si cada uno de los puntos constituye un nodo *core* o no. Un nodo *core* es un punto de la nube cuyo número de vecinos es mayor que un determinado valor (*minPoints*, pasado como parámetro). Este tipo de puntos se utilizarán posteriormente como referencias iniciales a la hora de construir el *cluster* al que pertenecen.

  Por su parte, la función *makeGraphStep2Kernel* se encarga de elaborar una lista de

adyacencia para cada punto de la nube una vez calculados en el primer paso. Estas listas son esenciales para el algoritmo ya que almacenan explícitamente qué puntos concretos son vecinos de otro dado. Esto facilitará posteriormente la agrupación de los puntos en diferentes obstáculos.

- En la CPU:

  Una vez elaborado el grafo, se transfieren los nodos a la memoria principal de la CPU para, posteriormente, iterar sobre ellos verificando si han sido visitados y si son nodos de tipo *core*. Aquellos nodos *core* no visitados se utilizarán como referencia para la agrupación de nuevos *clusters*, lanzando posteriormente la función *BFSKernel* para realizar una búsqueda en anchura y expandir el *cluster* a partir del nodo *core* inicial utilizando las listas de adyacencia.

- En la GPU:

  Por último, la función *BFSKernel* realiza una búsqueda en anchura sobre el grafo construido en los pasos previos y representado mediante las listas de adyacencia. Con ello permite identificar conjuntos de puntos que cumplan con los criterios de densidad y que, por lo tanto, serán reconocidos como *clusters* independientes.

- En la CPU:

  Finalmente, se recogen los *clusters* calculados y se proporciona la posibilidad de visualizar la agrupación realizada mediante la librería VTK [36].

Como se había mencionado previamente en este apartado, también se observa que una vez se ha realizado el paso de parámetros desde la CPU, y la GPU ya se ha encargado de copiar en su memoria todos los datos necesarios para ejecutar un determinado *kernel*, el flujo de ejecución vuelve a pasar por la CPU para lanzar dicho *kernel*. En el caso de la GPU, esta se encarga también de realizar las copias de datos necesarias desde la CPU a su propia memoria y de los resultados de los *kernels* a la memoria de la CPU. Estas copias son bloqueantes en el flujo de ejecución y pueden observarse en el diagrama antes de las ejecuciones de cada uno de los *kernels*.

Finalmente, una vez descrito el algoritmo, puede comprobarse que estas funciones procesan grandes volúmenes de datos por lo que, para hacerlo de manera eficiente y paralela, se justifica el uso de la GPU para ejecutarlas. Su arquitectura paralela permitirá la ejecución simultánea de múltiples hilos, facilitando este tipo de operaciones computacionalmente costosas y reduciendo significativamente los tiempos de procesamiento.

5.3. Integración del caso de uso

El objetivo de esta parte del proyecto es integrar la aplicación de conducción autónoma dentro del planificador de manera que se ejecute mediante un hilo dentro de una partición del sistema según se indica en la Figura 5.6.

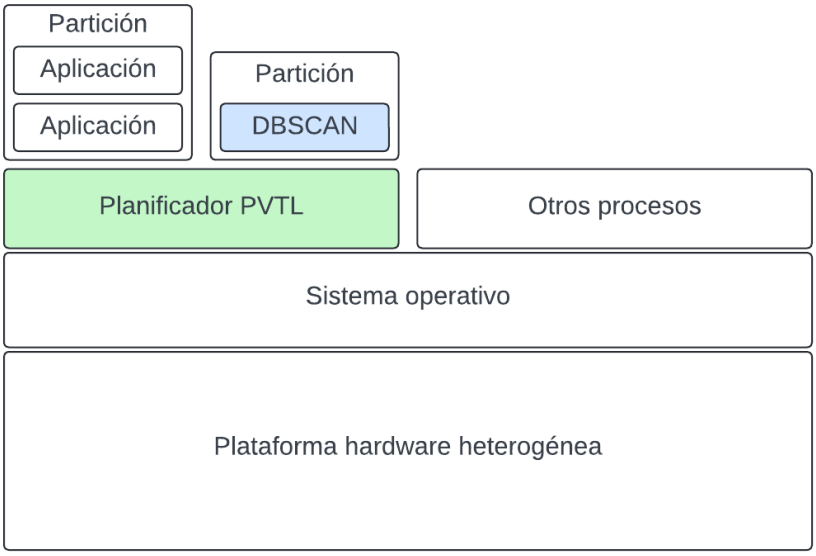

Figura 5.6: Arquitectura del sistema tras la integración del caso de uso.

Para ello, se ha definido una nueva función en el programa del planificador PVTL, la función *run_dbscan*, en la que se ha embebido el código de la aplicación de detección de obstáculos. De esta manera, ahora podrá lanzarse la función *run_dbscan* para que se ejecute por un hilo dentro de una partición del planificador. Sin embargo, todavía seguirá siendo necesario simular la ejecución de las tareas del resto de particiones.

Por su parte, el código de la aplicación DBSCAN no ha sido necesario modificarlo. En cuanto al planificador PVTL, las principales variaciones que ha sufrido respecto a la implementación descrita en el apartado 4.4, aparte de la integración de la aplicación de detección de obstáculos, han sido las siguientes:

- En la versión inicial del PVTL, se determinaba la configuración del particionado del sistema mediante el fichero de entrada *config.txt* descrito en el apartado 4.4.3.1. Ahora, este fichero se ha modificado de manera que se deba indicar también la partición en la que se vaya a ejecutar el caso de uso.
- En segundo lugar, se ha modificado también el proceso de construcción del escenario de ejecución para reservar la partición en la que se ha de ejecutar el caso de uso únicamente

para esta misión. Esto implica que no se deberá lanzar ninguna otra tarea simulada dentro de esta partición.

- Por último, se han tenido que realizar todas las adaptaciones necesarias para poder llamar a la aplicación DBSCAN como si fuese una función del programa del PVTL, como la creación de la estructura de atributos de la función, la definición del hilo que ejecuta la aplicación y todos sus atributos, etc.

Capítulo 6

Validación del entorno desarrollado

Este capítulo está destinado a describir las pruebas realizadas con las que se ha analizado el comportamiento del entorno de ejecución. Dentro de la explicación del proceso de validación, se describirá primero el planteamiento de los experimentos; a continuación, se detallará la estructura del escenario de ejecución diseñado y, finalmente, se mostrarán y analizarán los resultados obtenidos tras las pruebas.

6.1. Metodología de las pruebas realizadas

Para llevar a cabo el proceso de validación, se han diseñado una serie de pruebas con el objetivo de analizar cómo influye el tamaño de las diferentes particiones del sistema en el tiempo de respuesta de la ejecución del caso de uso.

Para ello, se ha diseñado un MAF (detallado en el siguiente apartado) en que que las particiones tienen una única ventana temporal, y cuya única variación entre una prueba y otra será el tamaño de esas ventanas temporales, conservando siempre la misma relación entre ellas. En cuanto a su estructura, se mantendrá fija para todas las pruebas: tanto el número de particiones como el orden en el que se ejecutan no variarán de una prueba a otra.

De esta manera, modificando el tamaño de las ventanas de las particiones entre una prueba y otra, se podrá observar el impacto de los tamaños escogidos sobre el tiempo de respuesta de la aplicación de detección de obstáculos y analizar su evolución a medida que se va modificando el tamaño de las particiones.

6.2. Escenario de ejecución para las pruebas realizadas

El escenario que se ha diseñado para analizar el comportamiento del entorno de ejecución se representa en el diagrama de la Figura 6.1. En él, se observa que se ha construido un MAF formado por tres particiones, las cuales mantendrán una relación entre ellas respecto a sus duraciones de 2:1:3. También puede apreciarse que la partición en la que se ejecutará el caso de uso es la partición 1, la cual ha sido reservada exclusivamente para ejecutar esa tarea.

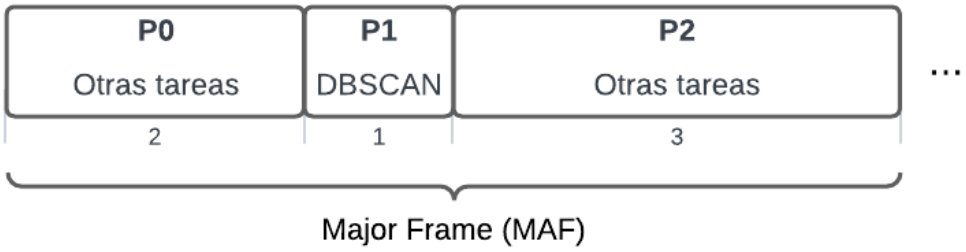

Figura 6.1: Estructura del MAF diseñado para las pruebas de validación.

Puede verse también que existen una serie de tareas aparte del caso de uso que serán ejecutadas en alguna de las otras dos particiones del sistema. Estas tareas han sido añadidas al escenario de ejecución para simular la ejecución de otros trabajos en las otras dos particiones. Sin embargo, los resultados de la ejecución de las mismas no son relevantes de cara al estudio que se quiere realizar del efecto del particionado sobre la ejecución del caso de uso, ya que el tiempo de expulsión que va a sufrir esta tarea siempre estará determinado por la duración de las otras particiones y nunca por la duración de las tareas que se ejecuten dentro de ellas.

6.3. Pruebas realizadas y resultados obtenidos

Las configuraciones que se han escogido para los tamaños de las particiones a la hora de realizar las pruebas para analizar el comportamiento del sistema se muestran en la tabla del Cuadro 6.1.

Una vez establecidas todas las configuraciones, se ha procedido a ejecutar cada una de las pruebas durante un tiempo estimado de 30 minutos, asignándole a la tarea correspondiente al caso de uso un periodo arbitrario que no fuese múltiplo de ninguno de los tamaños de los MAFs diseñadas. Concretamente, se le ha asignado un periodo de 19.75 segundos. De esta manera, se consigue introducir un pequeño factor aleatorio en cuanto a su activación respecto al inicio del MAF, haciendo que vaya variando a medida que avanza la ejecución del sistema. Esto provocará que el instante de activación del caso de uso no sea fijo respecto

	Partición	Tiempo (s)
Prueba 1	P0	4
	P1	2
	P2	6
Prueba 2	P0	2
	P1	1
	P2	3
Prueba 3	P0	0.8
	P1	0.4
	P2	1.2
Prueba 4	P0	0.4
	P1	0.2
	P2	0.6
Prueba 5	P0	0.2
	P1	0.1
	P2	0.3
Prueba 6	P0	0.1
	P1	0.05
	P2	0.15

Cuadro 6.1: Configuración de las particiones para las pruebas de validación.

al inicio del MAF y proporcione unos resultados respecto a sus tiempos de respuesta más variados.

Tras realizar las ejecuciones de todas las pruebas y haber recogido los datos de los tiempos de respuesta de la tarea que ejecuta el caso de uso, los resultados obtenidos han sido los que se muestran en el diagrama de la Figura 6.2.

Como se describía en el apartado 4.5.1 en el que se analizaban los tiempos de los cambios de contexto, el diagrama de la Figura 6.2 es un diagrama de cajas que muestra la distribución de los tiempos de respuesta obtenidos en cada prueba y que sirve para visualizar la dispersión de los resultados. Como novedad respecto al diagrama del apartado 4.5.1, en este gráfico aparecen unos puntos por fuera de las cajas que son considerados valores atípicos, los cuales sería importante analizar al tratarse el caso de uso de una aplicación crítica de tiempo real.

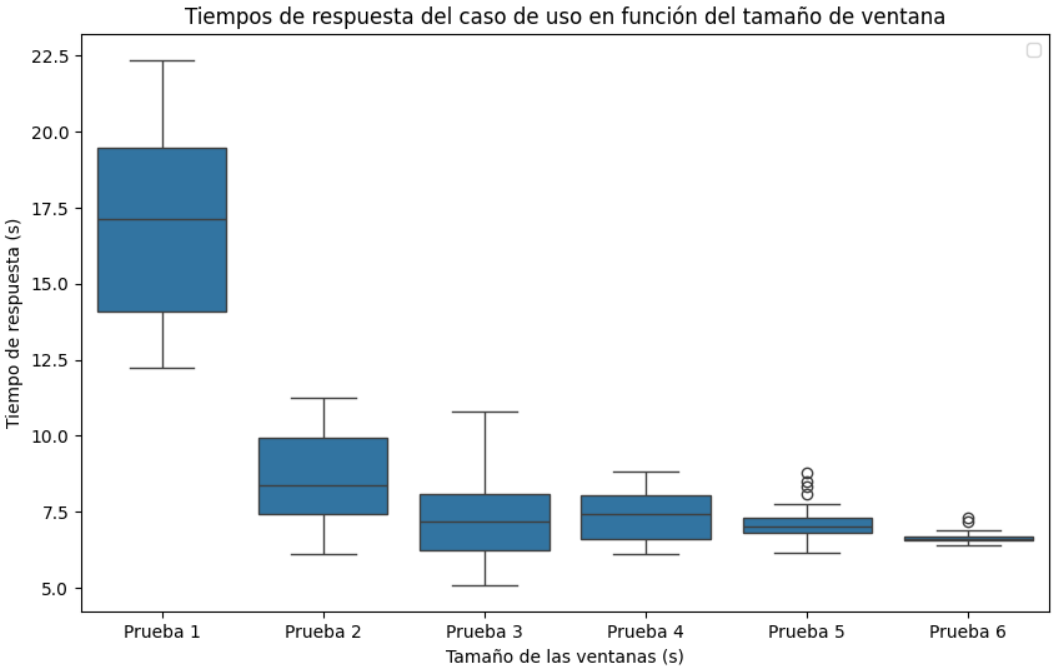

Figura 6.2: Resultados obtenidos al ejecutar los experimentos diseñados para las pruebas de validación.

En cuanto a los resultados obtenidos, en primer lugar cabe destacar cómo la variabilidad de los valores de los tiempos de respuesta registrados disminuye considerablemente a medida que se reduce el tamaño de las particiones. Se observa que, en la prueba realizada con el tamaño de las particiones más grande, la prueba 1, los valores del tiempo de respuesta obtenido oscilan entre los 12.5 y los 22.5 segundos aproximadamente, lo cual supone una variabilidad muy elevada (de unos 10 segundos). Sin embargo, a medida que el tamaño de las ventanas se va reduciendo, la dispersión en los tiempos de respuesta es cada vez menor, hasta llegar a una oscilación entre los 6.4 y los 7.3 segundos en la prueba 6, la de las ventanas más pequeñas.

Por otro lado, se puede apreciar también que el tiempo de respuesta obtenido para el caso de uso también disminuye significativamente a medida que se reduce el tamaño de las particiones. Como se observa, la mediana calculada para cada una de las pruebas evoluciona desde los 17.5 segundos aproximadamente con las ventanas temporales más grandes hasta los 7 segundos aproximadamente con las ventanas temporales más pequeñas.

Estos últimos resultados son coherente con los cálculos que se han realizado para analizar el tiempo de respuesta del caso de uso cuando se ejecuta en un entorno sin particiones. En estas pruebas en las que no sufría expulsiones, se obtenía un tiempo medio de respuesta de 1.2 segundos aproximadamente. Por tanto, unos tiempos de respuesta en un entorno

particionado de entre 7 y 8 segundos resulta razonable si se tiene en cuenta que la partición a la que pertenece (partición 1) obtiene una utilización de la CPU de 1/6.

No obstante, la evolución en los tiempos de respuesta medidos no es lineal, si no que se estabiliza (e incluso retrocede) una vez se llega a reducir el tamaño de las ventanas por debajo de cierto valor umbral. Como se ve en los resultados obtenidos en la prueba 4, a pesar de que sí se consigue disminuir la variabilidad de los tiempos de respuesta respecto a la prueba anterior, la mediana de los resultados obtenidos aumenta. Por tanto, aunque se sigue reduciendo la dispersión, los tiempos de respuesta se mantienen más o menos constantes.

Finalmente, se identifican algunos valores atípicos en las pruebas con las ventanas más pequeñas. A pesar de que estos resultados se obtienen con una frecuencia baja, son datos que, como se ha dicho previamente, se deberían tener en cuenta al tratarse de un sistema crítico de tiempo real.

En definitiva, se puede confirmar que un tamaño pequeño para las ventanas contribuye a la estabilidad del sistema, minimizando la variabilidad y, por lo tanto, mejorando la predictibilidad. Esto se explica porque cuanto menor sea el tamaño de las ventanas, menor será el tiempo de expulsión sufrido por la tarea encargada de ejecutar el caso de uso hasta que vuelva a ejecutarse en el siguiente MAF. Sin embargo, reducir el tamaño de las ventanas por debajo de un determinado valor umbral puede ser contraproducente, como también se concluyó en el artículo de investigación [38], ya que el tiempo de los cambios de contexto entre particiones puede aumentar en proporción respecto a la duración de cada una de las ventanas, reduciendo el tiempo efectivo durante el que se ejecutan tareas dentro de las particiones y provocando un aumento en los tiempos de respuesta.

Capítulo 7

Conclusiones y trabajo futuro

En este último capítulo, se procederá a sintetizar los logros obtenidos y las conclusiones extraídas tras la finalización del trabajo. Posteriormente, se propondrán una serie de trabajos futuros para profundizar en algunas líneas de trabajo que se han quedado abiertas y que ayudarían a conseguir una versión del entorno más completa.

7.1. Objetivos conseguidos

Dados los objetivos descritos en el apartado 1.2, el desarrollo de este trabajo ha conseguido llevar a cabo las siguientes aportaciones:

- Estado del arte de los hipervisores disponibles para plataformas con GPUs.

  Se ha realizado un estudio del estado del arte de los hipervisores disponibles para plataformas heterogéneas con una GPU integrada. Este estudio ha servido para investigar sobre las prestaciones que ofrecen los hipervisores a día de hoy para la ejecución de sistemas críticos de tiempo real sobre plataformas heterogéneas con una GPU integrada.

  Gracias a esta tarea de exploración, se ha visto que este campo es un tema de actualidad que despierta un gran interés y sobre el que todavía se está trabajando en el mundo de la investigación para ofrecer soluciones.

- Desarrollo del planificador con particionado temporal.

  Se ha conseguido desarrollar un particionado temporal con una planificación *time-table driven* que permite controlar la activación de las diferentes tareas a ejecutar dentro del sistema garantizando un aislamiento entre las diferentes particiones que se ejecuten.

De esta manera, se evita que se den situaciones en las que varias tareas estén haciendo uso de la GPU concurrentemente. Para ello, este planificador se encarga de gestionar grupos de hilos aprovechándose de la política de planificación mediante prioridades fijas de Linux.

- Implementación de un caso de uso sobre una plataforma con una GPU integrada e integración con el planificador.

  Se ha conseguido desarrollar un caso de uso en el que se ejecuta un algoritmo de detección de obstáculos sobre una imagen capturada mediante una cámara de profundidad.

  Para ello, se desarrolló primero una parte en la que se utilizaba la librería LibRealSense de Intel para realizar una captura mediante la cámara equipada a bordo del F1Tenth. Posteriormente, se adaptó una implementación del algoritmo DBSCAN sobre la GPU para llevar a cabo una detección de obstáculos sobre la nube de puntos generada a partir de la imagen tomada por la cámara, llegando a integrar esta segunda parte en el planificador desarrollado previamente.

  De esta manera, se conseguía ejecutar este algoritmo de detección de obstáculos, que podría formar parte de una aplicación real que se ejecute en un vehículo autónomo, dentro de una partición del planificador respetando el aislamiento y los momentos de activación predefinidos en la tabla de la planificación *time-table driven*.

- Análisis y validación de la plataforma hardware desarrollada.

  Por último, una vez finalizada la integración, se han llevado a cabo una serie de pruebas que han permitido comprobar el correcto funcionamiento del particionado temporal desarrollado y analizar el efecto del propio particionado sobre los tiempos de respuesta de la tarea encargada de ejecutar el caso de uso. Además, se han podido utilizar los resultados logrados para extraer una serie de conclusiones acerca del efecto del tamaño de las particiones sobre los tiempos de respuesta obtenidos.

  Principalmente, lo que se ha observado es que la variabilidad de los tiempos de respuesta obtenidos se reduce a medida que disminuye el tamaño de las ventanas de las particiones. Además, también se reducen los valores de los tiempos de respuesta, aunque no de forma lineal, ya que a partir de cierto valor umbral para los tamaños de las ventanas, tienden a estabilizarse e incluso llegan a empeorar ligeramente en algún caso.

7.2. Trabajos futuros

Durante el desarrollo de este proyecto, han ido apareciendo una serie de líneas de trabajo sobre las que sería interesante profundizar y que se han dejado abiertas por quedar fuera del alcance de este proyecto. Algunas de estas líneas de trabajo futuras son las siguientes:

- Desarrollar y poner en práctica la gestión de las activaciones sobre diferentes tareas que utilicen la GPU.

  Como se ha mencionado en el apartado anterior, uno de los objetivos del particionado temporal era otorgar al usuario la capacidad de poder gestionar la activación de las tareas de su sistema de manera que se pudiesen evitar accesos concurrentes a la GPU por parte de varias tareas que hiciesen uso de ella. Para poder poner en práctica esta gestión, queda pendiente una línea de trabajo en la que se integren en el planificador más aplicaciones que utilicen la GPU para que pueda llevarse a cabo esta gestión y analizar su comportamiento. Esto permitiría realizar un estudio sobre su verdadero impacto sobre los tiempos de respuesta de estas tareas al haber reducido las interferencias en la GPU.

  Aprovechando el estudio que requeriría este trabajo sobre las interferencias en la GPU, podría desarrollarse también la versión del propio planificador para que dejase de funcionar de manera monolítica y permitiese introducir con facilidad tareas en el sistema correspondientes a otros procesos.

- Modelado MAST [39] del algoritmo G-DBSCAN de detección de obstáculos.

  Durante el desarrollo del proyecto se ha realizado un diagrama que representa el flujo de ejecución del algoritmo G-DBSCAN y qué partes son ejecutadas por la CPU y cuáles por la GPU. Sin embargo, no se ha realizado un modelado de la aplicación sobre el particionado desarrollado que permita calcular de manera analítica los tiempos de respuesta obtenidos para la ejecución, por un lado, de cada una de las partes del algoritmo y, por otro lado, para la ejecución global.

  El modelado de este algoritmo facilitaría el estudio de la planificabilidad de un sistema real en el que se ejecutase la aplicación G-DBSCAN utilizada en este proyecto.

- Análisis de técnicas de optimización para la configuración del sistema.

  El capítulo anterior de validación (capítulo 7) se planteó como una manera de observar el comportamiento del sistema dadas diferentes configuraciones para las particiones del sistema. Sin embargo, se podría ampliar el trabajo en esa línea e investigar diferentes técnicas que optimicen los valores a asignar al tamaño de las diferentes particiones dado un determinado escenario de ejecución. En la línea del trabajo [40], esto implicaría estudiar ese balance óptimo entre el tamaño de las ventanas y el tiempo de respuesta de las tareas ejecutadas ya que, como se ha visto, reducir el tamaño de las ventanas a partir de cierto valor umbral puede llegar a ser contraproducente. Además, tampoco sería ventajoso reducir el tamaño de las ventanas de manera arbitraria ya que habría que tener en cuenta a la hora de asignar el tamaño de ventana otras aplicaciones que utilicen la GPU para evitar el acceso concurrente.

Bibliografía

[1] F1TENTH Project. Official F1TENTH Website: The Platform for Autonomous Racing. `https://f1tenth.org/`. Accesed July 2024.

[2] NVIDIA. Jetson Orin NX Documentation. `https://developer.nvidia.com/blog/delivering-server-class-performance-at-the-edge-with-nvidia-jetson-orin/`, 2024. Accessed June 2024.

[3] Matteo Andreozzi, Giacomo Gabrielli, Balaji Venu, and Giacomo Travaglini. Industrial Challenge 2022: A High-Performance Real-Time Case Study on Arm. *Leibniz International Proceedings in Informatics, LIPIcs*, 231, 7 2022.

[4] Frédéric Boniol and Sibin Mohan. IEEE RTSS 2022 Industry Challenge http://2022.rtss.org/industry-session.

[5] Iosu Gomez, Unai Díaz de Cerio, Jorge Parra, Juan M. Rivas, and J. Javier Gutiérrez. Using GPUs in Real-Time Applications - A Review of Techniques for Analyzing and Optimizing the Timing Parameters. *RIAI - Revista Iberoamericana de Automatica e Informatica Industrial*, 21:1–16, 2024.

[6] Roberto Cavicchioli, Nicola Capodieci, and Marko Bertogna. Memory interference characterization between CPU cores and integrated GPUs in mixed-criticality platforms. In *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–10, 2017.

[7] Guin Gilman and Robert J. Walls. Characterizing concurrency mechanisms for NVIDIA GPUs under deep learning workloads. *Performance Evaluation*, 151:102234, 2021.

[8] Airlines Electronic Engineering Committee. Avionics Application Software Standard Interface (ARINC-653), 2010.

[9] Jon Perez-Cerrolaza, Jaume Abella, leonidas kosmidis, Alejandro Calderon, Francisco Cazorla, and Jose Luis Flores. GPU Devices for Safety-Critical Systems: A Survey. *ACM Computing Surveys*, 55, 07 2022.

[10] Edoardo Cittadini, Mauro Marinoni, Alessandro Biondi, Giorgiomaria Cicero, and Giorgio Buttazzo. Supporting AI-powered real-time cyber-physical systems on heterogeneous platforms via hypervisor technology. *Real-Time Systems*, 59:1–27, 07 2023.

[11] Alessandro Biondi, Daniel Casini, Giorgiomaria Cicero, Niccolo Borgioli, Giorgio Buttazzo, Gaetano Patti, Luca Leonardi, Lucia Lo Bello, Marco Solieri, Paolo Burgio, Ignacio Sanudo Olmedo, Angelo Ruocco, Luca Palazzi, Marko Bertogna, Alessandro Cilardo, Nicola Mazzocca, and Antonino Mazzeo. SPHERE: A Multi-SoC Architecture for Next-Generation Cyber-Physical Systems Based on Heterogeneous Platforms. *IEEE Access*, 9:75446–75459, 2021.

[12] Siemens. Jailhouse Hypervisor Respository. `https://github.com/siemens/jailhouse`. Accessed June 2024.

[13] SYSGO. PikeOS - Safety and Security Platform. `https://www.sysgo.com/pikeos`. Accessed June 2024.

[14] SYSGO. PikeOS Board Support Package (BSP) List. `https://www.sysgo.com/pikeos-bsp`. Accessed August 2024.

[15] Universitat Politècnica de València. XtratuM Hypervisor Official Website. `https://aplicat.upv.es/exploraupv/ficha-tecnologia/patente software/15232`. Accessed June 2024.

[16] Leonidas Kosmidis, Marc Solé, Ivan Rodriguez, Jannis Wolf, and Matina Maria Trompouki. The METASAT Hardware Platform: A High-Performance Multicore, AI SIMD and GPU RISC-V Platform for On-board Processing. In *2023 European Data Handling & Data Processing Conference (EDHPC)*, pages 1–6, 2023.

[17] Xen Project Official Website. Xen Hypervisor. `https://xenproject.org/`. Accessed June 2024.

[18] Xen Project. RT-Xen: Real-Time Virtualization in Xen. `https://xenproject.org/2013/11/27/rt-xen-real-time-virtualization-in-xen/`. Accessed June 2024.

[19] Steven H. VanderLeest and Dagan White. MPSoC hypervisor: The safe & secure future of avionics. In *2015 IEEE/AIAA 34th Digital Avionics Systems Conference (DASC)*, pages 6B5–1–6B5–14, 2015.

[20] AMD. Zynq UltraScale+ Overview. `https://docs.amd.com/v/u/en-US/ds891-zynq-ultrascale-plus-overview`. Accessed September 2024.

[21] ACCELERAT Project. CLARE-Hypervisor Official Website: Climate Risk Early Warning System. `https://accelerat.eu/clare/`. Accessed June 2024.

[22] Edoardo Cittadini, Mauro Marinoni, Alessandro Biondi, Giorgiomaria Cicero, and Giorgio Buttazzo. Supporting AI-powered real-time cyber-physical systems on heterogeneous platforms via hypervisor technology. *Real-Time Systems*, 59:609–635, 12 2023.

[23] Red Hat. KVM Official Website. `https://www.redhat.com/es/topics/virtualization/what-is-KVM`. Accessed June 2024.

[24] Linux Foundation. KVM and Preempt-RT. `https://wiki.linuxfoundation.org/media/realtime/events/rt-summit2016/kvm_rik-van-riel.pdf`. Accessed June 2024.

[25] Lattice0. KVM implementation on Jetson Nano - Respository. `https://github.com/lattice0/jetson_nano_kvm`. Accessed August 2024.

[26] B-Man. KVM implementation on Xavier - Respository. `https://github.com/b-man/Xavier-KVM`. Accessed August 2024.

[27] Virtual Open Systems. AGL KVM on Renesas R-Car M3. `http://www.virtualopensystems.com/en/solutions/guides/agl-kvm-rcar-m3/`. Accessed September 2024.

[28] NVIDIA. DriveOS Development Guide for NVIDIA DRIVE Platforms. `https://docs.nvidia.com/drive/archive/drive_os_-5.1.15.0L/drive-os/index.html`. Accessed September 2024.

[29] AUTOSAR Consortium. AUTOSAR - Automotive Open System Architecture. `https://www.autosar.org/`, 2024. Accessed September 2024.

[30] Santiago Lozano, Tamara Lugo, and Jesus Carretero. A Comprehensive Survey on the Use of Hypervisors in Safety-Critical Systems. *IEEE Access*, 11:36244–36263, 2023.

[31] DataScientest. DBSCAN: Un algoritmo de clustering para Machine Learning. `https://datascientest.com/es/machine-learning-clustering-dbscan`, 2024. Accessed September 2024.

[32] Intel RealSense. LibRealSense Repository. `https://github.com/IntelRealSense/librealsense`, 2024. Accessed July 2024.

[33] Point Cloud Library. PCD File Format Documentation. `https://pointclouds.org/documentation/tutorials/pcd_file_format.html`, 2024. Accessed July 2024.

[34] Point Cloud Library. Point Cloud Library (PCL) Official Website. `https://pointclouds.org/`, 2024. Accessed July 2024.

[35] xmba15. Generic DBSCAN Repository. `https://github.com/xmba15/generic_-dbscan`, 2024. Accessed July 2024.

[36] VTK. Visualization Toolkit (VTK) Documentation. `https://vtk.org/`, 2024. Accessed July 2024.

[37] Guilherme Andrade, Gabriel Ramos, Daniel Madeira, Rafael Sachetto, Renato Ferreira, and Leonardo Rocha. G-DBSCAN: A GPU Accelerated Algorithm for Density-based Clustering. *Procedia Computer Science*, 18:369–378, 2013. 2013 International Conference on Computational Science.

[38] Andoni Amurrio, J. Javier Gutiérrez, Mario Aldea, and Ekain Azketa. Partition window assignment in hierarchically scheduled time-partitioned distributed real-time systems with multipath flows. *Journal of Systems Architecture*, 130:102671, 2022.

[39] Universidad de Cantabria Grupo ISTR. MAST: Modeling and Analysis Suite for Real-Time Applications. `https://mast.unican.es/`, 2024. Accessed September 2024.

[40] Andoni Amurrio, José Javier Gutiérrez García, Mario Aldea Rivas, and Ekain Azketa. Optimization in Heterogeneous Distributed Real-Time Systems based on Partitioning. In *43rd IEEE Real-Time Systems Symposium (RTSS) 2022, Industry Challenge, Houston (USA), December 2022.*, 12 2022.

Apéndice A

Mensajes de *log* durante la ejecución de las pruebas de validación del planificador

```
Inicio ejecucion

Se activa planificador -- 0.000

En ejecucion: particion 0
Comienza a ejecutarse: tarea 0 (ex. 0.100) -- 0.000
Fin tarea 0 -- 0.100
Comienza a ejecutarse: tarea 1 (ex. 0.025) -- 0.100
Fin tarea 1 -- 0.125
Inicio dummy -- 0.125

Se activa planificador -- 0.150

En ejecucion: particion 1
Comienza a ejecutarse: tarea 2 (ex. 0.300) -- 0.150
Fin tarea 2 -- 0.350
Comienza a ejecutarse: tarea 3 (0.150) -- 0.350

Se activa planificador -- 0.450

En ejecucion: particion 2
Comienza a ejecutarse: tarea 4 (ex. 0.075) -- 0.450
Fin tarea 4 -- 0.525
Comienza a ejecutarse: tarea 5 (ex. 0.100) -- 0.525
Fin tarea 5 -- 0.625
```

```
Comienza a ejecutarse: tarea 6 (ex. 0.025) -- 0.625
Fin tarea 6 -- 0.650
Inicio dummy -- 0.650

Se activa planificador -- 0.700

En ejecucion: particion 3
Comienza a ejecutarse: tarea 7 (ex. 0.050) --0.700
Fin tarea 7 -- 0.750
Comienza a ejecutarse: tarea 8 (ex. 0.175) -- 0.750
Fin tarea 8 -- 0.925
Comienza a ejecutarse: tarea 9 (ex. 0.100) -- 0.925

Se activa planificador -- 1.000

En ejecucion: particion 0
Comienza a ejecutarse: tarea 0 (ex. 0.100) -- 1.000
Fin tarea 0 -- 1.100
Inicio dummy -- 1.100

Se activa planificador -- 1.150

En ejecucion: particion 1
Fin tarea 3 -- 1.200
Inicio dummy -- 1.200

Se activa planificador -- 1.450

En ejecucion: particion 2
Inicio dummy -- 1.450

Se activa planificador -- 1.70

En ejecucion: particion 3
Fin tarea 9 -- 1.725
Inicio dummy -- 1.725

Se activa planificador -- 2.00
```

Listing A.1: Mensajes de Ejecución