UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Control de la ejecución de sistemas particionados de criticidad mixta

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Terrizzano Cinosi, Tomás
Cotutor: Balbastre Betoret, Patricia
Tutor: Crespo Lorente, Alfons

Curso 2021-2022

Resumen

Un sistema de criticidad mixta, compuesto por un procesador multinúcleo como unidad central de procesamiento, requiere de un controlador para asegurar la correcta ejecución de los componentes *software* de criticidad alta (los programas prioritarios). En este trabajo se muestra la interferencia que puede generar la ejecución en paralelo de varios programas informáticos sobre un mismo procesador multinúcleo (mostrando así la necesidad de dicho controlador) y una implementación efectiva del controlador de la ejecución a nivel de programa (que no de sistema operativo). La estrategia del controlador consiste en comprobar el avance de los programas de criticidad alta, mediante la unidad de monitorización de rendimiento, para posponer, si es necesario, la ejecución de los programas en los otros núcleos, cediendo de esta forma todos los recursos (especialmente el bus de datos) al programa de mayor prioridad. El escenario de trabajo se compone de la placa de desarrollo Cora Z7, que cuenta con un procesador de doble núcleo ARM Cortex A9 con unidad de monitorización de rendimiento; del hipervisor Xtratum, que ofrece un planificador temporal cíclico para poder programar escenarios de ejecución en los que el controlador propuesto entra en acción; y de un programa que multiplica matrices que sirve como carga de trabajo.

Palabras clave: Sistemas de criticidad mixta, XtratuM, Procesador multinúcleo, Controlador de la ejecución, Unidad de monitorización de rendimiento

Resum

Un sistema de criticitat mixta, composat per un processador multinucli com a unitat central de processament, requereix de un controlador per a asegurar la correcta execució dels components *software* de criticitat alta (els programes prioritaris). En aquest treball es mostra l'interferència que pot generar l'execució en parallel de diversos programes informàtics per damunt d'un mateix processador multinucli (mostrant així la necessitat de dit controlador) i una implementació efectiva del controlador de la execució a nivell de programa (que no de sistema operatiu). La estratègia del controlador consisteix en comprobar l'avanç dels programes de criticitat alta, mitjançant la unitat de monitorització del rendiment, per a posposar, si es necessari, la execució dels programes en els altres nuclis, cedint d'aquesta forma tots els recursos (especialment el bus de dades) al programa en major prioritat. L'escenari de treball es composa de la placa de desenvolupament Cora Z7, que compta amb un processador en dos nuclis i amb unitat de monitorització del rendiment; del hipervisor Xtratum, que ofereix un planificador temporal cíclic per a poder programar els escenaris d'experimentació en els que el controlador implementat entra en acció; i un programa que multiplica matrius que serveix com a carrega de treball.

Paraules clau: Sistemes de criticitat mixta, Xtratum, Processador multinucli, Controlador de l'execució, Unitat de monitorització del rendiment

Abstract

An execution controller is required in multicore mixed criticallity systems in order to accomplish execution deadlines of the highest criticallity software. Parallel software execution interference and an efective execution controller are both exposed in this work, showing the need of a controller and an implementation (at program level, not operating system level) of it. Constantly checking the progress of a high criticallity program, through the performance monitoring unit, and stopping the excecution of the programs executed in other cores, if it is needed, is the strategy of the controller for giving the high criticallity program all the resources of the processor for accomplishing execution deadlines. The experimenting scenario is composed of: a Cora Z7 development platform, with a dual-core Cortex A9 with performance monitoring unit, the Xtratum hypervisor, with a cyclic sheduller for programming the experiments and a matrix multiplier as a work-load.

Key words: Mixed criticallity systems, Xtratum, Multicore processor, Execution controller, Performance monitoring unit

Índice general

Índice de figuras

Índice de tablas

CAPÍTULO 1
Introducción

Siempre ha habido una tendencia en la industria de los sistemas embebidos a reducir costes, tanto informáticos (menor cantidad de memória, menor capacidad de computo, etc.) como energéticos (circuitos de bajo consumo, refrigeración pasiva, etc.) y de producción (fabricación en serie, optimización del diseño, etc.).

A esta tendencia se le ha sumado otra tendencia, en la fabricación de procesadores y de memorias, que es la de reducir o mantener precios mientras la productividad de los mismos incrementa constantemente.

Esta suma de tendencias forma gran parte de las condiciones de posibilidad de desarrollo e investigación de los sistemas de criticidad mixta. Estos sistemas se llaman así porque albergan en una misma plataforma *hardware* varios componentes *software* (procedimientos, funciones, programas, *scripts*, sistemas operativos, etc.), al mismo tiempo que estos componentes tienen diferentes niveles de importancia, conocidos como criticidades. Estos se pueden encontrar en la industria automovilística, aeroespacial o ferroviaria, por ejemplo, en los equipos informáticos con los que cuentan sus productos.

Los sistemas de criticidad mixta que utilizan procesadores multinúcleo tienen dificultades añadidas, ya que hay situaciones en las que en el mismo procesador se ejecutan programas de distinta criticidad. Una de ellas es la compartición de memoria, que debe ser gestionada por un sistema operativo o desde la fase de diseño del sistema. Otra, central en este trabajo, es la compartición del procesador. Concretamente, cuando este solo cuenta con un bus de acceso a datos, los componentes *software* se generan esperas los unos a los otros para poder acceder a memoria, lo que entorpece el determinismo del sistema.

Este tipo de problemáticas son estudiadas en el Instituto Universitario de Automática e Informática Industrial [1], que es una institución ubicada en la Universidad Politécnica de Valencia (UPV). Es una de las estructuras de investigación más importantes de la UPV, centrada en estrechar la relación entre la investigación académica (I+D+i) y las empresas e instituciones externas al ámbito universitario. Con sede en la Ciudad Politécnica de la Innovación, la integran más de 100 investigadores, muchos de ellos doctores de reconocido prestigio en las áreas de ingeniería industrial, informática, telecomunicaciones, electrónica y automática. Ahí se dedican al estudio de diferentes aspectos de la ingeniería industrial: sistemas de tiempo real, control de procesos, robótica, visión por computador, etc.

Figura 1.1: Logotipo del Instituto

Es en el contexto de una beca de colaboración con dicho instituto (y con la empresa FentISS [2]) surge la idea de este trabajo, que es la de experimentar con un sistema de criticidad mixta ejecutado sobre un procesador multinúcleo (con un solo bus de datos), observando las esperas generadas entre los componentes *software*, y la de diseñar un controlador que asegure que las esperas generadas no impidan la ejecución dentro de plazo de los componentes *software* más prioritarios.

Figura 1.2: Logotipo de FentISS

El hipervisor Xtratum se ha utilizado como herramienta principal en este trabajo, ya que ofrece un riguroso aislamiento espacial entre particiones, entiendiendo una partición como un componente *software* y el aislamiento espacial como una correcta gestión de la memoria, sin interferencias entre particiones, y también ofrece un planificador cíclico

1.1 Motivación

Debido a la gran potencia de cómputo de los procesadores actuales y el asequible coste de los mismos, son varios sectores industriales los que, con la intención de cumplir con exigencias cada vez mayores en cuestiones de optimización en general (coste de producción, peso, generación de calor, consumo energético, complejidad, etc.), apuestan por integrar lo que antes llevaban a cabo diferentes equipos informáticos separados (a lo sumo conectados para comunicarse entre ellos) en una misma plataforma *hardware* capaz de ofrecer todos los servicios demandados. Por ejemplo, la aviónica (electrónica de aeronaves) antes se componía de un sistema de radio para las comunicaciones, un sistema de localización por radar y un sistema de control automático de la aeronave, además de muchas otras funcionalidades. Hoy en dia estas necesidades se cubren con un sistema informático compuesto por sensores y actuadores por una parte, y por otra parte un computador y *software* (comunicadas ambas partes por un bus de datos).

La integración de diferentes tareas en una misma plataforma *hardware* genera nuevas problemáticas que deben ser resueltas para asegurar el correcto funcionamiento del sistema. Estas problemáticas tienen que ver con la compartición de los recursos *hardware*: para asegurar el comportamiento esperado de las diferentes funcionalidades que integran el sistema se debe asegurar que unas no interferiran en la ejecución de las otras.

Es esta problemática la que se pretende abordar en este trabajo. Mediante una monitorización del rendimiento de las diferentes tareas de sistema se sabe cuando se debe intervenir para asegurar la correcta ejecución de los procesos importantes

1.2 Objectivos

Los objetivos de este trabajo son tres:

- Modelar las diferentes cargas de trabajo a las que se va a enfrentar el sistema de experimentación para poder obtener información realista y valiosa de los experimentos.

- Plantear escenarios de ejecución de tareas que permitan observar las interferencias que se generan al ejecutar, paralelamente, diferentes tareas en un mismo procesador multinúcleo (en este caso un ARM Cortex A9 con dos núcleos de procesamiento).

- Diseñar un controlador que asegure la correcta ejecución de las tareas importantes (más adelante se define el criterio de importancia) en los casos en los que la interferencia entre tareas lo impida.

1.3 Estructura de la memória

Esta memória comienza exponiendo la actualidad de la mayoría de tecnologías que envuelven este trabajo. En el segundo capítulo, titulado "El estado del arte", se habla se citan libros y artículos relevantes que hablan sobre los sistemas embebidos, los sistemas de tiempo real, los sistemas de criticidad mixta y los sistemas particionados (concretandose en la aviónica modular integrada) y dos de las arquitecturas de procesadores más presentes a dia de hoy (la x86 y la ARM).

A continuación, en el tercer capítulo, titulado "Experimentación", se muestra:

- el **modelo** con el que se parámetrizan las tareas, de forma que queden claros los aspectos relevantes de la experimentación y si se obtienen los resultados esperados.
- las **herramientas de trabajo** utilizadas en la experimentación, tanto *hardware* como *software*.
- los **escenarios de trabajo** que componen los experimentos lelvados a cabo.
- los **resultados** obtenidos de la experimentación llevada a cabo.

Por último, en el cuarto capítulo ("Conclusiones") se analizan los resultados obtenidos y se comparten algunas reflexiones sobre el futuro del campo de investigación.

CAPÍTULO 2
Estado del arte

En este capítulo se expone la actualidad del conocimiento público que se tiene sobre las tecnologías que componen el objeto de estudio de este trabajo: sistemas embebidos, sistemas de tiempo real, sistemas de criticidad mixta y arquitecturas multinúcleo, sistemas operativos de tiempo real y placas de desarrollo. De cada tecnología se mencionan las características principales y de mayor interés para este estudio, así como las categorizaciones más usuales de cada campo. Todo esto con el objetivo de contextualizar este trabajo y de ofrecer las definiciones de los tecnicismos que aquí se emplean.

2.1 Sistemas embebidos

Los sistemas embebidos (SE) son muchos y muy variados, por lo que es complicado ofrecer una definición precisa y unívoca. La tarea de definir un SE ha llevado a muchos autores de libros sobre SE a exponer varias definiciones conocidas y empleadas usualmente, para después ceñirse a la definición que ponga mayor enfásis en los asuntos que ellos consideran centrales para el desarrollo o el estudio de SE. Véase, por ejemplo, [3] donde se pone enfásis en la modelización de los SE y en la interacción entre el computador y los procesos físicos; en [4] donde se centra en el propósito de los SE y sus componentes; en [5] donde la atención está sobre la finalidad de los SE y los desafíos que supone el desarrollo de los mismo; en [6] donde se le da mayor importancia a los estándares que debe implementar un SE, al *hardware* que lo compone y a la arquitectura de estos; o en **Shree K.K.**, donde se centra en los fundamentos de los SE y los componentes de este.

Algunas de las definiciones que se encuentran en los libros mencionados anteriormente son:

- Sistemas ciberfísicos que integran la computación con procesos químicos, biológicos, mecánicos, etc.
- Sistemas informáticos con recursos limitados (potencia de computo, mémoria ram, sistema operativo, etc.).
- Sistemas informáticos diseñados para una función específica.
- Sistemas informáticos con mayores requisitos de calidad y fiabilidad que los ordenadores de propósito general.
- Todo sistema de computación que no sea un ordenador de propósito general.
- Combinaciones de *software* y *hardware* encapsuladas en los dispositivos que controlan.

- Sistema electromecánico específicamente diseñado para una aplicación de un único dominio.

La definición que aquí se emplea es la siguiente: combinación de *software* y *hardware* diseñada para desempeñar tareas de control, comunicación o monitorización. Es una definición lo suficientemente abierta como para abarcar la gran variedad de sistemas embebidos que se pueden encontrar en cualquier sitio, y al mismo tiempo deja clara la finalidad con la que se emplean.

Hay muchos ámbitos de aplicación para los SE; van desde la automovilística, el control industrial, la robótica, la aviónica o la medicina, hasta pasar por la electrónica de consumo (DVD, juguetes, etc.) o la automatización de oficinas (fax, fotocopiadora, escáner, etc.) Por ejemplo, una lavadora es un sistema embebido, donde el sistema de control es un sistema informático y el resto de componentes, como el motor o el tambor, son los dispositivos de entrada/salida, que todos juntos forman la lavadora en su totalidad.

Figura 2.1: Lavadora como ejemplo de SE

Los SE se clasifican por generaciones (aunque también se pueden clasificar por su complejidad o su funcionalidad):

- **Primera generación**: en los primeros SE se utilizaban microprocesadores de 8 bits, como el 8085 de Intel®o el Z80 de zilog®, y microcontroladores de 4 bits. Los circuitos *hardware* eran muy simples y el *firmware* se programaba en código ensamblador. Ejemplos de SE de esta generación son las unidades de control de motores paso a paso, o un teléfono digital.

- **Segunda generación**: estos SE ya se diseñaban entorno a microprocesadores de 16 bits y microcontroladores de 8 y 16 bits, que contaban con juegos de instrucciones mucho más complejos y potentes que los de primera generación. En esta generación comienzaron a aparecer los sitemas operativos embebidos. Ejemplos de SE de esta generación son los sistemas de control y adquisición de datos (*SCADA systems*).

- **Tercera generación**: SE con microprocesadores de 32 bits y microcontroladores de 16 bits. A esta generación le acompañó la aparición de procesadores de aplicación

especifica, como los procesadores de señales digitales o los *Aplication Specific Integrated Circuits*. Los juegos de instrucciones eran más complejos y potentes que los de la generación pasada, y el concepto de segmentación de instrucciones estaba evolucionando también. SE de esta generación pueden encontrarse en areas como la robótica, los medios audiovisuales, el control de procesos industriales y muchos otros ámbitos.

- **Cuarta generación**: la aparición de los *System on Chip* (chips que integran multitud de funcionalidades aparte de la unidad de procesamiento central, como una interfaz de red), los procesadores reconfigurables (*FPGA*, por ejemplo) y los procesadores multinúcleo hace que los SE sean aún más potentes que los anteriores, con mayor capacidad de integración y con una tendencia hacia la miniaturización. Los SE de esta generación ya hacen uso de sistemas operativos embebidos de tiempo real de alto rendimiento. Ejemplos son los teléfonos inteligentes o los dispositivos móviles de internet.

Anteriormente, por los años sesenta, cuando aparecieron los SE, la problemática del diseño de SE era entendida como problemas de optimización ya que el *hardware* era muy caro y convenía gastar lo menos posible para la producción masiva de productos. Al querer contar con pocos recursos *hardware*, el *software* también se optimizaba lo máximo posible para adaptarlo a computadores con poca potencia de computo y escasa memoria principal. Por lo tanto los SE no se distinguian del resto de la informática más que por el aspecto de la optimización. Hoy en día, debido al abaratamiento y la mejora del *hardware* (como se puede ver en [5] y [6]), la cuestión de la optimización no ha sido olvidada, pero la cuestión de la interacción y el control de procesos externos (procesos físicos, químicos, biológicos, etc.) ha cobrado mayor importancia que la anterior, ya que contar con recursos *hardware* no es caro y se pretende que los SE hagan más cosas y con mayor fiabilidad (realidad virtual, realidad aumentada, expediciones espaciales, trenes, aviones, bracos, etc.).

2.1.1. Arquitectura de los sistemas embebidos

Siendo la interacción entre la computación y los procesos externos a esta la principal problemática de los SE actualmente, la modelización de los mismos es crucial ya que permite realizar diseños de SE sofisticados, escalables, fiables y seguros. La modelización de los SE consiste en describir ciertos aspectos de los mismos para conocerlos mejor. Cuanto más fiel sea a su comportamiento y deje fuera la mayor cantidad de detalles inecesarios, mejor será el modelo. Un ejemplo de modelo muy importante en lo SE es la arquitectura.

La arquitectura de un SE es una abstracción del mismo, sin entrar en detalles como serían su código fuente o los circuitos electrónicos que lo componen. Se trata de una mapa de interacciones entre los elemento. Elementos que representan tanto el *software* y el *hardware* que componen el SE, como entidades externas al dispositivo que interactuan con este. La arquitectura está formada por las diferentes estructuras que hacen falta para ilustrar el completo funcionamiento del dispositivo embebido, entendiendo por estructura lo siguiente: una instantánea de la interacción de diversos elementos del sistema, sin necesariamente incluir todos los elementos del sistema. Gracias a la arquitectura es más fácil definir y entender el diseño de un sistema embebido, diseñarlo dentro de unos costes limitados, determinar su integridad, su seguridad y su fiabilidad, etc. Todo esto antes si quiera de conocer la implementación física del sistema.

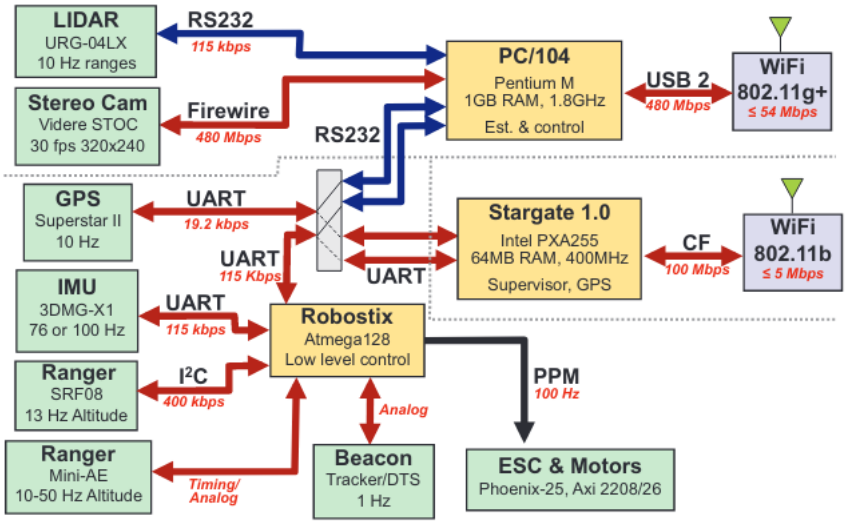

Figura 2.2: Ejemplo de arquitectura de un SE: Arquitectura del quadrotor Starmac

Arquitectura Von Neumann

La arquitectura Von Neumann encajaría con la definición de estructura antes dada. Es una estructura que representa, en abstracto, los elementos mínimos necesarios que componen un sistema informático; es una concreción de la máquina de Turing, donde, a diferencia de la arquitectura Harvard (donde se asigna una memória al almacenamiento de instrucciones y otra al almacenamiento de datos), las instrucciones y los datos se almacenan en la misma memoria. Esta arquitectura es interesante porque es común a todos los sistemas informáticos (que almacenan datos e instrucciones en la misma memoria) y, por lo tanto, a todos los sistemas embebidos que comparten esta característica.

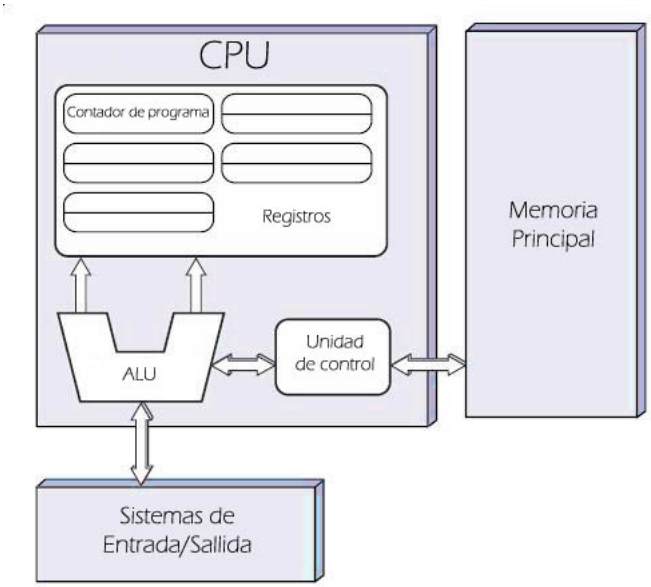

Figura 2.3: Arquitectura Von Neumann

Los componentes de la arquitectura son:

- ***CPU* (Unidad de procesamiento central)**: permite la interpretación y ejecución de instrucciones. Compuesta por un contador de programa, para saber a que dirección de memoria acceder para obtener la instrucción que corresponde ejecutar en cada momento; registros, para poder almacenar, acceder y operar con datos rápidamente; *ALU* (Unidad Aritmético-Logica en castellano), para poder realizar operaciones aritméticas y lógicas como sumas, restas, uniones o intersecciones; y unidad de control; para poder decodificar instrucciones y activar los operadores necesarios para llevar a cabo la ejecución de la instrucción decodificada. El mercado de ordenadores de propósito general está copado por la arquitectura x86, tanto en ordenadores personales como en servidores o supercomputadores. Sin embargo, el mercado de los SE, en cuanto a procesadores se refiere, es muy heterogéneo debido a la gran variedad de propósitos con los que se diseñañ SE. Algunos parámetros interesantes a la hora de escoger procesador son: la potencia de computo (medida en instrucciones por tiempo), el consumo energético (medido en energia por tiempo), el juego de instrucciones que ofrece (*RISC* o *CISC*), la memoria caché (medida en unidades de almacenamiento) o las velocidades a las que puede interactuar con la memoria principal (medida en unidades de frecuencia).
- **Memoria principal**: permite almacenar las instrucciones y los datos que componen los programas. Es donde reside principalmente el *software* del sistema. Se compone de celdas de memoria, cada una con su dirección correspondiente para permitir su selección. La gestión y el uso de la memoria principal es muy importante cuando se trata de computación paralela o concurrente. Es donde, debido a un mal planteamiento, se pueden provocar condiciones de carrera, bloqueos o incoherencias, que implican funcionamientos no deseados. Por otro lado, escoger la memoria de un SE no es una tarea trivial: las memórias pueden ser volátiles (sin corriente pierden la información) o persistentes (discos magnéticos, por ejemplo); cuanto más almacenan, más lentas son; cuanto más rápidas son, más energia consumen y son más caras. Generalmente se suele implementar una jerarquía de memorias, donde se combinan memorias de diferentes características, para optimizar el tamaño, la latencia (tiempo que tarda en responder), el coste y el consumo energético.
- **Sistemas de Entrada/Salida**: permiten la interacción del procesador con su entorno. Se trata de sensores o actuadores, en su sentido más amplio, análogicos o digitales, que miden o alteran fenómenos externos. La tendencia actual va en dirección del internet de las cosas, pero en muchos casos esa solución no es conveniente por cuestiones de rapidez y fiabilidad. Los parámetros determinantes a la hora de seleccionar los componentes de Entrada/Salida en el proceso de diseño de un SE son: la frecuencia de muestreo o actuación, según el caso (tiempo que conlleva tomar una medida o llevar a cabo una acción), la constante de proporcionalidad entre una muestra o una acción y el fenómeno medido o provocado (relación entre la energía adquirida o necesaria y los cambios medidos o provocados), el sesgo o *bias* (ruido o diferencia entre lo medido o actuado y el valor real del fenómeno) y el rango de valores posibles, tanto de medición como de actuación. En los SE, los componentes de entrada y salida son de gran importancia ya que, si el procesador y la memoria sirven para automatizar el procesado de información, los componentes de entrada/salida, bajo las ordenes del procesador, llevan a cabo las acciones que debe desempeñar un SE. Existen diversas formas de interacción entre el procesador y los dispositivos de entrada/salida: mediante interrupciones, donde el dispositivo en cuestión notifica al procesador que debe gestionar un evento del dispositivo mediante una señal que el procesador es capaz de identificar (un sensor de movi-

miento envía una señal de interrupción cuando detecta movimiento); mediante un bucle en el que el procesador comprueba el estado del dispositivo hasta percibir un cambio y entonces actuar; directamente ordenando operaciones al dispositivo, sin esperas ni señales de interrupción. Ejemplos de sistemas de entrada/salida son: un brazo mecánico, un servomotor, un pulsador, un puerto USB, una tarjeta de red, una tarjeta de vídeo, un altavoz, un sensor de calor, un lector de tarjetas, etc.

2.2 Sistemas de tiempo real

Como se expone en [7] los sistemas de tiempo real (STR) son sistemas informáticos en los que la ejecución de ciertas tareas debe realizarse dentro de cotas temporales predeterminadas. Es decir, sistemas que basan la corrección de su comportamiento, entendido este como la secuencia de *outputs* en el tiempo, en la lógica de sus resultados y, también, en el tiempo de producción de los mismos.

Se pueden encontrar STR en muy diversos ámbitos: sistemas de telecomunicación, sistemas de procesado de señales, sistemas de fabricación automática, aviónica, control aéreo, sistemas de control de energía nuclear, etc. Por ejemplo, en una máquina expendedora, el sistema embebido encargado de gestionar dicha máquina deberá tener en cuenta el tiempo que tarda el sistema de expulsión, ya sea este un brazo mecánico o una cinta transportadora, para poder desempeñar correctamente su función, por lo que se considera un STR.

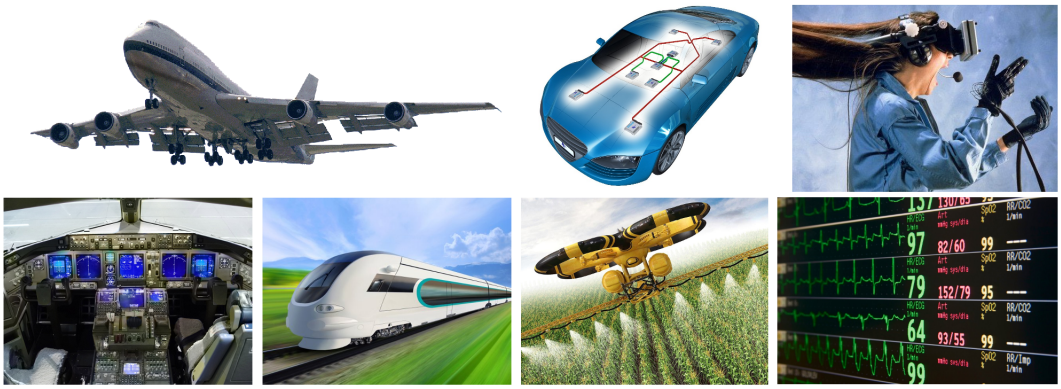

Figura 2.4: Ejemplos de STR

Generalmente, los STR no son visibles de cara al usuario y forman parte de un sistema mayor, al igual que los sistemas embebidos y al contrario que los ordenadores de propósito general. También se caracterizan por la importancia que tiene la interacción del STR con el mundo exterior, a través de sensores, accionadores y periféricos en general, y por requerir un proceso de diseño complejo, en el que se tienen en cuenta factores como la arquitectura *hardware* y *software*, el sistema operativo sobre el que todo se ejecuta (en caso de que se ejecute sobre un sistema operativo), el lenguaje de programación, etc. Al contrario que un ordenador de sobremesa, por ejemplo, donde se suele tener recursos de sobra para poder ser de propósito general.

No todos los STR requieren de un sistema operativo. Los hay que su *software* es compilado para ser ejecutado directamente sobre el *hardware* (lo que se conoce como *bare-metal programming*). En estos casos los costes temporales son, únicamente, los que implican el uso de las operaciones que ofrece el *hardware* (acceder a un byte de memoria, sumar dos números enteros, etc.). Sin embargo, también los hay que hacen uso de un sistema operativo. El sistema operativo que se utiliza en los STR es un sistema operativo de tiempo

real. Un sistema operativo de tiempo real es aquel que tiene medido el coste temporal de todas las operaciones que ofrece, más aparte reúne ciertas características que propician su uso en STR: acceso a *buffers* sin esperas, planificación de tareas, bajo uso de memoria, etc.

El sistema que contiene un STR se divide en tres *clusters* (subsistemas autocontenidos): el *cluster* computacional, que corresponde al STR en sí, encargado de ejecutar el algoritmo de control; el *cluster* operativo, que corresponde a la persona o máquina encargada de controlar y supervisar el comportamiento del STR; y el *cluster* controlado, que corresponde al objeto monitorizado y manipulado por el *cluster* computacional (por ejemplo, una reacción química o el freno de un coche). Estos dos últimos *clusters* mencionados conforman el entorno del *cluster* computacional. El *cluster* computacional se puede dividir en nodos, cada uno encargado de un algoritmo de control, y estos nodos estar conectados por una red de comunicación de tiempo real, conformando un STR distribuido, que son los más usuales. Dada está división del sistema se pueden identificar dos interfaces: la existente entre el *cluster* computacional y el *cluster* operador, compuesta por pantallas, teclados, etc. Y la interfaz existente entre el *cluster* computacional y el *cluster* controlado (interfaz de instrumentación), compuesta por sensores y actuadores. El STR debe reaccionar ante los estímulos de ambas interfaces (entorno) dentro de los plazos que requiere el entorno.

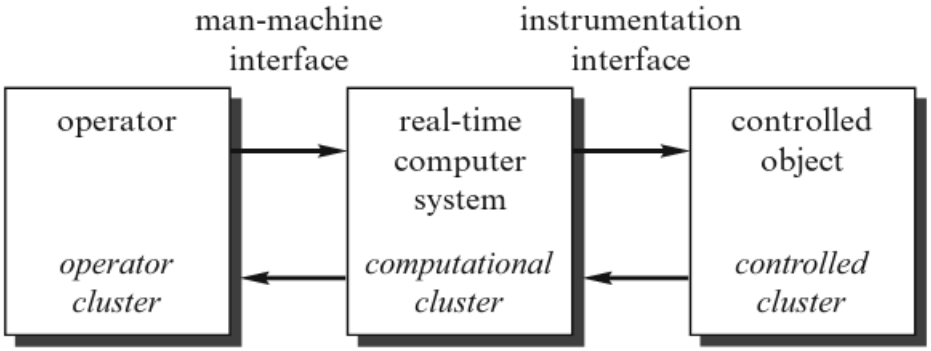

Figura 2.5: División de un STR en *clusters* y sus interfaces

La interfaz de instrumentación lleva a cabo la manipulación del *cluster* controlado mediante sensores y actuadores. Este tipo de control se conoce como control digital directo y suele ser muy estándar. Se trata de un bucle que, primero, hace uso de los sensores para conocer el estado del sistema, segundo, ejecuta un algoritmo que determina la acción a llevar a cabo para alcanzar un estado deseado, tercero, lleva a cabo la acción necesaria, y por último se vuelve a conocer el estado del sistema mediante los sensores para ver el efecto producido por la última acción, volviendo a empezar el bucle. La rama de ingeniería encargada del estudio de este tipo de manipulación se conoce como control.

La interfaz entre el *cluster* computacional y el *cluster* operativo se encarga de informar sobre el estado del sistema y asistir en el control del mismo. Esta interfaz debe ayudar al operario de la forma más amena posible ya que muchos accidentes están provocados por una mala interpretación del estado del sistema. Esta interfaz cuenta con un histórico de datos acorde a las demandas de la industria (por ejemplo, en algunos países, la industria farmacéutica esta intensamente monitorizada para poder rastrear fallos en la producción en caso de producirse efectos adversos indeseados en los consumidores).

Los STR, según la gravedad de las consecuencias provocadas por el incumplimiento de los plazos de ejecución, se pueden clasificar en tres tipos: STR rígidos, que son los que el incumplir un plazo de ejecución conlleva consecuencias catastróficas , es decir, que

ponen en riesgo la vida de personas o que conllevan un coste económico mucho mayor que el coste del STR (por ejemplo el freno de un coche); STR flexibles, que preservan la validez de los resultados de sus operaciones a pesar de estar llevados a cabo fuera de plazo; y los STR firmes, a los que no les sirven los resultados fuera de plazo pero tampoco provocan consecuencias catastróficas por ello.

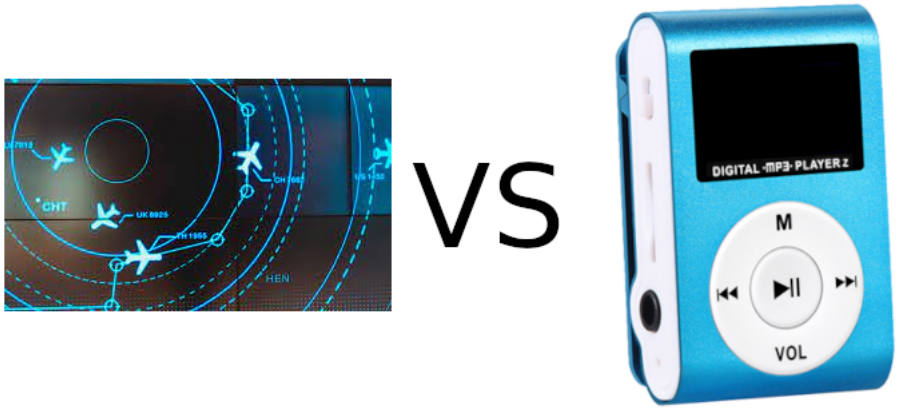

Figura 2.6: A la izquierda un STR rígido. A la derecha un STR flexible.

Para poder diseñar correctamente un STR es necesario conocer sus especificaciones temporales, esto es, saber la procedencia de los costes temporales y tener acotados los mismos. Las especificaciones temporales más relevantes del STR provienen de los bucles de control, ya que la interacción entre el *cluster* computacional y el *cluster* operador va a ser ordenes de magnitud mayor que cualquier otro proceso del sistema (debido a lo lento que reacciona un humano con respecto a una máquina). Para conocer bien las especificaciones temporales de un bucle de control se deben tener en cuenta dos tiempos clave: el tiempo existente entre la emisión de una orden y el instante en el que se comienzan a notar los efectos de tal acción, conocido como retraso objeto, y el tiempo que se tarda en alcanzar el estado deseado tras una acción, conocido como tiempo de alcance. Estos dos tiempos se pueden conocer mediante la gráfica que relaciona el estado y las acciones con el tiempo físico. A raíz de estos dos tiempos se establecen los tiempos que debe tardar el algoritmo de control en realizar sus operaciones: el periodo de observación (tiempo entre una medición del estado y la siguiente) debe ser una décima parte del tiempo de alcance; el retraso de computación (tiempo entre una observación del estado y la emisión de un valor para el actuador correspondiente) debe ser menor que el periodo de observación; el tiempo muerto (tiempo entre una medición y los primeros efectos tras la acción que se haya decidido realizar) debe ser lo menor posible (es la suma del retraso objeto y el retraso de computación) ; y, por último, las irregularidades (diferencia entre el mayor y el menor de los retrasos del sistema) deben ser lo menores posibles para diseñar un algoritmo de control con un retraso de computación constante, de forma que se pueda conocer con la mayor precisión posible el instante en el que se actualizó la medida del estado del sistema. Otro parámetro relevante es el tiempo de detección de fallos en el sistema, que debe tener un orden de magnitud, como máximo, igual al del bucle de control crítico más rápido, de forma que se asegure que un fallo en el sistema no trascenderá en daños severos.

Los requisitos para evaluar la confianza de un STR son los siguientes:

- **Fiabilidad**: es la probabilidad de que el sistema ofrezca el servicio esperado en un intervalo de tiempo. Se mide en fallos por intervalo de tiempo (*FIT* por sus siglas en ingl'es *Failure In Time*). 1 FIT = 1 fallo cada 10^9 horas. Un sistema con un ratio de fallos de 1 *FIT* tiene una fiabilidad superalta.
- **Seguridad**: es la fiabilidad durante modos de ejecución críticos. Aquí se debe distinguir entre fallos benignos (asumibles) y malignos (inasumibles). Un fallo maligno en estos modos de ejecución puede suponer costes de varios ordenes de magnitud mayores que la utilidad del sistema en sí. Un STR es seguro ante situaciones críticas cuando ofrece una fiabilidad superalta. La corroboración de la seguridad de un sistema se lleva a cabo mediante los procesos de certificación de los que se hablará más adelante.
- **Mantenibilidad**: es el tiempo necesario para reparar el sistema tras sufrir este un fallo benigno. La mantenibilidad entra en conflicto con la fiabilidad ya que, una buena mantenibilidad requiere de una alta modularidad de los componentes del sistema pero, una alta modularidad requiere de un sistema de comunicaciones (entre módulos) al que se puedan desconectar los componentes en un momento dado, lo que implica un mecanismo de conexión al sistema de comunicaciones menos fiable que uno al que no te puedas desconectar, ya que el anterior es susceptible de desconexiones involuntarias. La tendencia de la industria apunta hacia lo que se conoce como inteligencia ambiental (sistemas que se autodiagnostican y que son mantenibles por usuarios inexpertos).
- **Disponibilidad**: se mide por el tiempo que el sistema está listo para ofrecer sus servicios con respecto al tiempo de funcionamiento total.
- **Ciberseguridad**: consiste en la autenticidad e integridad de la información que maneja el sistema. Para ello se utilizan técnicas de autentificación y de encriptación. Con el auge del internet de las cosas este requisito cobra mayor importancia debido a que los dispositivos no solo comprometen la información si no también el entorno.

En varios de los ámbitos de aplicación de los STR, como la aviónica o los sistemas de control, un mal funcionamiento de estos sitemas puede implicar consecuencias muy graves, por lo que se destinan muchos esfuerzos en las tareas de validación rigurosas que demuestran que los sistemas se comportan como se espera, que son seguros según la definición ofrecida anteriormente. La forma más usual de certificar un sistema es contratando una empresa certificadora externa.

El proceso de certificación se puede simplificar si la empresa certificadora puede estar convencida de que:

- Los subsistemas críticos están protegidos frente a la propagación de fallos externos a ellos (contención de fallos).
- El sistema cuenta con los recursos necesarios para servir frente a todas las cargas posibles, sin mentar probabilidades de éxito.
- Los módulos del sistema se pueden certificar independientemente. Solo las propiedades emergentes de juntarlos deberán ser certificadas posteriormente.

Como se afirma en [8], las especificaciones para que un sistema sea validable son las siguientes:

- Modelo del sistema completo y preciso. Todas aquellas propiedades que escapen al modelo deben ser mesurables fácilmente.
- Mapa de estados sin transiciones a estados de error o fallidos, acompañado de argumentos analíticos que demuestren que los fallos que puedan ocurrir no llevarán a fallos en el funcionamiento del sistema.
- Toda solución de compromiso debe llevar al sistema a reducir el número de parámetros a medir.

Los STR se pueden clasificar de diferentes formas. Aquí se exponen varias de ellas: flexibles vs rígidos, seguro ante fallos vs operativo ante fallos, respuesta garantizada vs mejor esfuerzo, recursos adecuados vs recursos inadecuados y guiados por eventos vs guiados por tiempo:

- **Flexibles** vs **Rígidos**:
  - **Tiempo de respuesta**: mientras que los STR flexibles admiten la intervención de personas en el curso de su funcionamiento, y pueden funcionar con esperas del orden de segundos, los STR rígidos deben ser autónomos, sin permitir la intervención humana (ya que esta introduciría esperas no deterministas), y deben operar con esperas del orden de milisegundos.
  - Rendimiento ante situaciones de estrés: los STR flexibles ofrecen un rendimiento medio aceptable, pero no aseguran un completo funcionamiento durante situaciones donde la carga de trabajo es límite (muchas veces porque es inasumible económicamente poder funcionar ante todas las cargas de trabajo posibles). Durante situaciones de estrés funcionan a un rendimiento bajo pero tolerable. Los STR rígidos deben definir durante su diseño todas las cargas de trabajo a las que se pueden enfrentar, para así disponerlos de todos los recursos necesarios con el fin de operar correctamente en todo momento. Bajo ninguna circunstancia se puede permitir incumplir los plazos de ejecución.
  - **Ritmo**: un STR flexible, ante situaciones de estrés, puede trabajar a un ritmo própio al que se adaptará su entorno. Un STR rígido debe adaptarse al ritmo de su entorno en todo momento.
  - **Seguridad**: el fallo de un STR flexible no conlleva consecuencias catastróficas, al contrario que un STR rígido. Por ello, la detección de fallos en un STR rígido debe ser automática, a tiempo y autónoma, sin necesidad de intervención humana.
  - **Tamaño de los archivos**: un STR flexible debe centrarse en la integridad de los datos a largo plazo y la disponibilidad de los mismos, por lo que puede operar con archivos de gran tamaño. Un STR rígido, aparte de centrarse en la integridad de los datos, tiene su principal problemática, con respecto al uso de datos, en la precisión de su base de datos de tiempo real (conjunto de valores de las variables que definen el estado del sistema) y como afecta el paso del tiempo a esta, por lo que conviene que el STR trabaje con archivos lo más pequeños posible.
  - **Redundancia**: mientras que un STR flexible puede guardar el conjunto de variables de estado a modo de punto de retorno al que volver en caso de fallos, un STR rígido no puede ayudarse de tal mecanismo por los siguientes motivos: el tiempo necesario para restablecer un estado del sistema anterior es impredecible; hay acciones que han modificado el entorno las cuales son irreversibles; la imprecisión temporal que puede derivar por desconocer el tiempo necesario para restablecer el sistema.

- **Seguro ante fallos** vs **Operativo ante fallos**: un STR seguro ante fallos es un STR capaz de alcanzar estados seguros tras el acontecimiento de un fallo. La capacidad de alcanzar un estado seguro depende de la naturaleza del *cluster* controlado (por ejemplo, un sistema de semáforos, tras un fallo, puede alcanzar un estado seguro poniendo todos los semáforos en rojo). La detección de errores en este tipo de STR debe ser lo más próxima a 1 posible (muy alta). Normalmente se utiliza un computador que monitoriza externamente el estado del STR (*watchdog*). Un STR operativo ante fallos es un STR en el que, por la naturaleza del *cluster* controlado, es imposible definir estados seguros. En estos sistemas los esfuerzos se centran en diseños que ofrezcan un mínimo de operatividad en situaciones de error.

- **Respuesta garantizada** vs **Mejor esfuerzo**: un STR que ofrece respuestas garantizadas ha planteado, durante su diseño, unos fallos y unas cargas hipotéticas que van a poder ser atendidos con total seguridad (sin apelar a probabilidades de error). La probabilidad de errar en el planteamiento de fallos y cargas hipotéticas se llama cobertura de supuestos. En los casos en los que no se pueden ofrecer hipótesis completas, la opción a llevar a cabo es un STR que ofrezca un mejor esfuerzo. Este estilo de diseño no requiere de hipótesis rigurosas, sino más bien un proceso de prueba y error a través del cual se va mejorando un diseño inicial. Estos STR albergan incógnitas sobre su funcionamiento ante casos aislados.

- **Recursos adecuados** vs **Recursos inadecuados**: dado un STR en el que se pueden plantear hipótesis sobre todas las cargas a las que se va a enfrentar, un STR con recursos adecuados es al que se le dispone de los recursos necesarios para funcionar ante todas las hipótesis previamente planteadas (tanto cargas de trabajo extremas como escenarios de fallo). Estos STR ofrecen un funcionamiento garantizado. Sin embargo, si no se le dispone de todos los recursos necesarios se trata de un STR con recursos inadecuados (en la mayoría de los casos se trata de sistemas a los que es inviable económicamente hablando dotarlos de todos los recursos necesarios). Estos STR hacen uso de lo que se conoce como asociación dinámica y compartición de recursos. Con esta técnica son capaces de hacer frente a una cantidad aceptable de casos hipotéticos.

- **Guiados por eventos** vs **Guiados por tiempo**: si las acciones de control y comunicación de un STR se inician tras eventos de cualquier tipo, quitando los eventos generados por los ciclos de reloj, se trata de un STR guiado por eventos. Estos STR utilizan las interrupciones como mecanismo de notificación de eventos. Ello implica que se debe diseñar una planificación dinámica que se adecue al entorno de cada momento, ejecutando así la acción correspondiente al evento notificado en cada situación. Por otro lado, si el desencadenante de las acciones de control y comunicación son las interrupciones generadas por el reloj interno del sistema se trata de un STR guiado por tiempo. Estos STR trabajan con una gran predictibilidad. En un STR distribuido guiado por tiempo, todos los relojes deben estar sincronizados formando un reloj global disponible para todos los nodos. La precisión del reloj debe ser escogida de forma que se pueda ofrecer un histórico de sucesos, cada suceso con su marca temporal, fielmente secuenciado.

Cabe destacar los diferentes modelos de organización de software de tiempo real que se encuentran, al menos uno de ellos, en cada STR:

- **Con sistema operativo**: un sistema operativo ofrece una interfaz para acceder a los recursos *hardware* de forma segura, además de muchas otras funcionalidades, como la planificación de ejecución de procesos. Las diferentes formas de planificar la ejecución, según [9], son las siguientes:

- **Ejecutivos cíclicos** (*Frame-based*): se basan en secuencias de tareas, conocidas como marcos mayores (*major frames*), divididos en marcos menores (*minor frames*), donde se planifica la ejecución de cada tarea. Los marcos mayores se repiten constantemente de forma cícilica. Un reloj da una señal de aviso con el comienzo de cada marco menor, y una tarea se encarga de llamar a la tarea correspondiente para cada marco menor.
- **Dirección por eventos** (*event-driven*): se utilizan señales producidas por sistemas de entrada/salida, o eventos de un reloj, para desencadenar la ejecución de tareas planificables. Las tareas se pueden priorizar de varias maneras (en función de su duración, por ejemplo). Para permitir concurrencia de ejecuciones se debe hacer uso de semáforos, mutex, etc.
- ***Pipelined systems***: basados en mensajes entre tareas (preferiblemente con algún orden de prioridades), además de en señales de reloj y de sistemas de entrada/salida. El flujo de control de un evento va desde la fuente del evento hasta el destino, dejando el sistema como un conjunto de flujos de invocación de tareas. Debido a como se encadena la ejecución entre tareas, que es con mensajes, el análisis de estos sistemas se hace más complicado, teniendo que tener en cuenta, a la hora de modelizar las tareas, todos los mensajes diferentes que puede recibir.
- **Sistemas cliente-servidor**: de nuevo se utilizan mensajes entre tareas, además de señales de relojes o externas. Las comunicaciones se realizan sincrónicamente, bloqueándose las tareas que inician las comunicaciones (clientes) hasta que otra tarea les responda (servidores). El control de los eventos lo realiza una única tarea, lo que los hace mas fáciles de analizar (*debugging*, *checkpointing* y procesado de errores) que los *pipelined systems*.
- **Sistemas de estados**: basados en la modelización de los diferentes estados del sistema, donde cada estado representa unos posibles valores para las variables del sistema. Cada estado representa un modo de funcionamiento del sistema. Ante un evento se produce un cambio de estado, que puede desencadenar la ejecución de ciertas tareas. Generalmente, los sistemas de estados no aceptan eventos hasta haber finalizado la transición provocada por el evento anterior. Para cumplir con esto, suelen estar controlados por un único bucle de eventos que recibe y procesa los eventos. El análisis temporal de estos sistemas depende de la prioridad de cada tarea, la asociación entre estados y tareas y de la planificación sobre los eventos a recibir.

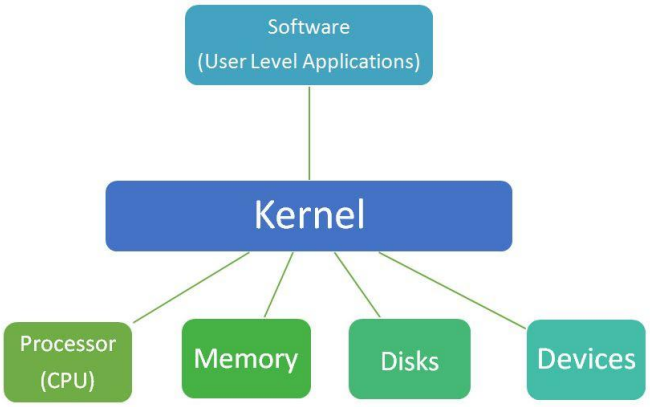

Figura 2.7: Esquema de un computador con sistema operativo

- **Sin sistema operativo**: otra forma de ejecutar el software es sin sistema operativo que medie el acceso a los recursos *hardware*. Esta forma de programar se conoce

como programación *bare metal*. Así se logra un control absoluto del *hardware*, pero requiere de un mayor tiempo de programación al tener que adaptarlo todo a la arquitectura del procesador, la arquitectura del computador, etc.

Por último, un ejemplo de STR claro es un motor de coche actual. Debe ser actual porque hasta los años ochenta no se introdujo electrónica para mejorar el funcionamiento del motor. El objetivo del STR que se encuentra en un motor es la optimización del funcionamiento en términos de consumo de combustible, de desgaste del propio motor y de contaminación. Para cumplir con estos propósitos, por ejemplo, se controla electrónicamente el momento, lo más preciso posible, en el que se debe inyectar combustible en los pistones, para que la combustión se genere en el momento que más se vaya a aprovechar la fuerza que genere, al igual que se controla la cantidad de combustible que se inyecta, para así no desperdiciar combustible y generar una combustión que genere el empuje deseado. Esto se logra mediante un conjunto de sensores y actuadores armonizados mediante un *software* de tiempo real. Concretamente, a través de un sensor en el cigüeñal (sensor inductivo o de efecto *hall*), se puede conocer la posición de los pistones en todo momento. Conociendo esto, la intención del operario, la carga del motor, la temperatura del mismo, y muchos otros factores, se puede calcular el momento de la inyección del combustible y la cantidad que se debe inyectar. Pongase por caso un motor a seis mil revoluciones por minuto. Este motor tarda diez milisegundos en realizar una revolución completa (una vuelta completa, de trescientos sesenta grados, del cigüeñal). Si se requiere un sensor con una precisión de cero coma un grados, temporalmente el sensor deberá tener una precisión de tres microsegundos. A su vez, se debe calcular en todo momento el retraso entre la emisión de la orden de inyectar combustible y el momento real en el que se inyecta combustible: tiempo que varia constantemente según las condiciones del entorno.

Con este ejemplo se puede ver claramente como se debe modelizar el sistema para cumplir con la función que se espera de el, al igual que se puede ver el necesario uso de protocolos para secuenciar bien las acciones a llevar a cabo.

2.3 Sistemas de criticidad mixta

En 2007 Steve Vestal publica, para el vigésimo octavo simposio internacional de STR organizado por la IEEE, un artículo [10] en el que se analiza el modelaje y la planificabilidad de unos STR donde conviven componentes con diferentes niveles de criticidad, llamados sistemas de criticidad mixta (SCM). Ahí se entiende la criticidad como el nivel de garantía con el que se asegura que una tarea va a finalizar su ejecución antes de superar el tiempo de ejecución del peor de los casos (*WCET: Worst Case Execution Time*). No se especifica como se mide ese nivel de garantía, sino más bien se afirma que, a mayor nivel de criticidad, más exhaustivas deben ser las pruebas que corroboren el cumplimiento deplazos por parte de la tarea en cuestión. Otra afirmación de partida de ese artículo es que a mayor nivel de criticidad, más largos y menos arriesgados son los plazos de ejecución, lo que provoca planificaciones menos eficientes. El objetivo del artículo de Vestal es lograr un análisis de planificabilidad de SCM más preciso y planificaciones, de prioridades fijas, más eficientes. Es en ese artículo donde se acuña el término SCM y es a ráiz de el que se avanza en las investigaciones de ese campo.

Figura 2.8: Ejemplo de SCM

El artículo de Vestal surge para mejorar el diseño y análisis de los SCM, un tipo de STR que son cada vez más usuales por la tendencia industrial, como son el sector automovilístico o el de la aviónica, de integrar, en una misma plataforma *hardware*, componentes con diferentes niveles de criticidad. Esta tendencia obedece a dos factores principalmente: por un lado, a la gran capacidad de computo de los procesadores contemporáneos, que permite que con menos *hardware* se ejecute más *software*, y por otro lado, a la intensificación de los requisitos del diseño de STR en términos de coste económico, espacio, peso, producción de calor y consumo energético.

Una de las problemáticas principales del diseño de SCM es la reconciliación entre el aislamiento, temporal y espacial, entre tareas (evitar todo tipo de interferencia entre tareas, simulando que el *hardware* pertenece únicamente a cada tarea) y la compartición de recursos (la integración de tareas en una misma plataforma *hardware*). Superar está problemática requiere de un buen análisis de los requisitos de las tareas y del sistema, un modelaje lo más realista y detallista posible y una planificación dinámica de la ejecución capaz de asegurar la ejecución de las tareas críticas en todo momento. En los casos en que se utiliza un procesador multinúcleo es imposible evitar completamente las interferencias entre tareas, ya que ciertos recursos del procesador son compartidos por todos los núcleos del mismo. De todas formas, los esfuerzos por aislar espacialmente las tareas se centran en el uso compartido de la memoria principal, permitiendo que cada tarea cuente con su espacio de direcciones propio, sin tener en cuenta el uso que hagan las otras tareas de este recurso. En [12] resuelven esta problemática con dos niveles de planificación. Ahí agrupan las tareas en conjuntos. Cada conjunto tiene una planificación de la ejecución propia, que corresponde al primer nivel de planificación, y, después, cada conjunto de tareas obtiene el uso del *hardware* tanto tiempo como dictamine un segundo planificador global, que corresponde al segundo nivel de planificación. Esta no es la única forma de resolver la problemática. En ese caso estudian los sistemas abiertos, que son un caso específico de SCM en el que no se pueden conocer a priori todas las combinaciones posibles de tareas ejecutandose simultáneamente. Esta problemática se conoce como el particionado de sistemas (un sistema particionado es aquel que permite compartir recursos *hardware* asegurando que no habrá interferencias entre tareas).

Como se señala en [11], el concepto de criticidad depende de la naturaleza de la tarea en cuestión; depende de las consecuencias que provoque el incumplimiento de plazos. Por lo tanto, en esa publicación, se propone el nombre de sistemas de modelos múltiples. El modelo de tarea propuesto en el artículo de Vestal se compone de cuatro parámetros: T (periodo de ejecución), D (instante límite de la ejecución de una tarea), C (tiempo de ejecución en el peor de los casos) y L (nivel de criticidad). Estos parétros pueden tener varios rangos de valores según el nivel de criticidad. Por ejemplo $D(CriticidadAlta) > D(CriticidadBaja)$. Es por ello que tiene sentido denominar a estos sistemas sistemas multimodelo. Alan Burns propone categorizar como metaparámetros aquellos parámetros que condicionen los rangos de posibles valores de otros parámetro y ofrece una lista de metaparámetros comunes: criticidad, importancia, valor, robustez, resiliencia y nivel de seguridad.

Los diferentes niveles (interpretaciones) de criticidad están registrados en estándares. Cada estándar se encarga de determinar las garantías contra fallos necesarias para un sector industrial en concreto. Por ejemplo, el estándar ISO 26262, definido por la Organizacion Internacional para la Estandarización (*ISO* por sus siglas en inglés) describe la seguridad que debe ofrecer un vehículo de carretera, incluyendo los componentes electrónicos; o el estándar IEC 61508, definido por la Comisión Electrotécnica Internacional, que describe como deben ser las diferentes partes del proceso de producción de los sistemas de protección automática, de todos los sectores industriales, para que estos sean seguros contra los diferentes riesgos presentes allá donde se implantan.

Task	ζ_i	C_i(LO)	C_i(HI)	T_i	U_i(LO)	U_i(HI)
τ_1^{HI}	HI	0.6	2.4	3	0.2	0.8
τ_1^{LO}	LO	0.8	0.8	2	0.4	0.4
τ_2^{LO}	LO	0.6	0.6	3	0.2	0.2

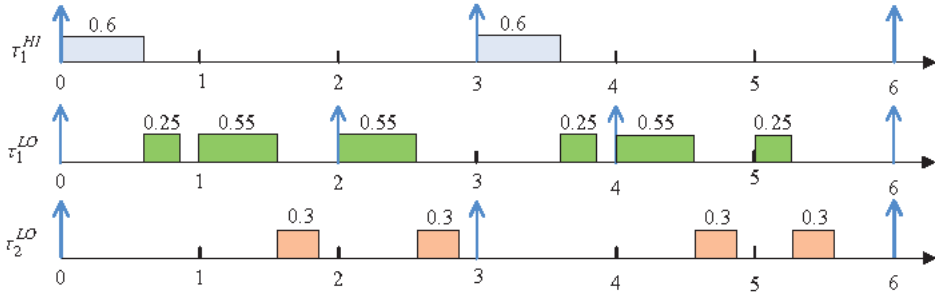

Figura 2.9: Planificación de la ejecución teniendo en cuenta dos niveles de criticidad para la tarea t1

Aunque existen SCM que operan con procesadores mononúcleo, para los que se han desarrollado planificadores de la ejecución eficaces y eficientes, este trabajo se centra en los SCM con procesadores multinúcleo, que cada vez están más presentes debido a la tendencia anteriormente mencionada. En estos sistemas se debe tener en cuenta el coste añadido que implica el uso de un sistema operativo para gestionar los recursos y las planificaciones, realizar los cambios de contexto (cambios de las tareas que ocupan los recursos *hardware*), etc. En [13], motivados por la existencia de SCM en la industria aeronáutica (aviónica), estudian un mecanismo que ofrezca aislamiento temporal entre los diferentes niveles de criticidad, es decir, conseguir que tareas de menor criticidad no interfieran negativamente en la ejecución de tareas de mayor criticidad. Según el modelo que utilizan para su sistema, a mayor nivel de criticidad, más exigentes deben ser los métodos empleados para determinar los costes de ejecución, y viceversa, a menor nivel de criticidad, métodos menos exigentes. Identificaron cinco niveles de criticidad, extraídos del estándar para la certificación de *software* aeronáutico (de la estadounidense comisión radiotécnica para la aeronáutica), que iban del nivel A (el más alto) al nivel E (el más bajo) según el daño que puede conllevar un malfuncionamiento del *software*, desde daños catastróficos hasta daños sin consecuencias.

Nivel de criticidad	Nivel de daño	Interpretación
A	Catastrófico	Puede conducir al estrellamiento de la nave.
B	Peligroso	Grave impacto en la seguridad o el rendimiento de la nave, o genera malestar físico o una carga de trabajo demasiado alta para operar correctamente, o genera daños graves entre los pasajeros.
C	Grave	Significativo pero con menor impacto que uno peligroso (por ejemplo, uno que genera incomodez en lugar de daños graves entre los pasajeros).
D	Leve	Notable pero con menor impacto que uno grave (por ejemplo, generando inconveniencias a los pasajeros o cambios en el itinerario)
E	Sin repercusiones	Sin impacto en la seguridad, operabilidad de la nave o capacitación de la tripulación

Con respecto a la planificación de la ejecución en SCM sobre procesadores multinúcleo comentan que son necesarios dos clases de algoritmos: algoritmos para las decisiones de planificación que formen parte del sistema operativo, y algoritmos que comprueben la correción temporal de los resultados de los anteriores algoritmos. Con respecto a los primeros se pueden encontrar tres clases: algoritmos de planificación globales, de particionado o una combinación de los dos, siendo los algoritmos de particionado mejores para SCM mayoritariamente rígidos (implican menor consumo de recursos). Para conseguir corrección temporal, el sistema operativo debe ofrecer aislamiento temporal.

La carga de trabajo que plantean es una secuencia de tareas con periodos harmónicos (periodos mayores son múltiplos de los periodos menores). Un ejemplo de esta tendencia son los sistemas de aviónica modular integrada (AMI).

El campo de los SCM está en constante expansión, con nuevas propuestas de investigación con cada investigación concluida. De momento no se conocen formas de modelaje y planificación que sean, ni perfectas, ni convenientes para todos los SCM, por lo que, a la hora de diseñar un SCM, cabe conocer bien los requisitos del sistema, para ubicarse correctamente y encontrar las mejores soluciones para el tipo de sistema que se está diseñando.

2.3.1. Aviónica modular integrada

La aviónica es la electrónica que se encuentra en la industria aeronáutica (naves espaciales, satélites artificiales, aviones, etc.), más precisamente la que se encuentra en las naves. Esta electrónica se encarga de dotar a las naves de funcionalidades como la comunicación por radio, la localización GPS, el piloto automático, etc. La aviónica se componía inicialmente por muy pocos dispositivo: primero la radio, luego el radar, más tarde el piloto automático, etc. Cada dispositivo contaba con su propio *hardware*. Actualmente, siguiendo la tendencia anteriormente mencionada, se están integrando las funcionalidades que ofrecían los diferentes dispositivos que componían la aviónica en SCM que comparten *hardware*. La aviónica, entonces, se compone principalmente de un conjunto de sensores, actuadores, computadores y programas informáticos conectados por un bus ARINC.

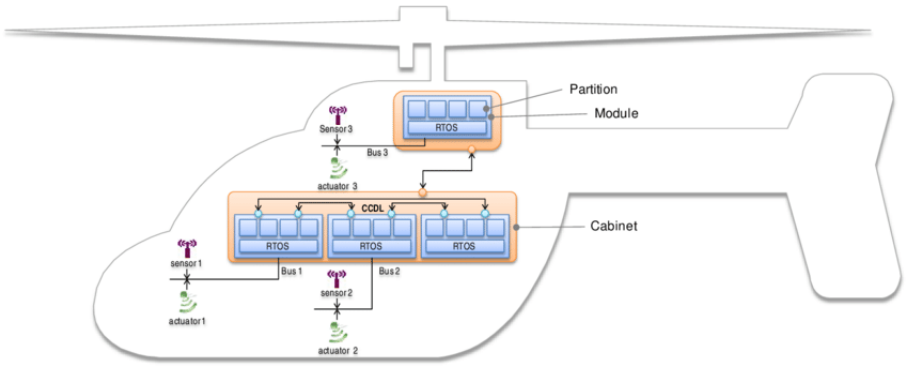

Figura 2.10: Ejemplo de aviónica modular integrada

Es aviónica modular porque el sistema es ampliable, se compone de dispositivos reemplazables interconectados por un bus de datos. Es aviónica integrada porque donde antes había cientos de controladores distribuidos ahora hay computadores con software realizando las diferentes tareas de control.

En [17] se muestran los beneficiarios y los beneficios de esta integración: compañías aéreas, fabricantes de aeronaves, fabricantes de aviónica y compañías certificadoras.

- **Compañías aéreas**: decremento del coste del ciclo de vida de la aviónica gracias al aumento de rendimiento de la misma. Reducción del mantenimiento de imprevistos y del recambio de piezas y simplificación de la ampliación de las funcionalidades de la aviónica.
- **Fabricantes de aeronaves**: reducción de costes gracias a un desarrollo más simple, una certificacón más accesible y fácil y una producción más barata. Además de reducirse el peso de las naves, aumentando la carga que pueden soportar.
- **Fabricantes de aviónica**: ampliación del mercado de componentes especiales y subsistemas gracias a un volumen de mercado mayor, tiradas de producción mayores y flexibilidad a la hora de satisfacer los requisitos de los clientes.
- **Compañías certificadoras**: creación de un medio para la calificación de componentes y entornos independiente de la aplicación de la nave en cuestión.

La aviónica modular integrada es un claro ejemplo de la tendencia industrial anteriormente comentada. Todavía a dia de hoy se producen avances en este campo.

2.3.2. Sistemas particionados

Como se ha comentado anteriormente en este trabajo, para superar la principal problemática de los SCM —asegurar el aislamiento, tanto espacial (memória, procesador, interfaces, etc.) como temporal (tiempo de uso de los recursos)— se deben aplicar técnicas de particionado. Otra forma de atajar esta problemática es disponiendo al sistema de tanto *hardware* como indique la suma de los requisitos *hardware* de todas las tareas, pero esta solución acarrea costes económicos mucho mayores que el diseño de un sistema particionado. Por lo que las técnicas de particionado suponen una opción mucho más viable en términos económicos. Estas técnicas no consisten en otra cosa que en el uso de un sistema operativo que gestione la memória, el acceso al bus y la asignación de procesadores a tareas (aislamiento espacial) y el uso de algoritmos de planificación que asignen tiempo de uso del *hardware* a las tareas.

Sistemas operativos que lleven a cabo estas funcionalidades hay muchos (RTOS, LITMUS, KVM o IBM System z Hypervisor, por ejemplo). En este trabajo interesa destacar los llamados hipervisores ya que el tipo de sistema operativo utilizado en esta experimentación. Concretamente se hace uso de xtratum, un hipervisor para operar en STR.

Como se expone en [14] un hipervisor, o monitor de máquinas virtuales, es un *software* que ofrece virtualización del *hardware*. La virtualizacón del *hardware* consiste en la abstración del *hardware* de cara a las tareas, consiguiendo que las tareas cuenten con recursos *hardware* que pueden concretarse en cualquiera de los recursos que ofrezca el dispositivo en cuestión. La virtualización del *hardware* permite a un dispositivo ejecutar varios entornos de operabilidad mientras se comparten los núcleos de procesamiento, la memória y los periféricos. Existen dos tipos de hipervisores: los que se ejecutan directamente sobre el procesador, sin un sistema operativo por debajo mediando el uso de los recursos, o los que se ejecutan dentro de un sistema operativo.

XtratuM es un hipervisor que se ejecuta directamente sobre el procesador. Como se puede ver en [15], XtratuM emplea el concepto de partición para referirse a un entorno de ejecución (recursos *hardware*) virtual que será ejecutado sobre el hipervisor. El código que se ejecute dentro de una partición debe ser escrito de acuerdo a los recursos ofrecidos por Xtratum. Una partición puede contener:

- Una **aplicación** compilada para ser ejecutada directamente sobre el *hardware* en cuestión.

- Un **sistema operativo de tiempo real** con sus correspondientes aplicaciones.
- Un **sistema operativo de propósito general** con sus correspondientes aplicaciones.

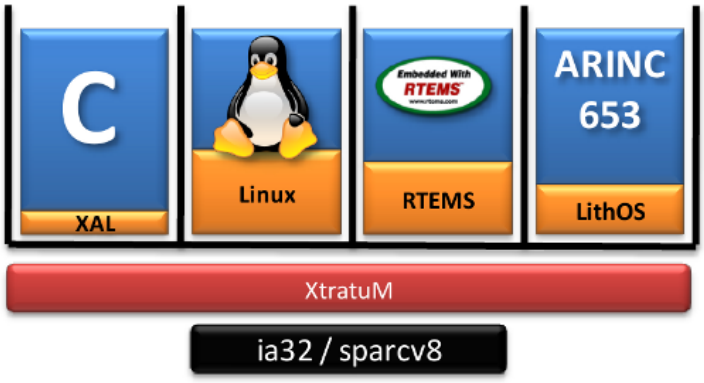

Figura 2.11: Diferentes tipos de particiones ejecutadas sobre XtratuM

Es a través de las particiones que XtratuM ofrece aislamiento espacial y temporal:

- **Aislamiento espacial fuerte**: mientras que XtratuM se ejecuta usando el modo privilegiado del procesador (supervisor), las particiones se ejecutan usando el modo de usuario. Cada partición es ubicada un espacio de direcciones de memória exclusivo para ella. Las particiones solo tienen acceso al espacio de memória que XtratuM les ha proporcionado (previa configuración del sistema). Además, XtratuM ofrece un sistema de etiquetado de particiones con el que dispone a las particiones de según que utilidades del hipervisor dependiendo de las etiquetas que tenga la partición: por ejemplo, todos los *hypercalls* de gestión del sistema solo pueden ser utilizados por particiones etiquetadas como *system partition*.
- **Aislamiento temporal**: XtratuM otorga, en cada momento, acceso a los recursos del sistema según haya sido configurado su planificador fijo cíclico.

2.4 Arquitecturas multinúcleo

Un procesador es un circuito integrado compuesto por transistores. Se le conoce como unidad central de procesamiento porque ocupa un rol central en los sistemas informáticos (junto con la memória principal y los periféricos): llevar a cabo las operaciones de los programas que se ejecután sobre el.

Un procesador ofrece un conjunto de operaciones que puede ejecutar conocido como juego de instrucciones. También dispone al programador de diferentes formas de acceder a la memória principal conocidas como modos de direccionamiento. A su vez, cuenta con niveles de privilegios que permiten ejecutar subconjuntos de instrucciones llamados modos de ejecución. Todas estas características y más están (número de registros, flujo de datos, etapas de las instrucciones, etc.) están definidas por la arquitectura que implementa el procesador.

La arquitectura de un procesador es el conjunto de características que definen como va a operar este, lo que trasciende en que *software* se va a poder ejecutar sobre el y con que componentes podrá interactuar. Existen multitud de arquitecturas de procesadores, cada una especializada en uno o varios aspectos (computación paralela, bajo consumo

energético, etc.). En este apartado se van a comentar las características más relevantes de dos de las arquitecturas de procesadores más ampliamente utilizadas: x86 y ARM.

La arquitectura x86 merece se analiza porque es la más utilizada (aunque principalmente se utilice en computadores de propósito general). La arquitectura ARM es estudiada en este trabajo porque es la que implementa el procesador utilizado para realizar los experimentos que se proponen más adelante.

2.4.1. Arquitectura x86

En términos de ordenadores que usan esta arquitectura [18] y la potencia de cómputo total que ofrecen, x86 es la arquitectura más privilegiada del mundo. Los procesadores que implementan esta arquitectura forman una enrome lista que comienza desde los modelos 8086 y 8088, ambos de 16 bits, hasta llegar a la décima generación de la familia de procesadores Intel®Core™, con sus modelos i3, i5 e i7, ya de 64 bits, con transistores de 14 nanómetros; pasando por las familias de procesadores Intel®Xeon®, Intel®Atom™, Intel®Pentium®, etc. Cada familia con sus diferentes modelos.

Principalmente, x86 es una arquitectura diseñada para dar soporte, en cada nuevo modelo que la implementa, al software diseñado para los modelos anteriores (*backward compatibility*). Existen dos variantes de la arquitectura, diferenciadas principalmente por el tamaño de palabra (cantidad máxima de memória utilizable en una sola instrucción), siendo una la IA-32 (palabra de 32 bits) y la otra Intel®64 (palabra de 64 bits). Se trata de una arquitectura *CISC* (*Complex Instruction Set Computer*), lo que quiere decir que ofrece un juego de instrucciones donde hay instrucciones que ejecutan varias instrucciones "fundamentales"(carga, almacenamiento, operaciones aritméticas, etc.). Por ejemplo, una instrucción *CMOVE*(*Conditional MOVe if Equal*) realiza un movimiento de datos entre memoria y registros si se cumple una condición de igualdad, lo que conlleva dos operaciones: comprobar si se cumple la igualdad y a continuación mover los datos en función del resultado obtenido. Otro ejemplo es una instrucción *BTS* (*Bit Test and Set*) que comprueba el valor de un bit y lo establece a un valor deseado en función del resultado de la comprobación. En total, x86 ofrece unas 1500 instrucciones diferentes [19] (contando las diferentes extensiones existentes que no llevan todos los modelos que implementan la arquitectura).

Los procesadores que implementan esta arquitectura son máquinas *little endian*, lo que significa que los *bytes* de una palabra se almacenan dejando en la dirección más baja el *byte* menos significativo.

Tres son los modos de ejecución que ofrece la arquitectura, de los que depende las instrucciones disponibles y otras características:

- **Modo *Real-address***: presente cuando se enciende el sistema o cuando se restablece. Implementa el entorno de programación del procesador Intel 8086 con extensiones para, por ejemplo, poder cambiar a modo protegido o de gestión del sistema.
- **Modo protegido**: es el modo nativo del procesador. Permite ejecutar *"real-address mode"* 8086 *software* en un entorno multitarea protegido.
- **Modo de gestión del sistema**: permite al sistema operativo o a un programa implementar, de manera transparente, funciones como la gestión de la energia o de la seguridad del sistema. Se entra en este modo cuando se recibe una interrupción al *pin* correspondiente o cuando se recibe una señal del controlador programable de interrupciones avanzado. Cuando se cambia a este modo, el procesador guarda el contexto actual y cambia a un espacio de direcciones diferente, volviendo al contexto anterior una vez abandonado este modo.

Existe otro modo para Intel®64, IA-32e, con dos submodos más:

- **Modo compatibilidad**: permite ejecutar código de las variantes de 16 y 32 bits de la arquitectura sin recompilarlo, aunque no toda aplicación funcionaria en este modo (concretamente, aplicaciones pensadas para el modo virtual 8086 o que gestionen las tareas del hardware). Este modo debe ser habilitado por el sistema operativo. Es un modo similar al modo protegido de IA-32, con un espacio de direccionamiento igual que IA-32 (4GB de espacio lineal y hasta 64GB de espacio físico).
- **Modo 64 bits**: permite el uso de direcciones de 64 bits. Habilita el uso de registros de propósito general de 64 bits y 8 registros extra de propósito general y de operaciones *SIMD* (*Simple Instruction Multiple Data*, accesibles con operaciones con un prefijo especial (REX). Usando este prefijo se habilita el uso de operadores de 64 bits (si no se usa el prefijo los operadores son de 32 bits). El modo debe ser activado por el sistema operativo.

El entorno de programación que ofrece la arquitectura se compone de varios elementos. En primer lugar, el espacio de direccionamiento es de 4GB de direccionamiento lineal y 64GB de memoria física en el caso de IA-32. Intel®64, por su parte, ofrece hasta 2^{64} *bytes* de direccionamiento lineal y hasta 2^{46} *bytes* de memoria física. También ofrece una gran cantidad de registros: de propósito general, de segmentación de la memoria, de estado y control del sistema (*EFLAGS* en IA-32 y *RFLAGS* en Intel®64), de puntero a instrucción, de la unidad de coma flotante x87 (datos, control, estado, puntero a instrucción, puntero a datos, configuración y código de operación), de las unidaded de operaciones *single-instruction, multiple-data* (*SIMD*) MMX (64 bits), XMM (128 bits) e YMM (256 bits), de direcciones límite de *buffers* en memoria, de configuración y estado del modo *memory protection extension* (*MPX*).

Los recursos más interesantes para programar que ofrece la arquitectura son los siguientes:

- **Registros básicos**: IA-32 ofrece ocho registros de propósito general, seis registros de segmentos, un registro de configuración (*EFLAGS*) y un registro para el puntero a instrucción, que conforman el entorno mínimo necesario para ejecutar un conjunto básico de instrucciones (aritmética con enteros, flujo del programa, etc.). Intel®64 ofrece por su parte el doble de registros de propósito general, siendo estos de 64 bits, al igual que el puntero a instrucción y el registro de configuración, que pasa a llamarse *RFLAGS*.
- **Registros límite**: cuatro registros de 64 bits para almacenar las direcciones inicial y final de *buffers* de memoria.
- ***BNDCFGU* y *BNDSTATUS***: registros de configuración y de estado de las instrucciones *MPX*.
- **Pila**: implementada, con un registro de puntero (que es de 64 bits en Intel®64) y con un registro de selección de segmento, para dar soporte a la ejecución de rutinas y procedimientos y al paso de parametros entre estos.
- **Puertos de entrada/salida**: a través de un espacio de direccionamiento específico, o a través de puertos mapeados en el espacio de direccionamiento de memoria principal, se da soporte a la interacción con periféricos.
- **Registros de control**: cinco registros que indican el modo de ejecución y las características de la tarea en proceso. En Intel®64 estos registros son de 64 bits.

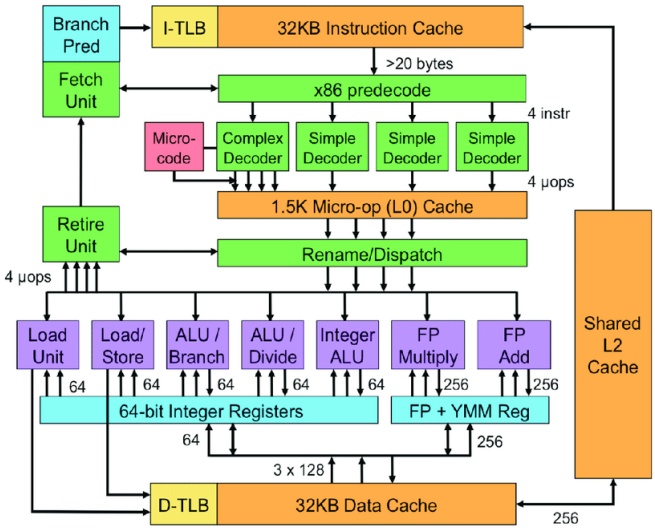

Figura 2.12: Implementación de la arquitectura x86

- **Registros de gestión de la memoria**: se trata de registros que almacenan la dirección de memoria de las tablas de descriptores y el espacio que ocupan, que se usan para implementar la segmentación de la memoria. Estos registros son el *GDTR*
- **Registros de depuración**: ocho registros para establecer puntos de parada de programas y monitorizar y controlar la depuración de programas. En Intel®64 estos registros son de 64 bits.
- **Contadores de monitorización del rendimiento**: permiten monitorizar eventos del procesador (Accesos al primer nivel de la memoria cache, accesos al bus de datos, ciclos de reloj realizados, etc.)

Los contadores de monitorización del rendimiento son clave para este trabajo, ya que es mediante ellos que se puede obtener información sobre el avance que tiene un programa en tiempo de ejecución. Con ellos se pueden contabilizar, de manera aproximada, eventos como los ciclos de reloj, instrucciones de salto en las que se ha efectuado el salto de dirección, instrucciones de carga bloqueadas por instrucciones de almacenamiento, fallos en la cache de segundo nivel al pedir nuevas instrucciones, hasta llegar a medir eventos tan peculiares como el número de microoperaciones ejecutadas por el detector de bucles.

El procesador Intel®i7-11850HE es una implementación comercial de la arquitectura x86 (una de las más actuales). Está fabricada con transistores de diez nanómetros de ancho. Tiene ocho núcleos de procesamiento, dieciséis hilos de ejecución (dos por cada núcleo de procesamiento), una frecuencia de reloj máxima de cuatro coma siete gigahercios (cuatro mil setecientos millones de operaciones por segundo), veinticuatro *megabytes* de memória cache, hasta ciento veintiocho *gigabytes* de memória principal, etc.

Por último, los ámbitos de uso de la arquitectura x86 son muchos, comenzando desde ordenadores personales, tanto pórtatiles como de sobremesa (quitando los teléfonos inteligentes), hasta llegar a los supercomputadores de la lista TOP500, pasando por estaciones de trabajo y servidores (*midrange computers*). Donde no tiene tanta presencia esta arquitectura, a día de hoy (porque en los años setenta y ochenta si que la tuvo), es en los SE.

CAPÍTULO 3
Experimentación

Como se ha comentado en la introducción, en este capítulo se muestra: el modelo con el que se parametrizan las diferentes tareas que componen los experimentos de este trabajo, que es el modelo utilizado por Vestal en el artículo que se acuñó el término SCM; las herramientas de trabajo empleadas en los diferentes experimentos (el hardware, en este caso la placa de desarrollo Cora Z7, y el software, que viene a ser todo el entorno de programación de Xtratum, Xtratum mismo y las tareas ejecutadas); los escenarios de trabajo, que son planificaciones de ejecución concretas específicamente diseñadas para medir la interferencia entre las tareas; y, para finalizar, los resultados obtenidos y un controlador que asegure la ejecución dentro de plazo de los procesos críticos.

3.1 El modelo de Vestal

Vestal propone un modelaje de tareas que deja ver las características relevantes de las tareas que componen un sistema de criticidad mixta, clarificando y facilitando el análisis del mismo. Este modelo esta compuesto por k, que es el conjunto de n tareas t_i (para $i = 1..n$) que se van a ejecutar dentro del sistema. Cada t_i está compuesta por: un T_i, que es el periodo de ejecución (tiempo entre una ejecución y la sucesiva de t_i); un D_i, que es el plazo que se le asigna, dentro del cual se debe llevar a cabo la ejecución de t_i (por lo general $D_i = T_i$) para que el sistema funcione como se espera y, por ejemplo, no hayan fallos catastróficos; por un C_i, que es el peor tiempo de ejecución muestreado de t_i; y por un L_i, que indica el nivel de criticidad de t_i (puede tener un valor de entre los comprendidos en l, que es el conjunto de posibles niveles de criticidad, e.g. $\{Baja, Media, Alta\}$).

Por lo tanto, k es un conjunto de t_i $\{t_0, t_1..t_n\}$ al mismo tiempo que cada t_i es un conjunto compuesto por un T_i, un D_i, un C_i y un L_i ($t_i = \{T_i, D_i, C_i, L_i\}$).

En los escenarios de ejecución que componen los experimentos se ha observado como ha variado C_i de un escenario a otro. Los cambios de C_i son la medición de la interferencia. l está compuesto por dos nieveles en este trabajo: criticidad alta y baja. El objetivo último del control de la ejecución que aquí se hace es lograr que cualquier t_i con $L_i = Alta$ nunca llegue a $C_i > D_i$. Las tareas críticas deben ejecutarse dentro de plazo.

3.2 Herramientas de trabajo

Para la experimentación de este trabajo se ha utilizado una placa de desarrollo Digilent®Cora Z7. Esta cuenta con un chip Xilinx Zynq™XC7Z010-1CLG400C, compuesto por un procesador ARM Cortex A9 y una *FPGA* Artix A7 (*Field-Programmable Gate Array*).

El Cortex A9 tiene una frecuencia máxima de reloj de 667MHz. La placa cuenta también con 512 MB de memoria RAM DDR3 a la que se accede a través de un bus de 16 bits. El acceso a memoria, por lo tanto, se puede llevar a cabo a una velocidad de transferencia de hasta 1050Mbps. Además tiene una interfaz JTAG para programarla y depurarla, dos botones, dos LED RGB y varios puertos de entrada/salida de propósito general.

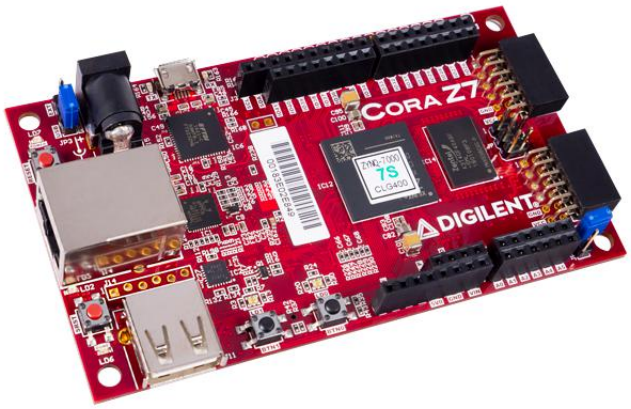

Figura 3.1: Digilent®Cora Z7

El Cortex A9 de esta placa implementa la unidad de monitorización del rendimiento própia de esta arquitectura (hay implementaciones de la arquitectura que no cuentan con la unidad de monitorización del rendimiento). A través de ella se puede contabilizar el número de instrucciones ejecutadas, la cantidad de ciclos de reloj que han sucedido o los fallos de cache incurridos. En la propuesta de controlador de la ejecución de este trabajo se utiliza la unidad de monitorización del rendimiento para conocer el progreso de las tareas críticas.

Del lado del *software* se ha utilizado el hipervisor ("sistema operativo") Xtratum. Este es un *software* de virtualización del *hardware*, certificado para su uso en aeronáutica, diseñado para funcionar en sistemas embebidos de tiempo real seguros y eficientes. Ejerce de sistema operativo en lo que se refiere a la gestión de la memoria o el control de la ejecución, pero se diferencia de este en lo que se refiere a la unidad mínima de *software*, que en este caso son particiones (en lugar de procesos, que son en un sistema operativo). Una de sus características principales es el fuerte aislamiento temporal y espacial que ofrece, siendo idóneo para sistemas particionados. Cuenta con un planificador cíclico fijo (que admite varias planificaciones distintas para distintas situaciones). Xtratum virtualiza el *hardware* para que lo usen las particiones. Una partición puede ser un sistema operativo de tiempo real con sus correspondientes aplicaciones, un sistema operativo de propósito general o bien una aplicación ejecutándose, en modo usuario, directamente sobre el *hardware*. Las particiones gestionadas por Xtratum se ejecutan directamente sobre el *hardware*, permitiendo así, entre muchas otras ventajas, la reutilización de aplicaciones antiguas ya certificadas (solo se tendrían que cambiar las llamadas al sistema para que funcionasen sobre Xtratum), aparte de que genera muy poco sobrecosto con los cambios de contexto (cambio de la tarea que hace uso del *hardware*).

Para utilizar Xtratum se ha contado con una máquina virtual, con Xubuntu (linux) como sistema operativo, dotada de todas las herramientas necesarias para generar ejecutables compatibles con la Cora Z7 (compilador, *sdk*, editor de textos, código fuente de Xtratum, Makefiles para compilar Xtratum, etc.). Con un solo coman-

```
xal-arm-examples
├── cora_ps7_init.tcl
├── cora_xsdb.ini
├── example.000
│   ├── example_def.txt
│   ├── gbatch
│   ├── Makefile
│   ├── partition0.c
│   ├── resident_sw.elf
│   ├── tbatch
│   └── xm_cf.arm.xml
└── example.001
```

Figura 3.2: Escenario de ejecución

do \foreign{english}{make} se puede generar un binario ejecutable en la Cora Z7. La máquina virtual viene con una carpeta llamada xal-arm-examples-cora. Dentro de esta carpeta se encuentran varios directorios, cada uno correspondiente a un escenario de ejecución en el que se usa una funcionalidad de Xtratum (son ejemplos de uso del entorno de programación). Uno de los ejemplos de uso es el clásico hello-world. Cada escenario de ejecución se compone de:

- Un **Makefile** encargado de generar el ejecutable que se introduce en la placa de desarrollo Cora Z7.
- Unos archivos .c correspondientes a las **particiones** que se van a ejecutar. Algunos .c son librerías auxiliares, y no particiones directamente, como es el caso de pmc.c, que provee de funciones para manejar la unidad de monitorización de rendimiento.
- Un **archivo de configuración** .xml de Xtratum. Es en este archivo donde se configura el planificador cíclico fijo, donde se indica el espacio de memoria RAM que se le asigna a cada partición, donde se declaran los puertos serie, donde se le indica a Xtratum las características del *hardware*, como la frecuencia de reloj, la cantidad de memoria RAM, el número de núcleos de procesamiento, etc.
- Un **resident-sw**, que es el archivo resultante de ejecutar el Makefile. Se trata de un ejecutable que contiene Xtratum, adecuadamente configurado, y las particiones.
- Un ***script*** de comandos para la consola xmd o xsdb. Xmd y xsdb son consolas de comandos para interactuar con los dispositivos diseñados por Xilinx. Es a través de estas consolas, gracias a la ayuda del *script* mencionado, que se configura la Cora y se introduce el resident-sw en ella.

Para poder generar un archivo binario ejecutable que contenga a Xtratum (aparte de las particiones) hace falta tener compilado Xtratum para la arquitectura del procesador sobre la que se va a programar. Los pasos a seguir para compilar Xtratum son los siguientes:

1. Se debe acudir a la carpeta donde está el código fuente de Xtratum. En esta máquina virtual el código fuente se encuentra en el directorio ~/xm-arm-distros/xm-arm-2.0.5.base/xm-src (2.0.5 es la versión de Xtratum utilizada; arm porque es la arquitectura del Cortex A9) por lo que llegar hasta ella es ejecutar el comando cd ~/xm-arm-distros/xm-arm-2.0.5.base/xm-src.
2. En la carpeta donde se encuentra el código fuente de Xtratum hay un Makefile con varios comandos para ejecutar. En primer lugar se debe ejecutar make distclean (comando que borra todos los archivos de configuración y los binarios generados anteriormente).
3. A continuación se establece como archivo de configuración (xmconfig) de Xtratum el que contenga la terminación "arm" en su nombre: cp xmconfig.arm xmconfig.
4. Lo siguiente es ejecutar el comando make menuconfig, que abre un menú en la consola de comandos para establecer los valores de ciertos parámetros. Concretamente, para que Xtratum funcione en la Cora Z7, se debe habilitar la memoria *cache* de segundo nivel, con una pólitica de escritura *write-through no-allocate* y habilitar el comunicador serie UART0 a una frecuencia de 100KHz.

5. Una vez finalizado el menú de configuración, solo quedan ejecutar los comandos `make` y `make distro`, en ese orden, y ya se puede contar con una distribución de Xtratum lista para ser usada. Esta distribución se encuentra dentro de un directorio, recientemente creado, `/opt/sdk-xm-arm-205-cora`, al que hay que ponerle como dueño al usuario que más adelante programe y compile los escenarios de ejecución.

Con Xtratum ya compilado, lo que queda es generar los ejecutables (particiones) correspondientes a los diferentes escenarios de trabajo. La tarea que se ha programado para este experimento es una multiplicación de matrices de tamaño $n * n$. Para generar las matrices de tamaño n se ha programado un *script* en python3. Este *script* genera un archivo llamado `big_matrix.h` que contiene dos matrices bidimensionales de tipo entero y una variable, también de tipo entero, que almacena el valor de n. El programa que lleva a cabo la multiplicación de matrices es un binario compilado a raíz de un archivo `.c` llamado `mat_mul.c`. Este programa multiplica las matrices de `big-matrix.h` con un algoritmo clásico de coste temporal n^3, imprimiendo por pantalla, finalmente, el valor de cada posición de la matriz resultante. Además, `mat_mul.c` hace uso de la unidad de monitorización del rendimiento para conocer el número de instrucciones que conlleva su ejecución, el número de accesos a memoria principal a lo largo de su ejecución y el numero de ciclos de reloj sucedidos en su ejecución. Aparte de medirse el tiempo de ejecución en microsegundos a través de las interrupciones del reloj realizadas por Xtratum. El tiempo se mide con llamadas a Xtratum (*hypercalls*) `XM_get_time (XM_EXEC_CLOCK, &tInicial)` y la unidad de monitorización del rendimiento se maneja a través de la librería `pmc.c`

A través del archivo `xm_cf.arm.xml` se realiza la planificación de la ejecución de las particiones de Xtratum (planificador cíclico fijo). Los diferentes experimentos, expuestos en la siguiente sección, consisten en diferentes planificaciones de la ejecución, haciendo coincidir más o menos tiempo la ejecución de dos particiones (cada partición siendo una multiplicación de matrices autónoma) para observar la interferéncia que se genera.

Por último, para insertar los ejecutables en la Cora y ponerla en marcha se utiliza la consola `xmd`, una herramienta ofrecida por Xilinx®para programar y depurar dispositivos de la compañía en cuestión. Ejecutando el comando `xmd -tcl ../cora_xmd.ini` se lleva a cabo la programación de la Cora. Para poder leer el *output* de las particiones se utiliza la aplicación Cutecom, que permite leer y escribir a través de puertos serie UART.

3.2.1. pmc.c

Esta librería ofrece un conjunto de constantes y de funciones que permiten el uso de la unidad de monitorización del rendimiento. Las diferentes constantes, declaradas en el archivo `pmc.h`, son los códigos que se deben introducir en los registros de la unidad de monitorización del rendimiento para poder medir los eventos deseados:

```
1 #define ARMV7_PERFCTR_PMNC_SW_INCR          0x00
2 #define ARMV7_PERFCTR_L1_ICACHE_REFILL      0x01
3 #define ARMV7_PERFCTR_ITLB_REFILL           0x02
4 #define ARMV7_PERFCTR_L1_DCACHE_REFILL      0x03
5 #define ARMV7_PERFCTR_L1_DCACHE_ACCESS      0x04
6 [...]
```

Ejemplos de constantes de pmc.h

Las funciones que ofrece la librería son las siguientes:

```
[...]
void initPMU();
void allocCounterPMU(xm_u32_t r, xm_u32_t event);
void enablePMR(int counter);
void enableCCNT();
xm_u32_t readPMR(int r);
xm_u32_t readCCNT();
void clearAllPMR();
void clearCCNT();
void disablePMU (void);
```

Funciones de pmc.c declaradas en pmc.h

`initPMU ()` inicializa la unidad de monitorización del rendimiento (configura el registro de control habilitando los contadores, restableciendo todos los contadores, restablece el contador de ciclos, etc; borra los desbordamientos, habilita el acceso al modo usuario y deshabilita las interrupciones por desbordamientos), `enablePMR (int r)` habilita el registro de conteo indicado en `r`, `enableCCNT ()` habilita el contador de ciclos (registro especial que solo puede contar ese evento en concreto), `allocCounter (xm_u32_t r /*REGISTRO*/,xm_u32_t event)` configura el registro indicado en `r` con el código de evento indicado en `event` para posteriormente leer el número de veces que ha sucedido tal evento con la función `readPMR (int r)` indicando en `r` el registro que se desea leer, `readCCNT ()` permite leer el registro de conteo de ciclos de reloj, `clearAllPMR ()` establece a 0 el valor de todos los registros de la unidad de monitorización del rendimiento, `clearCCNT ()` establece a 0 el contador especial de ciclos de reloj y `disablePMU ()` inhabilita el uso de la unidad de monitorización del rendimiento.

```
xm_u32_t readCCNT() {
  xm_u32_t value;
  // Read CPU cycle count from the CCNT Register
  __asm__ __volatile__  ("mrc p15, 0, %0, c9, c13, 0\t\n": "=r"(value));
  return value;
}
```

Funcion de pmc.c que lee el contador específico de ciclos de reloj

En la sección de escenarios de trabajo, donde se muestra el código de `mat_mul.c`, se expone el uso concreto que se le da a esta librería para llevar a cabo la experimentación de este trabajo.

3.2.2. mat_mul.c

La carga de trabajo escogida para este experimento es una multiplicación de matrices. Las matrices se generan con el *script* anteriormente mencionado (`generate_matrix.py`). El el archivo que contiene la multiplicación de matrices es `mat_mul.c`. A raíz de el ejecutable generado tras compilar este archivo se crean las particiones que se ejecutan en los escenarios de trabajo planteados.

Se definen cuatro constantes al comienzo de `mat_mul.c`:

```
#define NEVENTOS 3     //Eventos registrados
#define NMULTS   5000 //Multiplicaciones que se realizan en una iteracion
#define NITER    5     //Iteraciones que se realizan en un plan de ejecucion
#define NPLANES  5     //Planes de ejecuci n que se van a llevar a cabo
```

Constantes de mat$_m ul.c$

La configuración que aquí se muestra es la que se ha utilizado en la experimentación. Con una multiplicación de matrices de $n * n$ donde $n = 20$, registrando tres eventos diferentes (que se muestran a continuación), realizando cinco mil veces la misma multiplicación por iteración, con cinco iteraciones por plan de ejecución, con cinco planes de ejecución (que se muestran en la sección de Escenarios de trabajo) se logra caracterizar las particiones de acuerdo al modelo de Vestal, más aparte se logra generar diferentes niveles de interferencia y medir la misma.

En primer lugar se configura la unidad de monitorización del rendimiento

```
/* Configuracion de la UMR */
initPMU ();

allocCounterPMU (0, ARMV7_A9_INST_EXEC);
allocCounterPMU (1, ARMV7_PERFCTR_CLOCK_CYCLES);
allocCounterPMU (2, ARMV7_PERFCTR_MEM_READ);

for (r = 0; r < NEVENTOS; r++) {
    enablePMR(r);
}
```

Configuración de la unidad de monitorización del rendimiento

Los eventos que se miden son las instrucciones ejecutadas (`ARMV7_A9_INST_EXEC`), los ciclos de reloj (`ARMV7_PERFCTR_CLOCK_CYCLES`) y las lecturas de memoria principal (`ARMV7_PERFCTR_MEM_READ`). Al final de cada iteración se recogen los valores de los registros, junto al tiempo de ejecución calculado mediante la *hypercall* `XM_get_time(XM_EXEC_CLOCK, &tInicial)` (Xtratum ofrece dos parámetros posibles para esta *hypercall*: XM_EXEC_CLOCK y XM_HW_CLOCK. El primero mide el tiempo transcurrido dentro del T_i mientras que el segundo mide el tiempo absoluto transcurrido entre llamada y llamada, independientemente de que la partición haya estado ejecutándose o no). Los valores recogidos se almacenan en un vector bidimensional de dimensiones $NPLANES * (NEVENTOS + 1)$ llamado eventos:

```
/* Registro del instante final y calculo del tiempo transcurrido desde el
   instante inicial */
XM_get_time (XM_EXEC_CLOCK, &tFinal);
eventos [plan] [0] += (xm_u32_t)(tFinal-tInicial);

/* Lectura de los registros de eventos */
for (i = 1; i <= NEVENTOS; i++)
    eventos [plan] [i] += readPMR (i - 1);
```

Recopilación de los resultados

Por cada plan de ejecución se almacenan los valores medios de cada evento registrado:

```
for(i = 0; i <= NEVENTOS; i++){
    eventos [plan] [i] /= NEVENTOS;
}
```

Cálculo de las medias

Finalmente, cuando ya se han ejecutado todas las iteraciones de cada plan de ejecución, se imprimen por pantalla los valores de los contadores de eventos almacenados en cada plan de ejecución:

```
for (j = 0; j < NPLANES; j++){
    PRINT("Datos del plan %d -> [Tiempo medio de ejecucion = %d
                                | Media de instrucciones ejecutadas = %d
```

```
                                | Media de ciclos de reloj = %d] \n",
                                j, eventos [j] [0], eventos [j] [1], eventos [j]
                                    [2], eventos [j] [3], eventos [j] [4]);
}

/* RESULTADO DE LA IMPRESION POR PANTALLA */
[P0] Datos del plan 0 -> [Tiempo medio de ejecucion = 384492 ms | Media de
    instrucciones ejecutadas = 4594223 | Media de ciclos de reloj = 39372706
```

Muestra de los valores recogidos en cada plan de ejecución

La multiplicación de matrices se lleva a cabo mediante un algoritmo clásico de coste N^3. Este consiste en multiplicar cada fila de la primera matriz por cada columna de la segunda matriz, calculando así los valores, uno a uno, de la matriz resultante:

```
/* Multiplicacion de la matriz */
while (row_A < NCOLS) {
    while (col_B < NCOLS) {
        res = 0;
        for (i = 0; i < NCOLS; i++) {
            res += MATRIX_A[row_A][i] * MATRIX_B[i][col_B];
        }

        res_matrix [row_A] [col_B] = res;
        col_B ++;
    }

row_A ++;
}
```

Algoritmo de multiplicación de matrices de mat$_m$*ul.c*

El programa ha sido diseñado para ejecutar todos los planes de ejecución en un solo lanzamiento, de modo que cada `NITER` iteraciones la partición cambia de planificación, hasta haber ejecutado las `NPLANES` planificaciones existentes, momento en el que se finaliza la ejecución de la partición mediante la *hypercall* `XM_halt_partition (XM_PARTITION_SELF)`. En la siguiente sección se muestra la configuración de cada escenario de trabajo, incluyendo cada planificación de la ejecución.

```
/* Cambio del plan de ejecucion */
if (iter == NITER && plan < NPLANES){
[...]
    plan ++;
    iter = 0;

    /* Si va a finalizar la ejecucion se imprimen todos los resultados */
    if (plan == NPLANES){
    [...]

       PRINT("Ejecucion finalizada\n");
       XM_halt_partition(XM_PARTITION_SELF);
    }

    XM_switch_sched_plan (plan, &planActual);
    PRINT("Cambio a plan %d\n", plan);
}
```

Cambio de la planificación de la ejecución

3.3 Escenarios de trabajo (experimentos)

Como se ha comentado anteriormente, uno de los dos objetivos de este trabajo es medir la interferencia generada por la ejecucón en paralelo de varios procesos en el mismo procesador multinúcleo (cada proceso en un núcleo diferente). La interferencia se mide como la diferencia en el valor de C_i, para una tarea dada t_i, entre escenario y escenario. Esta se mide tanto en milisegundos como en ciclos de reloj completados. Para ello, `mat_mul.c` configura los registros de la unidad de monitorización del rendimiento del Cortex A9, para medir los ciclos de reloj, las instrucciones ejecutadas y el tiempo en microsegundos de su ejecución, y en sus últimas instrucciones muestra por pantalla el valor de los registros, permitiendo visualizar los resultados del conteo.

Si bien `mat_mul.c` se encarga de llevar a cabo la multiplicación de matrices y la medida de los parámetros del modelo, Xtratum se encarga de ejecutar cada partición en el momento que se le indique, gracias a su planificador cíclico fijo. Para lograr un comportamiento concreto de Xtratum este se debe configurar. La configuración de Xtratum se realiza en el archivo `xm_cf.arm.xml`. En este archivo se declaran los recursos *hardware* y *software*. Para este experimento la configuración del *hardware* es la siguiente:

```
<HwDescription>
    <MemoryLayout>
        <Region type="rom" start="0x0" size="1MB"/>
        <Region type="sdram" start="0x00100000" size="511MB"/>
    </MemoryLayout>
    <ProcessorTable>
        <Processor id="0" frequency="667MHz">
            [...]
        </Processor>
        <Processor id="1" frequency="667MHz">
            [...]
        </Processor>
    </ProcessorTable>
    <Devices>
        <Uart id="0" baudRate="115200" name="Uart"/>
    </Devices>
</HwDescription>
    <XMHypervisor console="Uart">
        <PhysicalMemoryArea size="512KB"/>
    </XMHypervisor>
```

Configuración de los recursos hardware

Aquí se le indica que cuenta con 512 MB de memoria principal, de los cuales dedica 1MB a memoria de tipo ROM, y los 511 MB restantes los dedica a dar espacio de direccionamiento a las particiones (memoria de tipo sdram), entendiendo por espacio de direccionamiento la porción de memoria RAM con la que cuenta una partición. La memoria sdram comienza justo donde acaba la memoria de tipo rom (rom comienza en la 0x0 y acaba en la 0x000fffff y sdram comienza en la 0x00100000 y acaba en la 0x1fffffff). También se le indica que cuenta con dos procesadores que trabajan a una frecuencia de 667 MHz (en los puntos suspensivos es donde se ubica la planificacón de la ejecución, que se muestra más adelante en esta sección). Por último, se le indica que cuenta con un puerto serie UART, trabajando a una frecuencia de 115200 baudios, con una memoria de 512 KB.

Por otro lado, la configuración *software* (quitando de las planificaciones de ejecución, que son expuestas más adelante en esta sección) es la siguiente:

```
<PartitionTable>
    <Partition id="0" name="partition0" flags="boot system" console="Uart">
        <PhysicalMemoryAreas>
            <Area start="0x10000000" size="512KB" flags = "uncacheable"/>
        </PhysicalMemoryAreas>
    </Partition>
    <Partition id="1" name="partition1" flags="boot system" console="Uart">
        <PhysicalMemoryAreas>
                <Area start="0x2000000" size="512KB" flags = "uncacheable"/>
        </PhysicalMemoryAreas>
    </Partition>
</PartitionTable>
```

Configuracide las particiones de Xtratum

Aquí se le indica (a Xtratum) que hay dos particiones a ejecutar, llamadas partition0 y partition1. Cada una contando con 512 KB de memoria RAM (partition0 comenzando desde la dirección 0x10000000 y partition1 desde la 0x2000000). Ambas utilizan el puerto serie UART y están etiquetadas como particiones boot (etiqueta que le indica a Xtratum que la partición se ejecutará tras cualquier reinicio de la ejecución del sistema) y system (etiqueta que le indica a Xtratum que esta partición puede ejecutar *hypercalls* de gestión del sistema, como por ejemplo `XM_switch_sched_plan ()`, que es la función que cambia la planificación de la ejecución en caso de haber varias, como es este caso). La etiqueta `uncacheable` de la memoria de cada partición indica que no hacen uso de la memoria cache (configurado así a propósito para aumentar los tiempos de ejecución, acentuando así los efectos de la interferéncia).

Por último, a Xtratum se le debe indicar la planificación de la ejecución (cuando y durante cuanto tiempo se ejecuta cada partición). Xtratum admite múltiples planificaciones fijas. En este experimento se hace uso de esta capacidad para, en un solo archivo de configuración, configurar todos los escenarios de ejecución necesarios para medir la interferencia ante diferentes grados de coincidencia en el tiempo de ejecución. Habiendo ejecutado la multiplicación de matrices de forma aislada se sabe que, aproximadamente, $Ci = 380$ ms, por lo tanto, para plantear cinco escenarios de ejecución donde haya respectivamente 0 % de interferencia, 25 % de interferencia, 50 % de interferencia, 75 % de interferencia y 100 % de interferencia, se calcula el instante en el que debe comenzar la partition1: si $Ci = 380$ ms, el 25 % de 380 es 76, por lo que $380 - 76 = 304$ es el instante en el que debe comenzar partition1 para generar una interferencia, a priori, del 25 %, y así sucesivamente con los diferentes escenarios (por simplicidad se ha tomado el valor de C_i como 400 ms).

```
[...]
<CyclicPlanTable>
    <Plan id="0" majorFrame="1000ms">
        <Slot id="0" start="0ms" duration="400ms" partitionId="0"/>
    </Plan>
    <Plan id="1" majorFrame="1000ms">
        <Slot id="0" start="0ms" duration="400ms" partitionId="0"/>
    </Plan>
    <Plan id="2" majorFrame="1000ms">
        <Slot id="0" start="0ms" duration="450ms" partitionId="0"/>
    </Plan>
    <Plan id="3" majorFrame="1000ms">
        <Slot id="0" start="0ms" duration="500ms" partitionId="0"/>
    </Plan>
    <Plan id="4" majorFrame="1000ms">
        <Slot id="0" start="0ms" duration="600ms" partitionId="0"/>
```

```
    </Plan>
</CyclicPlanTable>
[...]
<CyclicPlanTable>
    <Plan id="0" majorFrame="1000ms">
        <Slot id="0" start="400ms" duration="400ms" partitionId="1"/>
    </Plan>
    <Plan id="1" majorFrame="1000ms">
        <Slot id="0" start="300ms" duration="400ms" partitionId="1"/>
    </Plan>
    <Plan id="2" majorFrame="1000ms">
        <Slot id="0" start="200ms" duration="450ms" partitionId="1"/>
    </Plan>
    <Plan id="3" majorFrame="1000ms">
        <Slot id="0" start="100ms" duration="500ms" partitionId="1"/>
    </Plan>
    <Plan id="4" majorFrame="1000ms">
        <Slot id="0" start="0ms" duration="600ms" partitionId="1"/>
    </Plan>
</CyclicPlanTable>
[...]
```

Planificaciones para la medida de la interferéncia

Para ofrecer una forma más gráfica de ver la planificación se ha realizado la siguiente ilustración, que muestra fielmente los escenarios de ejecución diseñados para este experimento. Las zonas rayadas son los intervalos de tiempo dedicados a la ejecución de cada partición, y las zonas azules lisas son los intervalos de tiempo libres de ejecuciones, dentro de los *major frames* (ranuras de tiempo repetidas cíclicamente). .

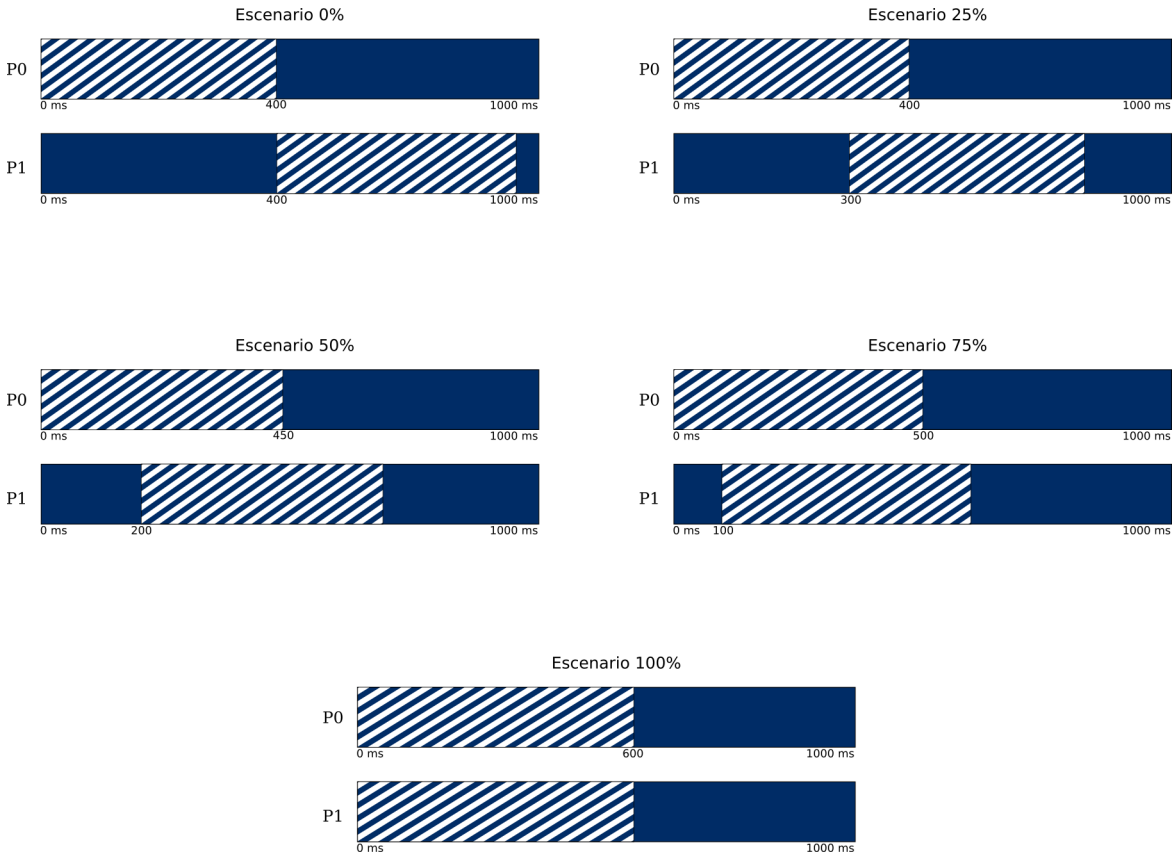

Figura 3.3: Escenarios de planificación ilustrados

3.4 Resultados

Como se ha mostrado en la sección dedicada al archivo `mat_mul.c`, el propio programa recopila todos los datos relevantes para este estudio en la matriz `eventos`. Al finalizar la ejecución de todas las planificaciones programadas `mat_mul.c` muestra por pantalla el contenido de `eventos`, permitiendo así el análisis de los resultados. Tras la ejecución del experimento con la configuración mostrada en la sección anterior (Escenarios de trabajo), `mat\-mul.c` imprime por pantalla lo siguiente:

```
[P0] Datos del plan 0 -> [Tiempo medio de ejecucion = 386350 | Media de
    instrucciones ejecutadas = 22971284 | Media de ciclos de reloj = 197811708]
[P0] Datos del plan 1 -> [Tiempo medio de ejecucion = 386344 | Media de
    instrucciones ejecutadas = 22973119 | Media de ciclos de reloj = 197809276]
[P0] Datos del plan 2 -> [Tiempo medio de ejecucion = 406331 | Media de
    instrucciones ejecutadas = 22971555 | Media de ciclos de reloj = 208043404]
[P0] Datos del plan 3 -> [Tiempo medio de ejecucion = 482548 | Media de
    instrucciones ejecutadas = 22984615 | Media de ciclos de reloj = 247067050]
[P0] Datos del plan 4 -> [Tiempo medio de ejecucion = 559215 | Media de
    instrucciones ejecutadas = 23020872 | Media de ciclos de reloj = 286324341]
```

El plan 0 corresponde al plan con un 0 % de interferencia y el plan 4 al plan con un 100 % de interferencia. El plan 0 revela las características de las particiones cuando estas se ejecutan aisladamente, es decir, cuando cuentan con todos los recursos del sistema para ellas. En el plan 0 $Ci = 386,350$ms, con $Ti = Di = 400$ms y sin establecer L_i. Las instrucciones ejecutadas en este plan son 22971284 y los ciclos de reloj que han sucedido son 197811708, por lo que la relación ciclos por instrucción (CPI) es de $197811708/22971284 = 8,61$ ciclos por instrucción. En este plan la ejecución de la partition1 comienza en el instante 400ms.

El plan 1 corresponde al plan con un 25 % de interferencia, siendo el instante de inicio de la partition1 300ms. En este plan la interferencia no repercute en las características de las particiones, manteniendose así Ci, Ti y D_i, al igual que el ratio de CPI.

En el plan 2, el correspondiente al plan con un 50 % de interferencia, el instante de inicio de la partition1 es 200 ms. En este plan la interferencia comienza a notarse en las características de las particiones, habiendo un aumento en C_i y en el número de ciclos de reloj (el número de instrucciones ejecutadas no varía, ya que los programas no crecen en número de instrucciones al interferirse entre ellos). T_i pasa a ser 450ms, al igual que D_i. Como el número de instrucciones ejecutadas es el mismo (`mat_mul.c no varia`) y el número de ciclos de reloj aumenta, la relación de CPI aumenta también, quedando en este plan en $208043404/22971555 = 9,057$ ciclos por instrucción.

En el plan 3, el correspondiente al plan con un 75 % de interferencia, donde el inicio de la partition1 se sitúa en el instante 100ms, se acentúan los efectos de la interferencia. C_i pasa a valer 485ms, por lo que también crecen T_i y D_i hasta los 500ms. Al igual que en el plan 2, los ciclos de reloj y la relación de CPI aumentan mientras que el número de instrucciones ejecutadas se mantiene. Los CPI quedan en este caso en $247067050/22984615 = 10,75$ ciclos por instrucción.

Por último, en el plan 4, el correspondiente al plan con un 100 % de interferencia, el inicio de la partition1 se traslada hasta donde está el inicio de la partition 0 en todos los planes, a los 0ms. En este plan C_i aumenta hasta llegar a los 560ms, al igual que T_i y D_i, que aumentan hasta los 600ms. La relación de CPI tambien aumenta hasta los $286324341/23020872 = 12,44$ ciclos por instrucción.

La conclusión de esta parte del experimento es, por lo tanto, que a mayor interferencia entre los procesos, mayor es la relación de CPI

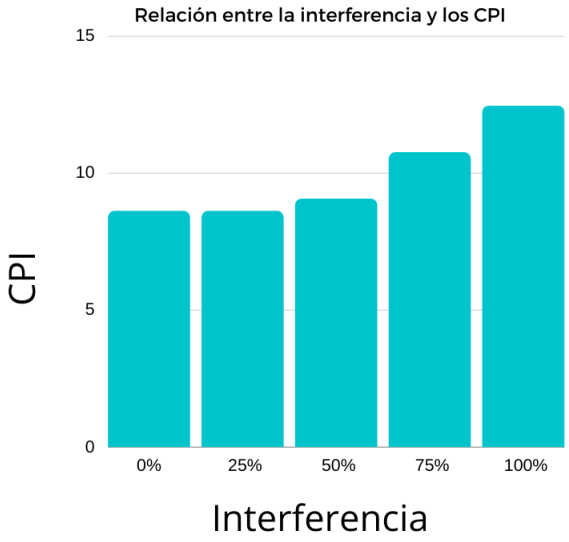

Figura 3.4: Conclusiones de la experimentación

La relación de CPI aumenta debido a que el acceso a memoria principal es compartido por todos los núcleos de procesamiento. Ello conlleva que si varios núcleos quieren acceder simultáneamente a la memoria principal, solo uno de ellos puede acceder (por ser prioritário en algún aspecto, mientras que el resto permanece a la espera de que acabe el núcleo que accedió). Esta espera es la que hace que las operaciones de acceso a memoria principal requieran más ciclos de reloj que si lo hicieran sin la existencia de interferencias.

Como ya se ha afirmado en el capítulo del estado del arte, estás interferencias generan incertidumbre en la ejecución, por lo que en sistemas críticos se deben aplicar medidas para evitar que estas incertidumbres provoquen accidentes catastróficos. En este trabajo se ofrece una solución para este problema aplicable en sistemas de criticidad mixta.

3.5 Controlador de la ejecución en sistemas de criticidad mixta

Entendiendo la criticidad como el grado con el que se debe asegurar el cumplimiento de los plazos de ejecución de las tareas, un sistema de criticidad mixta es aquel que alberga tareas con diferentes criticidades.

En este apartado se plantea una solución ante los problemas causados por la interferencia generada en la ejecución de varias tareas simultáneamente en un procesador multinúcleo, para sistemas de criticidad mixta, con el fin de asegurar la ejecución dentro de plazo de aquellas tareas que, por su criticidad, no debieran, bajo ningún concepto, ejecutarse fuera de plazo (por las consecuencias que ello pudiera acarrear).

En un sistema de criticidad mixta, donde hay tareas de criticidad alta (las que no deben ejecutarse fuera de plazo) y de criticidad baja (las que no pasa nada si se ejecutan fuera de plazo) ($l = Alta, Baja$), para poder aplicar la solución que se propone a continuación se debe cumplir la siguiente premisa: nunca se va a ejecutar más de una tarea de criticidad alta simultáneamente, o, de otra manera, en ningún instante de tiempo va a coincidir la ejecución de dos o más tareas de criticidad alta.

Si se cumple la premisa enunciada, la solución que a continuación se expone asegura la ejecución dentro de plazo de todas las tareas críticas de un sistema de criticidad mixta.

El sistema de criticidad mixta de prueba que aquí se utiliza es el mismo sistema de los experimentos sobre la interferencia entre procesos anteriormente realizados, pero con una única planificación de la ejecución con una interferencia del 100 %.

```
<ProcessorTable>
    <Processor id="0" frequency="667MHz">
        <CyclicPlanTable>
            <Plan id="0" majorFrame="1000ms">
                <Slot id="0" start="0ms" duration="1000ms" partitionId="0"/>
            </Plan>
        </CyclicPlanTable>
    </Processor>
    <Processor id="1" frequency="667MHz">
        <CyclicPlanTable>
            <Plan id="0" majorFrame="1000ms">
                <Slot id="0" start="0ms" duration="1000ms" partitionId="1"/>
            </Plan>
        </CyclicPlanTable>
    </Processor>
</ProcessorTable>
```

Planificación de la ejecución del sistema de criticidad mixta propuesto

En este sistema se va a considerar como tarea de criticidad alta la partition0, siendo, entonces, la partition1 de criticidad baja (cumpliendo así con la premisa formulada). Como en los experimentos anteriores, ambas tareas llevan a cabo una multiplicación de matrices, pero esta vez `mat_mul.c` es ligeramente diferente. En este caso, `mat_mul.c` de la partition0 implementa un algoritmo que periódicamente, leyendo los registros de la unidad de monitorización del rendimiento, va a comprobar su progreso en el tiempo, para que, en caso de evaluar que no va a llegar a ejecutarse dentro de plazo, por culpa de la interferencia generada por la ejecución de la partition1, suspenda la ejecución de esta hasta finalizar su propia ejecucion dentro de plazo.

Para poder llevar a cabo está función, la partition0 necesita tres componentes y características esenciales:

1. Una caracterización (modelo Vestal) de la tarea, ejecutandola en aislamiento para así conocer cuantas instrucciones conlleva su ejecución y cual es su plazo de ejecución.
2. Unas líneas de código que implementen el controlador de la ejecución.
3. Permisos de partición de sistema para poder suspender la ejecución de la partition1 cuando sea necesario.

Para la caracterización se ha planteado el siguiente escenario, de este modo que solo partition0 se ejecute:

```
<ProcessorTable>
    <Processor id="0" frequency="667MHz">
        <CyclicPlanTable>
            <Plan id="0" majorFrame="1000ms">
                <Slot id="0" start="0ms" duration="1000ms" partitionId="0"/>
            </Plan>
        </CyclicPlanTable>
    </Processor>
    <Processor id="1" frequency="667MHz">
        <CyclicPlanTable>
            <Plan id="0" majorFrame="1000ms">
                <Slot id="0" start="0ms" duration="0ms" partitionId="1"/>
            </Plan>
```

```
        </CyclicPlanTable>
    </Processor>
</ProcessorTable>
```

Planificación para la ejecución aislada de la partition0.

Para conocer el tiempo, el número de instrucciones y de ciclos de reloj que conlleva la ejecución de la partition0 se hace el siguiente uso de la unidad de monitorización del rendimiento y de los relojes del sistema:

```
[...]

/* Registros de eventos */
int nCiclos, nInst;

/* Marcas temporales */
xmTime_t tInicial, tFinal;

[...]

/* Registro del instante incial */
XM_get_time (XM_EXEC_CLOCK, &tInicial);

/* Configuracion de la UMR */
initPMU ();

allocCounterPMU (0, ARMV7_A9_INST_EXEC);
allocCounterPMU (1, ARMV7_PERFCTR_CLOCK_CYCLES);

for (r = 0; r < NEVENTOS; r++) {
        enablePMR(r);
}

[...]

/* Registro del instante final y calculo del tiempo transcurrido desde el
    instante inicial */
XM_get_time (XM_EXEC_CLOCK, &tFinal);

nInst   = readPMR(0);
nCiclos = readPMR(1);
PRINT("[Tiempo de ejecucion = %dms | Instrucciones ejecutadas = %d | Ciclos de
    reloj = %d] \n", (xm_u32_t)(tFinal-tInicial), nInst, nCiclos);
clearAllPMR ();
clearCCNT ();
```

Tras varias ejecuciones, con el controlador de la ejecución ya embebido en `mat_mul.c` se obtienen los siguientes resultados:

```
[P0] [Tiempo de ejecucion = 278910 ms | Instrucciones ejecutadas = 15155543 |
    Ciclos de reloj = 144749086]
[P0] [Tiempo de ejecucion = 278723 ms | Instrucciones ejecutadas = 15155122 |
    Ciclos de reloj = 144642986]
[P0] [Tiempo de ejecucion = 278724 ms | Instrucciones ejecutadas = 15155609 |
    Ciclos de reloj = 144948780]
[P0] [Tiempo de ejecucion = 278990 ms | Instrucciones ejecutadas = 15155087 |
    Ciclos de reloj = 144890134]
[P0] [Tiempo de ejecucion = 277999 ms | Instrucciones ejecutadas = 15155555 |
    Ciclos de reloj = 144749122]
[P0] [Tiempo de ejecucion = 279100 ms | Instrucciones ejecutadas = 15155419 |
    Ciclos de reloj = 144690019]
```

Por lo tanto, se puede afirmar que $Ci = 278\text{ms}$ y que la relación de CPI adecuada para la ejecución dentro de plazo es $144749086/15155543 = 9,55$ ciclos por instruc-

ción. Para dejar cierto margen de maniobra, se establece que $Di = Ti = 300$ms. La frecuencia efectiva medida a través de la unidad de monitorización del rendimiento es de $144642986/278724 = 519$ MHz, por lo que, si la ejecución se alargará fruto de la interferencia, el número máximo de ciclos que se debe alcanzar es $Frecuencia * TiempoEnMicroSegundos = 519 * 300000 = 155684103$ ciclos, que suponen una relación de CPI de $155684103/15155543 = 10,27$ ciclos por instrucción. Está relación de CPI es el límite antes del cual se sabe que la ejecución se va a llevar a cabo dentro de plazo. Si el CPI medido por el controlador está por encima del valor medio entre ese valor máximo y el valor calculado en esta ejecución ($(10,27 + 9,55)/2 = 9,91$ ciclos por instrucción), se debe suspender la ejecución de la partition1.

En este caso C_i es menor que en la experimentación anterior ya que no se recopilan los diferentes resultados en un vector, sino que separadamente se han ejecutado los diferentes escenarios.

Las líneas de código que implementan el controlador de la ejecución son las siguientes:

```
1  [...]
2  char suspended = 0;
3
4  void ManejadorDelTemporizador (trapCtxt_t *ctxt)
5  {
6      int instActuales   = readPMR (0);
7      int ciclosActuales = readPMR (1);
8      float cpi          = (float)ciclosActuales/instActuales;
9      PRINT("Ciclos = %d Instrucciones = %d\n", ciclosActuales, instActuales);
10
11     /* Si el CPI actual esta por encima de la media entre el maximo */
12     /* CPI admisible y el CPI minimo obtenido, la particion1 se suspende */
13     if (cpi > (CPI + maxCPI) / 2 && suspended == 0){
14             XM_suspend_partition ((xm_u32_t) 1);
15             PRINT("Suspendida la particion 1\n");
16             suspended = 1;
17     }
18 }
19
20 void PartitionMain (void) {
21         InstallIrqHandler (XAL_XMEXT_TRAP(XM_VT_EXT_HW_TIMER),
               ManejadorDelTemporizador);    /* Se instala el manejador del
               temporizador */
22         HwSti ();

               /* Se habilitan las interrupciones          */
23         XM_clear_irqmask (0, (1<<XM_VT_EXT_HW_TIMER));
                                                /* Se desenmascaran las
               interrupciones      */
24                                                                          /*
25         [...]
26
27         /* Configuracion del temporizador para que de una interrupcion cada
               decima parte del WCET*/
28         XM_get_time (XM_HW_CLOCK, &relojHw);
29         relojHw += (xmTime_t) 1000;
30         XM_set_timer (XM_HW_CLOCK, relojHw, (xmTime_t)C/10);
```

```
31
32          [...]
33
34          /* Se comprueba si la ejecucion se ha llevado a cabo dentro del plazo
                deseado */
35          if((xm_u32_t)(tFinal-tInicial) <= (xm_u32_t) D)
36                  PRINT("Ejecutado completamente dentro de plazo %d <= %d\n", (
                        xm_u32_t)(tFinal-tInicial), (xm_u32_t) D);
37          else
38                  PRINT("Ejecutado completamente fuera de plazo\n");
39
40          /* Se reanuda la ejecucion de la particion1 si esta fue parada con
                anterioridad */
41          if (suspended == 1){
42                  XM_resume_partition ((xm_u32_t) 1);
43                  suspended = 0;
44          }
```

Controlador de la ejecución al completo.

El algoritmo de control es muy sencillo. Tras caracterizar la tarea de criticidad alta (en este caso `mat_mul.c`) se conocen los parámetros C_i, T_i y D_i, además del número de ciclos de reloj, el número de instrucciones y el tiempo que conlleva su ejecución (deduciendo también la frecuencia del reloj). Primero se programa un temporizador para que notifique mediante una interrupción *software* cada vez que transcurren $Ci/10$ milisegundos. Esta interrupción es atendida por una función que, a través de la unidad de monitorización del rendimiento y los parámetros que caracterizan a la tarea evalúa el progreso de la misma. Con el número de instrucciones y el de ciclos se calcula el CPI ($nCiclos/nInstrucciones$) con el que la tarea se ejecuta dentro de plazo en aislamiento (en el mejor de los casos posibles). También con el número de instrucciones, pero ahora con D_i y la frecuencia de reloj (que se obtiene con C_i y con el número de ciclos), se calcúla el CPI máximo ($((nCiclos/Ci) * Di)/nInstrucciones$) que puede alcanzar la ejecución, por encima del cual la ejecución finalizaría una vez pasado el plazo (D_i). Con el CPI y el CPI máximo se calcula un CPI intermedio ($(CPI + CPImaximo)/2$) que es el que utiliza el algoritmo de control de la ejecución para saber cuando suspender la ejecución de la partición 1 (si el CPI que lleva la tarea en un momento dado es mayor que la media de los CPI anteriormente mencionada, entonces, se suspende la ejecución de la partición 1). Después simplemente se espera a que la partición de criticidad alta finalice su ejecución (dentro de plazo) para reanudar la ejecución de la partición 0.

```
[...]

if (cpi > (CPI + maxCPI) / 2 && suspended == 0){
        XM_suspend_partition ((xm_u32_t) 1);

[...]

if (suspended == 1){
        XM_resume_partition ((xm_u32_t) 1);

[...]
```

Suspensión y reanudación de la partición 1

Al final del bucle principal de `mat_mul.c` se comprueba y se muestra por pantalla si la ejecución de la partición ha sido dentro del plazo establecido D_i:

```
[...]

if((xm_u32_t)(tFinal-tInicial) <= (xm_u32_t) D)
```

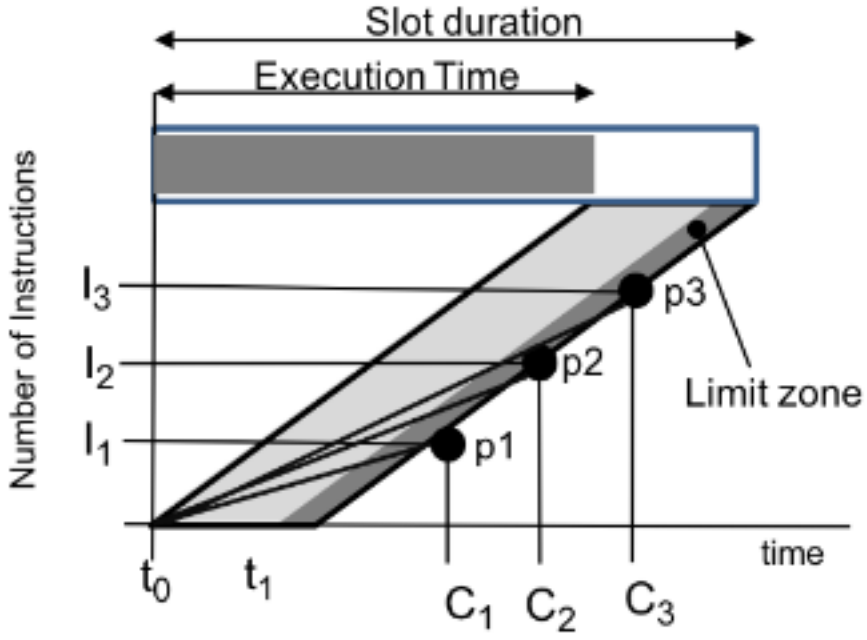

Figura 3.5: Gráfica que ilustra el CPI que se busca para lograr una ejecución dentro de plazo. Es cuando el CPI entra en la zona oscura que se suspende la ejecución de la partición 1. *Slot duration* = periodo de ejecución. *Execution time* = C_i

```
4           PRINT("Ejecutado completamente dentro de plazo %d <= %d\n", (xm_u32_t)(
                tFinal-tInicial), (xm_u32_t) D);
5   else
6           PRINT("Ejecutado completamente fuera de plazo\n");
7
8   [...]
```

Comprobación del cumplimiento del plazo

Para probar el funcionamiento del controlador primero se realizan varias ejecuciones sin este funcionando (con las líneas de suspensión y reanudación de la particion 1 comentadas), para ver como se comportan las particiones dados ciertos instantes de inicio y periodos de ejecución. A continuación se muestran los diferentes escenarios puestos a prueba y los resultados obtenidos en su ejecución:

```
1
2   [...]
3   <Plan id="0" majorFrame="500ms">
4       <Slot id="0" start="0ms" duration="300ms" partitionId="0"/>
5   </Plan>
6   [...]
7   <Plan id="0" majorFrame="500ms">
8       <Slot id="0" start="300ms" duration="200ms" partitionId="1"/>
9   </Plan>
10  [...]
```

(Escenario 0) Planificación de la ejecución en aislamiento

```
1   [P0] [Tiempo de ejecucion = 269886 ms | Instrucciones ejecutadas = 14932433 |
        Ciclos de reloj = 138183935]
2   [P0] Ejecutado completamente dentro de plazo 269886 <= 300000
3   [P0] [Tiempo de ejecucion = 266187 ms | Instrucciones ejecutadas = 15159130 |
        Ciclos de reloj = 141613343]
```

```
[P0] Ejecutado completamente dentro de plazo 266187 <= 300000
[P1] [Tiempo de ejecucion = 531531 ms]
[P0] [Tiempo de ejecucion = 266196 ms | Instrucciones ejecutadas = 14965004 |
    Ciclos de reloj = 138231937]
[P0] Ejecutado completamente dentro de plazo 266196 <= 300000
[P1] [Tiempo de ejecucion = 531522 ms]
[P0] [Tiempo de ejecucion = 266187 ms | Instrucciones ejecutadas = 14965352 |
    Ciclos de reloj = 138227107]
[P0] Ejecutado completamente dentro de plazo 266187 <= 300000
[P0] [Tiempo de ejecucion = 266205 ms | Instrucciones ejecutadas = 14965263 |
    Ciclos de reloj = 138233987]
[P0] Ejecutado completamente dentro de plazo 266205 <= 300000
[P1] [Tiempo de ejecucion = 531526 ms]
[P1] [Tiempo de ejecucion = 531515 ms]
[P1] [Tiempo de ejecucion = 531521 ms]
```

Resultados de la ejecución sin interferencias

En este caso, sin interferencias entre las particiones, la partición 0 se ejecuta, de media, en 268ms, es decir, dentro de plazo. La partición 1 se ejecuta en 531ms de media porque al tener un $Ti = 200$ms no le da tiempo a acabar en un solo periodo (el coste de ejecución de la partición 1 es prácticamente el mismo que el de la partición 0, son del mismo orden de magnitud $N^3 * NMults$). Si en `mat\_mul2.c`, que es el archivo que se ejecuta en la partición 1 (es igual que `mat_mul.c` pero sin el controlador de la ejecución), se utilizase `XM_EXEC_CLOCK` en lugar de `XM_HW_CLOCK` se podría ver como el tiempo de ejecución, sumando solamente la ejecución dentro de cada periodo (sin tener en cuenta la espera entre periodo y periodo, como sucede con `XM_HW_CLOCK`) sería prácticamente el mismo que el de la partición 0, pero sumandole los costes temporales de los cambios de contexto.

```
[...]
<Plan id="0" majorFrame="500ms">
    <Slot id="0" start="0ms" duration="300ms" partitionId="0"/>
</Plan>
[...]
<Plan id="0" majorFrame="500ms">
    <Slot id="0" start="200ms" duration="300ms" partitionId="1"/>
</Plan>
[...]
```

(Escenario 1) Planificación de la ejecución con 100ms (25 %) de interferéncia

% de interferencia

```

[P0] Suspendida la particion 1
[P1] [Tiempo de ejecucion = 257899 ms]
[P0] [Tiempo de ejecucion = 516700 ms | Instrucciones ejecutadas = 15501226 |
    Ciclos de reloj = 162181906]
[P0] Reanudada la ejecucion de 1
[P1] [Tiempo de ejecucion = 231411 ms]
[P0] Suspendida la particion 1
[P1] [Tiempo de ejecucion = 258033 ms]
[P0] [Tiempo de ejecucion = 508735 ms | Instrucciones ejecutadas = 15407591 |
    Ciclos de reloj = 160047559]
[P0] Ejecutado completamente fuera de plazo
[P0] Reanudada la ejecucion de 1
[P0] Reanudada la ejecucion de 1
[P1] [Tiempo de ejecucion = 231409 ms]
[P0] Suspendida la particion 1
[P1] [Tiempo de ejecucion = 258082 ms]
[P0] [Tiempo de ejecucion = 508769 ms | Instrucciones ejecutadas = 15404168 |
    Ciclos de reloj = 160064080]
[P0] Ejecutado completamente fuera de plazo
```

```
[P0] Ejecutado completamente fuera de plazo
[P0] Reanudada la ejecucion de 1
[P0] [Tiempo de ejecucion = 281294 ms | Instrucciones ejecutadas = 15334632 | Ciclos de reloj = 145964181]
[P0] Ejecutado completamente dentro de plazo 281294 <= 300000
[P0] [Tiempo de ejecucion = 281302 ms | Instrucciones ejecutadas = 15334418 | Ciclos de reloj = 145971882]
[P0] Ejecutado completamente dentro de plazo 281302 <= 300000
[P1] [Tiempo de ejecucion = 260754 ms]
[P0] [Tiempo de ejecucion = 514738 ms | Instrucciones ejecutadas = 15424212 | Ciclos de reloj = 161177500]
[P0] Ejecutado completamente fuera de plazo
[P1] [Tiempo de ejecucion = 231411 ms]
[P1] [Tiempo de ejecucion = 260941 ms]
[P0] [Tiempo de ejecucion = 506828 ms | Instrucciones ejecutadas = 15327131 | Ciclos de reloj = 159067524]
[P0] Ejecutado completamente fuera de plazo
[P1] [Tiempo de ejecucion = 231409 ms]
[P1] [Tiempo de ejecucion = 260921 ms]
[P0] [Tiempo de ejecucion = 506820 ms | Instrucciones ejecutadas = 15327160 | Ciclos de reloj = 159061444]
[P0] Ejecutado completamente fuera de plazo
[P0] [Tiempo de ejecucion = 266195 ms | Instrucciones ejecutadas = 14965438 | Ciclos de reloj = 138229179]
[P0] Ejecutado completamente dentro de plazo 266195 <= 300000
[P0] [Tiempo de ejecucion = 266195 ms | Instrucciones ejecutadas = 14963946 | Ciclos de reloj = 138231968]
[P0] Ejecutado completamente dentro de plazo 266195 <= 300000
```

Con un 25 % de interferencia en la ejecución la partición 0 ya se ejecuta sobrepasando D_i ($Ci = 510$ms), como se puede ver en la segunda y tercera línea de los resultados de la ejecución con un 25 % de interferencia. Ello quiere decir que, en los experimentos mostrados más adelante donde la interferencia es del 25 % también, con el controlador de la ejecución ya puesto en marcha, la correcta ejecución (dentro de plazo) de la partición 0 es gracias al controlador de la ejecución implementado. En este caso crece 200ms el tiempo de ejecución debido a que se utiliza `XM_HW_CLOCK`, como en el caso anterior, que tiene en cuenta la espera entre periodo y periodo (`XM_EXEC_CLOCK` solo mide el tiempo de ejecución de una partición transcurrido dentro del periodo, mientras que `XM_HW_CLOCK` mide el tiempo absoluto transcurrido).

```
[...]
<Plan id="0" majorFrame="500ms">
    <Slot id="0" start="0ms" duration="400ms" partitionId="0"/>
</Plan>
[...]
<Plan id="0" majorFrame="500ms">
    <Slot id="0" start="100ms" duration="400ms" partitionId="1"/>
</Plan>
[...]
```

(Escenario 2) Ejecución con 300ms (75 %) de interferencia

```

[P0] [Tiempo de ejecucion = 359373 ms | Instrucciones ejecutadas = 15364109 | Ciclos de reloj = 184001874]
[P0] Ejecutado completamente fuera de plazo
[P1] [Tiempo de ejecucion = 306183 ms]
[P0] [Tiempo de ejecucion = 360608 ms | Instrucciones ejecutadas = 15640799 | Ciclos de reloj = 189959820]
[P0] Ejecutado completamente fuera de plazo
[P1] [Tiempo de ejecucion = 309652 ms]
[P0] [Tiempo de ejecucion = 355480 ms | Instrucciones ejecutadas = 15395424 | Ciclos de reloj = 183947595]
```

```
[P0] Ejecutado completamente fuera de plazo
[P1] [Tiempo de ejecucion = 306122 ms]
[P0] [Tiempo de ejecucion = 365443 ms | Instrucciones ejecutadas = 15680220 |
    Ciclos de reloj = 189048248]
[P0] Ejecutado completamente fuera de plazo
[P1] [Tiempo de ejecucion = 306130 ms]
[P0] [Tiempo de ejecucion = 365417 ms | Instrucciones ejecutadas = 15679501 |
    Ciclos de reloj = 189034169]
[P0] Ejecutado completamente fuera de plazo
[P1] [Tiempo de ejecucion = 306154 ms]
```

Resultados de la ejecución con un 75 % de interferencias

En este caso, de nuevo, la partición 0 excede D_i en su ejecución. El tiempo de ejecución es menor que en el caso anterior porque el tiempo de ejecución dentro del periodo es mayor, por lo que las marcas temporales no recogen, en este caso, el tiempo de espera transcurrido entre periodo y periodo (en el caso anterior el tiempo de ejecución dentro de un periodo era de 300ms y el periodo era de 500ms, por lo que una ejecución de 510ms implicaba 200ms de espera entre tiempo de ejecución y tiempo de ejecución, siendo entonces 310ms el tiempo de uso del procesador, tiempo menor que los 360ms de uso del procesador medidos en esta última ejecución).

```
[...]
<Plan id="0" majorFrame="500ms">
    <Slot id="0" start="0ms" duration="500ms" partitionId="0"/>
</Plan>
[...]
<Plan id="0" majorFrame="500ms">
    <Slot id="0" start="0ms" duration="500ms" partitionId="1"/>
</Plan>
[...]
```

(Escenario 3) Planificación de la ejecución con 500ms (100 %) de interferencia

```
[P1] [Tiempo de ejecucion = 325105 ms]
[P0] [Tiempo de ejecucion = 382267 ms | Instrucciones ejecutadas = 15484228 |
    Ciclos de reloj = 195723479]
[P0] Ciclos = 200521100 Instrucciones = 15762181
[P0] Ejecutado completamente fuera de plazo
[P1] [Tiempo de ejecucion = 323079 ms]
[P0] [Tiempo de ejecucion = 373779 ms | Instrucciones ejecutadas = 15625890 |
    Ciclos de reloj = 196720911]
[P0] [Tiempo de ejecucion = 373779 ms | Instrucciones ejecutadas = 15625890 |
    Ciclos de reloj = 196720911]
[P0] Ejecutado completamente fuera de plazo
[P1] [Tiempo de ejecucion = 325096 ms]
[P0] [Tiempo de ejecucion = 378576 ms | Instrucciones ejecutadas = 15513652 |
    Ciclos de reloj = 195782663]
[P0] Ejecutado completamente fuera de plazo
[P1] [Tiempo de ejecucion = 325052 ms]
[P0] [Tiempo de ejecucion = 378475 ms | Instrucciones ejecutadas = 15514739 |
    Ciclos de reloj = 195733907]
[P0] Ejecutado completamente fuera de plazo
[P1] [Tiempo de ejecucion = 325185 ms]
[P0] [Tiempo de ejecucion = 378551 ms | Instrucciones ejecutadas = 15511721 |
    Ciclos de reloj = 195772291]
[P0] Ejecutado completamente fuera de plazo
```

Resultados de la ejecución del plan con una interferencia del 100 %

De nuevo, sin novedad alguna, la particion 0 se ejecuta fuera de plazo (a mayor interferencia mayor tiempo de ejecución).

A continuación se muestran los resultados de la ejecución de cuatro escenarios posibles, con el controlador de la ejecución funcionando en partition0 (sin línea de código alguna comentada). Pudiendo observar así el cumplimiento de la ejecución dentro de plazo a pesar de los diferentes grados de interferéncia, que son del 0 %, del 33 %, del 66 % y del 100 %. A diferencia de los experimentos anteriores, en los que no se hace uso del controlador de la ejecución, en estos experimentos se observa un descenso general del C_i, debido en gran parte a que no se recopilan los resultados en un vector, lo que ahorra muchos fallos de acceso a memoria.

```

[...]
<Plan id="0" majorFrame="500ms">
    <Slot id="0" start="0ms" duration="300ms" partitionId="0"/>
</Plan>
[...]
<Plan id="0" majorFrame="500ms">
    <Slot id="0" start="300ms" duration="200ms" partitionId="1"/>
</Plan>
[...]
```

(Escenario 0) Planificación de la ejecución en aislamiento

```
[P0] Ciclos = 516424 Instrucciones = 66909
[P0] Ciclos = 14704596 Instrucciones = 1758480
[P0] Ciclos = 28895538 Instrucciones = 3436221
[P0] Ciclos = 43086449 Instrucciones = 5115774
[P0] Ciclos = 57277894 Instrucciones = 6794370
[P0] Ciclos = 71468715 Instrucciones = 8473723
[P0] Ciclos = 85659685 Instrucciones = 10151888
[P0] Ciclos = 99851082 Instrucciones = 11826950
[P0] Ciclos = 114041987 Instrucciones = 1350355
[P0] [Tiempo de ejecucion = 249132 ms | Instrucciones ejecutadas = 14885721 |
    Ciclos de reloj = 127557688]
[P0] [Tiempo de ejecucion = 249132 ms | Instrucciones ejecutadas = 14885721 |
    Ciclos de reloj = 127557688]
[P0] Ciclos = 132342334 Instrucciones = 15163365
[P0] Ejecutado completamente dentro de plazo 249132 <= 300000
[P0] Ciclos = 129581669 Instrucciones = 15036301
[P0] [Tiempo de ejecucion = 255246 ms | Instrucciones ejecutadas = 15326448 |
    Ciclos de reloj = 134619479]
[P0] Ejecutado completamente dentro de plazo 255246 <= 300000
[P1] [Tiempo de ejecucion = 531534 ms]
[P0] [Tiempo de ejecucion = 255245 ms | Instrucciones ejecutadas = 15209448 |
    Ciclos de reloj = 132623154]
[P0] Ejecutado completamente dentro de plazo 255245 <= 300000
[P0] [Tiempo de ejecucion = 255186 ms | Instrucciones ejecutadas = 15205010 |
    Ciclos de reloj = 132595405]
[P0] Ejecutado completamente dentro de plazo 255186 <= 300000
[P1] [Tiempo de ejecucion = 531529 ms]
[P0] [Tiempo de ejecucion = 255161 ms | Instrucciones ejecutadas = 15206525 |
    Ciclos de reloj = 132582760]
[P0] Ejecutado completamente dentro de plazo 255161 <= 300000
[P1] [Tiempo de ejecucion = 531534 ms]
[P1] [Tiempo de ejecucion = 531533 ms]
[P1] [Tiempo de ejecucion = 531528 ms]
```

Resultados de la ejecución del escenario 0

En este escenario la interferencia es del 0 %. La partición 0 logra ejecutarse dentro de plazo, con $C_i = 249132$ ms, sin dificultad alguna, mientras que la partición 1 necesita dos ranuras de tiempo (*major frames* para ejecutarse, alcanzando un C_i de 531534 ms. Los mensajes que se muestran no son todos los que se han producido en la ejecución, sino que

son una selección de los mismos, intentando ser lo menos redundante posible. En primer lugar se muestra el número de ciclos y de instrucciones que lleva contadas la unidad de monitorización del rendimiento en cada parada de control de la ejecución. Después esta el mensaje que muestra el tiempo total consumido en la completa ejecución de cada partición, y por último un mensaje que indica si el tiempo de ejecución es menor que el plazo límite de ejecución.

```
[...]
<Plan id="0" majorFrame="500ms">
    <Slot id="0" start="0ms" duration="300ms" partitionId="0"/>
</Plan>
[...]
<Plan id="0" majorFrame="500ms">
    <Slot id="0" start="200ms" duration="300ms" partitionId="1"/>
</Plan>
[...]
```

(Escenario 1) Ejecución con 100ms (33 %) de interferencia

```
[P0] Ciclos = 516065 Instrucciones = 66823
[P0] Ciclos = 14704536 Instrucciones = 1758009
[P0] Ciclos = 28895308 Instrucciones = 3435337
[P0] Ciclos = 43086805 Instrucciones = 5114540
[P0] Ciclos = 57277602 Instrucciones = 6791989
[P0] Ciclos = 71468535 Instrucciones = 8471432
[P0] Ciclos = 85659789 Instrucciones = 10148976
[P0] Ciclos = 99850823 Instrucciones = 11823729
[P0] Ciclos = 114043307 Instrucciones = 12979439
[P0] Ciclos = 128234917 Instrucciones = 14116168
[P0] SUMA total de los valores de la matriz resultante = 7926100
[P0] [Tiempo de ejecucion = 274283 ms | Instrucciones ejecutadas = 15003327 |
    Ciclos de reloj = 140435815]
[P0] [Tiempo de ejecucion = 274283 ms | Instrucciones ejecutadas = 15003327 |
    Ciclos de reloj = 140435815]
[P0] Ciclos = 145247584 Instrucciones = 15273277
[P0] Ejecutado completamente dentro de plazo 274283 <= 300000
[P1] SUMA total de los valores de la matriz resultante = 7926100
[P1] [Tiempo de ejecucion = 253471 ms]
[P0] Ejecutado completamente dentro de plazo 274283 <= 300000
[P0] Ciclos = 154994387 Instrucciones = 15845387
[P0] Ciclos = 159910886 Instrucciones = 16130894
[P1] SUMA total de los valores de la matriz resultante = 7926100
[P1] [Tiempo de ejecucion = 231411 ms]
[P0] [Tiempo de ejecucion = 270316 ms | Instrucciones ejecutadas = 15036182 |
    Ciclos de reloj = 140340069]
[P0] [Tiempo de ejecucion = 270316 ms | Instrucciones ejecutadas = 15036182 |
    Ciclos de reloj = 140340069]
[P0] Ejecutado completamente dentro de plazo 270316 <= 300000
[P1] [Tiempo de ejecucion = 253345 ms]
[P0] Ejecutado completamente dentro de plazo 270316 <= 300000
[P1] [Tiempo de ejecucion = 231408 ms]
[P0] [Tiempo de ejecucion = 270196 ms | Instrucciones ejecutadas = 15034006 |
    Ciclos de reloj = 140281979]
[P0] [Tiempo de ejecucion = 270196 ms | Instrucciones ejecutadas = 15034006 |
    Ciclos de reloj = 140281979]
[P0] Ejecutado completamente dentro de plazo 270196 <= 300000
[P1] [Tiempo de ejecucion = 253325 ms]
[P0] Ejecutado completamente dentro de plazo 270196 <= 300000
[P0] [Tiempo de ejecucion = 255169 ms | Instrucciones ejecutadas = 15204952 |
    Ciclos de reloj = 132584145]
[P0] Ejecutado completamente dentro de plazo 255169 <= 300000
[P0] Ejecutado completamente dentro de plazo 255169 <= 300000
[P0] [Tiempo de ejecucion = 249598 ms | Instrucciones ejecutadas = 15042052 |
    Ciclos de reloj = 129730521]
```

```
[P0] Ejecutado completamente dentro de plazo 249598 <= 300000
```

Resultados de la ejecución del escenario 1

En este escenario la interferencia es del 33 % (100 ms de interferencia). La partición 0 logra ejecutarse dentro de plazo, con $C_i = 274283$ ms, ahora con algunas dificultades (C_i ha crecido con respecto al escenario 0) causadas por la interferencia en la ejecución, pero no las suficientes como para detener la ejecución de la partición 1, que alcanza un C_i de 253325 ms, lo que significa que se ha ejecutado en una única ranura temporal, siendo, por lo tanto, C_i menor en este escenario que en el escenario 0.

```
[...]
<Plan id="0" majorFrame="500ms">
    <Slot id="0" start="0ms" duration="300ms" partitionId="0"/>
</Plan>
[...]
<Plan id="0" majorFrame="500ms">
    <Slot id="0" start="100ms" duration="300ms" partitionId="1"/>
</Plan>
[...]
```

(Escenario 2) Planificación de la ejecución con 200ms (66 %) de interferencia

```
[P0] Ciclos = 516683 Instrucciones = 66842
[P0] Ciclos = 14704555 Instrucciones = 1758411
[P0] Ciclos = 28895218 Instrucciones = 3436247
[P0] Ciclos = 43086574 Instrucciones = 5115625
[P0] Ciclos = 57278747 Instrucciones = 6519493
[P0] Ciclos = 71470704 Instrucciones = 7659874
[P0] Ciclos = 85660456 Instrucciones = 8799994
[P0] Suspendida la particion 1
[P0] Ciclos = 99851335 Instrucciones = 10374193
[P0] Ciclos = 114042211 Instrucciones = 12048811
[P0] Ciclos = 128233075 Instrucciones = 13721518
[P0] SUMA total de los valores de la matriz resultante = 7926100
[P0] [Tiempo de ejecucion = 276530 ms | Instrucciones ejecutadas = 15082545 |
    Ciclos de reloj = 141585448]
[P0] [Tiempo de ejecucion = 276530 ms | Instrucciones ejecutadas = 15082545 |
    Ciclos de reloj = 141585448]
[P0] Ciclos = 146370219 Instrucciones = 15360219
[P0] Ejecutado completamente dentro de plazo 276530 <= 30[P0] Ciclos =
    153428824 Instrucciones = 15796594
[P0] Ciclos = 158085660 Instrucciones = 16068564
[P0] Reanudada la ejecucion de 1
[P1] [Tiempo de ejecucion = 681157 ms]
[P0] Suspendida la particion 1
[P0] [Tiempo de ejecucion = 272128 ms | Instrucciones ejecutadas = 15113769 |
    Ciclos de reloj = 141267459]
[P0] [Tiempo de ejecucion = 272128 ms | Instrucciones ejecutadas = 15113769 |
    Ciclos de reloj = 141267459]
[P0] Ejecutado completamente dentro de plazo 272128 <= 300000
[P0] Ejecutado completamente dentro de plazo 272128 <= 300000
[P0] Reanudada la ejecucion de 1
[P1] [Tiempo de ejecucion = 682693 ms]
[P0] Suspendida la particion 1
[P0] [Tiempo de ejecucion = 281218 ms | Instrucciones ejecutadas = 15405439 |
    Ciclos de reloj = 145920051]
[P0] Ejecutado completamente dentro de plazo 281218 <= 300000
[P0] Reanudada la ejecucion de 1
[P0] Reanudada la ejecucion de 1
[P1] [Tiempo de ejecucion = 684065 ms]
[P0] Suspendida la particion 1
[P0] [Tiempo de ejecucion = 282886 ms | Instrucciones ejecutadas = 15236179 |
    Ciclos de reloj = 146776309]
```

```
[P0] Ejecutado completamente dentro de plazo 282886 <= 300000
[P0] Ejecutado completamente dentro de plazo 282886 <= 300000
[P0] Reanudada la ejecucion de 1
[P1] [Tiempo de ejecucion = 666363 ms]
[P0] Suspendida la particion 1
[P0] [Tiempo de ejecucion = 282147 ms | Instrucciones ejecutadas = 15238896 |
    Ciclos de reloj = 146397215]
[P0] Reanudada la ejecucion de 1
[P1] [Tiempo de ejecucion = 667767 ms]
```

Resultados de la ejecución del escenario 2

En este escenario la interferencia es del 66 % (200ms de interferencia). La partición 0 logra ejecutarse dentro de plazo, con C_i = 276530ms, ahora con algunas dificultades (C_i ha crecido ligeramente con respecto al escenario 1) causadas por la interferencia en la ejecución, siendo en este escenario dificultades suficientes como para detener la ejecución de la partición 1, que alcanza un C_i de 684065ms, lo que significa que se ha ejecutado en tres ranuras temporales. La detención de la ejecución de la partición 1 se conoce gracias al mensaje impreso en pantalla que dice lo siguiente: [P0] Suspendida la particion 1. Este mensaje se muestra cuando el controlador de la ejecución detiene la partición 1.

```
[...]
<Plan id="0" majorFrame="500ms">
    <Slot id="0" start="0ms" duration="300ms" partitionId="0"/>
</Plan>
[...]
<Plan id="0" majorFrame="500ms">
    <Slot id="0" start="0ms" duration="300ms" partitionId="1"/>
</Plan>
[...]
```

(Escenario 3) Planificación de la ejecución con 300ms (100 %) de interferencia

```
[P0] Ciclos = 516784 Instrucciones = 44385
[P0] Suspendida la particion 1
[P0] Ciclos = 14703427 Instrucciones = 1632016
[P0] Ciclos = 28894102 Instrucciones = 3309805
[P0] Ciclos = 43085153 Instrucciones = 4989246
[P0] Ciclos = 57276351 Instrucciones = 6667578
[P0] Ciclos = 71467336 Instrucciones = 8346925
[P0] Ciclos = 85658679 Instrucciones = 10025163
[P0] Ciclos = 99849679 Instrucciones = 11699992
[P0] Ciclos = 114040921 Instrucciones = 13376754
[P0] SUMA total de los valores de la matriz resultante = 7926100
[P0] Ciclos = 129130894 Instrucciones = 14962233
[P0] [Tiempo de ejecucion = 262043 ms | Instrucciones ejecutadas = 15252298 |
    Ciclos de reloj = 134168817]
[P0] Ejecutado completamente dentro de plazo 262043 <= 300000
[P0] Reanudada la ejecucion de 1
[P0] Reanudada la ejecucion de 1
[P0] Ciclos = 143162092 Instrucciones = 15771988
[P0] Suspendida la particion 1
[P0] [Tiempo de ejecucion = 259387 ms | Instrucciones ejecutadas = 15228299 |
    Ciclos de reloj = 136756051]
[P0] Ejecutado completamente dentro de plazo 259387 <= 300000
[P0] Ejecutado completamente dentro de plazo 259387 <= 300000
[P0] Reanudada la ejecucion de 1
[P0] Suspendida la particion 1
[P0] [Tiempo de ejecucion = 260328 ms | Instrucciones ejecutadas = 15116436 |
    Ciclos de reloj = 135236341]
[P0] Ejecutado completamente dentro de plazo 260328 <= 300000
[P0] Ejecutado completamente dentro de plazo 260328 <= 300000
[P0] Reanudada la ejecucion de 1
```

```
[P0] Suspendida la particion 1
[P0] [Tiempo de ejecucion = 259996 ms | Instrucciones ejecutadas = 15115823 |
    Ciclos de reloj = 135066214]
[P0] Ejecutado completamente dentro de plazo 259996 <= 300000
[P0] Ejecutado completamente dentro de plazo 259996 <= 300000
[P0] Reanudada la ejecucion de 1
[P0] Suspendida la particion 1
[P0] [Tiempo de ejecucion = 259598 ms | Instrucciones ejecutadas = 15115385 |
    Ciclos de reloj = 134865915]
[P0] Ejecutado completamente dentro de plazo 259598 <= 300000
[P0] Ejecutado completamente dentro de plazo 259598 <= 300000
[P0] Reanudada la ejecucion de 1
[P1] [Tiempo de ejecucion = 2636581 ms]
[P1] [Tiempo de ejecucion = 231408 ms]
[P1] [Tiempo de ejecucion = 231406 ms]
[P1] [Tiempo de ejecucion = 231404 ms]
[P1] [Tiempo de ejecucion = 231411 ms]
```

Resultados de la ejecución del escenario 3

En este escenario la interferencia es del 100 % (300ms de interferencia). La partición 0 logra ejecutarse dentro de plazo, exponiendo el éxito del controlador de la ejecución, alcanzando $C_i = 262043$ms, con algunas dificultades en este experimento también (llegando a detener la ejecución de la partición 1, que alcanza un C_i máximo de 2636581ms). El primer C_i que manifiesta la partición 1 indica que, debido a la precocidad con la que la partición 0 debe detener la ejecución de la partición 1, hasta que no se han realizado todas las ejecuciones completas de la particioń 0 no se ha realizado aún la primera ejecución completa de la partición 1. Después de esta ejecucón, el C_i de la partición 1 baja hasta alcanzar valores menores que los alcanzados por la partición 0: $C_i = 231408$ms.

CAPÍTULO 4
Conclusiones

Como se puede observar en la sección anterior, los abjetivos planteados al comienzo del trabajo han sido alcanzados al completo, además de haber sido alcanzados con éxito.

En primer lugar, se han modelado las cargas de trabajo, determinando los valores de C_i, D_i, T_i y L_i. Esto ha permitido medir los resultados obtenidos y compararlos entre escenario y escenario.

En segundo lugar, se han planteado varios experimentos en los que se ha sometido la ejecución de las particiones a diversos grados de interferencia, pudiendo observar la influencia de esta interferencia en los valores del modelo utilizado.

Y, por último, se ha implementado un controlado de la ejecución que ha logrado la ejecución dentro de plazo de la partición 0 en todos los experimentos planteados.

4.0.1. Modelaje

El primer objetivo planteado en este trabajo es el modelaje de las cargas de trabajo a alas que se iba someter el sistema con el que se ha experimentado. Para ello el modelo utilizado ha sido el modelo de Vestal.

Este objetivo se ha cumplido satisfactoriamente gracias a la unidad de monitorización del rendimiento, que ha permitido cuantificar los parámetros del modelo seguido, y a la configuración de Xtratum, que a través de su planificador cíclico ha permitido ubicar en el tiempo la ejecución de los experimentos de una forma sencilla y fiable.

4.0.2. Escenarios de ejecución

El segundo objetivo del trabajo es plantear escenarios de ejecución con los que provocar las interferencias problemáticas para los sistemas de criticidad mixta con procesadores multinúcleo.

Este objetivo también se ha cumplido satisfactoriamente gracias al planificador cíclico de Xtratum y su configuración del *hardware*.

4.0.3. Controlador de la ejecución

El tercer y último objetivo del trabajo es diseñar un controlador de la ejecución que asegure la correcta ejecución de los componentes críticos del sistema, a pesar de las interferéncias que entorpecen el determinismo del sistema.

Objetivo que tambíen se ha cumplido satisfactoriamente, como se puede observar en la sección tercera de este trabajo.

4.0.4. Otras consideraciones y posibles mejoras

A lo largo del trabajo se demuestra que, las problemáticas teóricas agregadas al utilizar un procesador multinúcleo son una realidad, y que el hecho de compartir recursos principales afecta considerablemente en los parámetros observados según el modelo Vestal, provocando el aumento de los tiempos de ejecución, por ejemplo.

También se demuestra en este trabajo que, si se plantean unas premisas adecuadas, se caracterizan bien los elementos de un sistema de criticidad mixta y se sigue una estrategia de control, se puede alcanzar un determinismo clave para la seguridad del sistema y la confianza en el mismo, ya que toda ejecución con el controlador de la ejecución en funcionamiento se realiza como se espera.

Con respecto a posibles mejoras, por la parte de la experimentación y la modelización, se podría haber medido la la influencia de la presencia del controlador de la ejecución, midiendo el agregado de instrucciones ejecutadas que supone la instanciación de este. También se podría haber refinado la interpretación extraída de las mediciones recogidas en la unidad de monitorización del rendimiento, observando si las instrucciones ejecutadas tienen en cuenta las operaciones del hipervisor, el coste de los cambios de contexto, de las llamadas al sistema, etc.

Del lado del controlador de la ejecución, una posible mejora es la recogida del CPI de las particiones críticas en diferentes periódos de tiempo de la ejecución, permitiendo así que el controlador de la ejecución evalúe el progreso de la ejecución de la partición en función del CPI exacto que se espera en cada parada de control, y no en función de un CPI medio calculado tras dividir el número de ciclos totales entre el número de instrucciones ejecutadas total, previniendo así paradas de la ejecución de las particiones no críticas en los puntos en los que el CPI crece y no es debido a la interferencia entre los procesos, sino más bien a la própia naturaleza de la ejecución de la partición crítica.

Bibliografía

[1] Instituto Universitario de Automática e Informática Industrial de la UPV Consultado en https://www.ai2.upv.es/en/home-2/

[2] Fent Innovative Software Solutions Consultado en https://fentiss.com/company/about-fentiss/

[3] E. A. Lee y S. A. Seshia. *Introduction to Embedded Systems - A Cyber-Physical Systems Approach* Segunda Edición. Massachusetts: MIT Press, 2017. ISBN 978-0-262-53381-2.

[4] Shibu Kizhakke Vallathai. *Introduction to Embedded Systems.* Nueva Delhi: McGraw Hill Education (India) Private Limited, 2009. ISBN 0-07-014589-4.

[5] Elecia White. *Making Embedded Systems.* Sebastopol: O'Reilly Media, Inc., 2017. ISBN 978-1-449-30214-6.

[6] Tammy Noergaard. *Embedded Systems Architecture - A Comprehensive Guide for Engineers and Programmers.* Burlington, E.E.U.U.: Newnes, Elsevier., 2005. ISBN 0-7506-7792-9.

[7] Hermann Koepetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications (Second Edition).* Springer New York Dordrecht Heidelberg London, 2011. ISBN 978-1-4419-8236-0 .

[8] S. C. Johnson and R. W. Butler. *Design for validation. IEEE Aerospace and Electronic Systems Magazine* vol. 7, no. 1, pp. 38-43, Jan. 1992, DOI 10.1109/62.127129.

[9] Timesys Corporation. *The Concise Handbook of Real-Time Systems.* TimeSys Corporation, Pittsburgh PA, E.E.U.U 2002.

[10] Steve Vestal. *Preemptive Scheduling of Multi-Criticality Systems with Varying Degrees of Execution Time Assurance. IEEE 28th IEEE International Real-Time Systems Symposium* 2007, DOI 10.1109/RTSS.2007.47.

[11] Alan Burns. *Multi-Model Systems - an MCS by any other name. Department of Computer Science, University of York, UK.*

[12] Mok, Aloysius and Feng, Xiang and Chen, Deji. *Resource partition for real-time systems. Real-TimeTechnology and Applications - Proceedings. 75 - 84.* DOI 10.1109/RTTAS.2001.929867.

[13] James H. Anderson, Sanjoy K. Baruah, and Bjorn B. Brandenbur *Multicore Operating-System Support for Mixed Criticality roc. of the Workshop on Mixed Criticality: Roadmap to Evolving UAV Certification, San Francisco, 2009*

[14] Rui Zhou and Alfons Crespo i Lorente *Partitioned System with XtratuM on PowerPC Tesina de Máster en Automática e Informática Industrial* 2009.

[15] Sobre el concepto de partición, Consultado en https://fentiss.com/products/hypervisor/partitioning-concept/.

[16] Xtratum Hypervisor for x86 *Reference Manual Fent innovative Software Solutions* October, 2013.

[17] P. J. Prisaznuk *Integrated modular avionics. Proceedings of the IEEE 1992 National Aerospace and Electronics Conference* NAECON 1992, 1992, pp. 39-45 vol.1, doi: 10.1109/NAECON.1992.220669.

[18] *Intel®64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture* Order Number: 253665-060US September 2016

[19] How many x86 instructions are there?, Consultado en https://fgiesen.wordpress.com/2016/08/25/how-many-x86-instructions-are-there/.